Peng Long and Xiaozhou Guo

Generative Adversarial Network
Principle and Practice

Peng Long
Beijing YouSan Educational Technology, Beijing, China

Xiaozhou Guo
• China Electronics Technology Group Corporation No. 54 Research Institute, Shijiazhuang, China

ISBN 978-981-96-9403-7 e-ISBN 978-981-96-9404-4
https://doi.org/10.1007/978-981-96-9404-4

Jointly published with China Machine Press Co.,Ltd.
ISBN of the Co-Publisher's edition: 978-7-111-71223-7

Contents

1 Generative Model

1.1 Unsupervised Learning and Generative Model

1.1.1 Supervised Learning and Unsupervised Learning

1.1.2 Discriminative and Generative Model

1.1.3 Unsupervised Generative Models

1.2 Explicit and Implicit Generative Model

1.2.1 Maximum Likelihood Estimation

1.2.2 Fully Visible Belief Network

1.2.3 Flow Model

1.2.4 Variation Autoencoder

1.2.5 Boltzmann Machine

1.3 Implicit Generative Model

References

2 Objective Function of GAN

2.1 GAN

2.1.1 General Understanding of GAN

2.1.2 GAN Model

2.1.3 Nature of GAN

2.2 LSGAN

2.2.1 Gradient Disappearance

2.2.2 LSGAN Design

2.3 EBGAN

2.4 fGAN

2.5 WGAN

2.5.1 Wasserstein Distance
2.5.2 WGAN Design
2.6 LS-GAN
2.7 GAN-GP
2.7.1 Weights Clipping
2.7.2 WGAN-GP
2.8 IPM
2.8.1 IPM Definition
2.8.2 GAN-Based IPM
2.8.3 IPM and f-Divergence
2.9 Other Objective Functions
2.9.1 RGAN
2.9.2 BEGAN
References
3 Training of GAN
3.1 Several Issues of Training
3.1.1 Gradient Vanishing Problem
3.1.2 Non-convergence
3.1.3 Mode Collapse
3.2 Annealing Noise
3.3 Spectral Normalization
3.3.1 Eigenvalue and Singular Value
3.3.2 Spectral Norm and the 1-Lipschitz Constraint
3.4 Consistent Optimization
3.4.1 Ordinary Differential Equations and Euler's Method
3.4.2 GAN Dynamical System

3.4.3 Consistent Optimization
3.5 GAN Training Techniques
3.5.1 Feature Matching
3.5.2 Historical Averages
3.5.3 Single-Side Label Smoothing
3.5.4 Virtual Batch Regularization
3.5.5 TTUR
3.5.6 Zero Center Gradient
3.5.7 Other Recommendations
3.6 Mode Collapse
3.6.1 Two Solutions for Mode Collapses
3.6.2 unrolledGAN
3.6.3 DRAGAN
3.6.4 MADGAN and MADGAN-Sim
3.6.5 VVEGAN
3.6.6 Mini-Batch Discriminator
References
4 Evaluation and Visualization of GAN
4.1 Evaluation Indicators
4.1.1 Requirements of Evaluation Indicators
4.1.2 IS Series
4.1.3 FID
4.1.4 MMD
4.1.5 Wasserstein Distance
4.1.6 1-Nearest Neighbor Classifier
4.1.7 GANtrain and GANtest

4.1.8 NRDS
4.1.9 Image Quality Measures
4.1.10 Average Log-Likelihood
4.2 GAN Visualization
4.2.1 Setting Up the Model
4.2.2 Training Model
4.2.3 Visualization Data
4.2.4 Two Demos
References
5 Image Generation
5.1 Image Generation Applications
5.1.1 Training Data Expansion
5.1.2 Data Quality Improvement
5.1.3 Content Creation
5.2 Deep Convolutional GAN
5.2.1 Deep Convolutional GAN
5.2.2 Thinking About DCGAN
5.3 Conditional GAN
5.3.1 Supervised Conditional GAN
5.3.2 Unsupervised Conditional GAN
5.3.3 Semi-supervised Conditional GAN
5.3.4 Complex Forms of Conditional Input
5.4 Multi-scale GAN
5.4.1 LAPGAN
5.4.2 Progressive GAN
5.5 Attribute GAN

5.5.1 Explicit Attribute GAN
5.5.2 Implicit Attribute GAN
5.6 Multi-discriminator and Generator GAN
5.6.1 Multi-discriminator GAN
5.6.2 Multi-generator GAN
5.7 Data Enhancement and Simulation GAN
5.7.1 Data Augmentation GAN
5.7.2 Data Simulation GAN
5.8 DCGAN Image Generation in Practice
5.8.1 Project Interpretation
5.8.2 Experimental Results
5.9 StyleGAN Face Image Generation in Practice
5.9.1 Project Profile
5.9.2 Model Interpretation
5.9.3 Use of Pre-trained Models
5.9.4 Summary
References

6 Image Translation
6.1 Basics of Image Translation
6.1.1 What Is Image Translation?
6.1.2 Types of Image Translation Tasks
6.2 Supervised Image Translation Model
6.2.1 Pix2Pix
6.2.2 Pix2PixHD
6.2.3 Vid2Vid
6.3 Unsupervised Image Translation Model

6.3.1 Unsupervised Model Based on Domain Migration and Domain Alignment

6.3.2 Unsupervised Model Based on Circular Consistency Constraints

6.4 Key Improvements to the Image Translation Model

6.4.1 Multi-Domain Transformation Network GAN

6.4.2 Enriching the Generation Mode of Image Translation Model

6.4.3 Adding Supervisory Information to the Model

6.5 Pix2Pix Model-Based Image Coloring Practice

6.5.1 Data Processing

6.5.2 Interpreting the Model Code

6.5.3 Model Training and Testing

6.5.4 Summary

References

7 Face Image Editing

7.1 Face Expression Editing

7.1.1 Expression Editing Issues

7.1.2 Key Point Controlled Expression Editing Model

7.2 Face Age Editing

7.2.1 Age Editing Issues

7.2.2 Conditional Adversarial AutoEncoder Based on Latent Space

7.3 Face Pose Editing

7.3.1 Pose Editing Issues

7.3.2 Pose Editing Model Based on 3DMM Model

7.4 Face Style Editing

7.4.1 Style Editing Issues

7.4.2 Stylized Model Based on Attention Mechanism
7.5 Face Makeup Editing
7.5.1 Makeup Editing Issues
7.5.2 GAN-Based Makeup Migration Algorithm
7.6 Face Swapping Algorithm
7.6.1 Identity Editing Issues
7.6.2 Deepfakes Face Swapping Algorithm
7.7 Generic Face Attribute Editing
7.7.1 Key Issues in StyleGAN Face Editing
7.7.2 Solving for Latent Coding Vectors
7.8 Hands-on Face Attribute Editing Based on StyleGAN Model
7.8.1 Face Reconstruction
7.8.2 Face Attribute Blending and Interpolation
7.8.3 Face Attribute Editing
7.8.4 Summary
References

8 Image Quality Enhancement
8.1 Image Noise Reduction
8.1.1 Image Noise Reduction Problem
8.1.2 GAN-Based Image Denoising Framework
8.2 Image Deblurring
8.2.1 Image Deblurring Problem
8.2.2 GAN-Based Image Deblurring Framework
8.3 Image Tone Mapping
8.3.1 Image Tone Mapping Problem
8.3.2 Image Tone Mapping Dataset

8.3.3 GAN-Based Image Tone Mapping Framework
8.4 Image Super-Resolution
8.4.1 Image Super-Resolution Problem
8.4.2 GAN-Based Image Super-Resolution Framework
8.5 Image Restoration
8.5.1 Image Restoration Basics
8.5.2 GAN-Based Image Restoration Framework
8.6 Face Super-Resolution Reconstruction Based on SRGAN
8.6.1 Project Interpretation
8.6.2 Model Training
8.6.3 Model Testing
8.6.4 Summary
References

9 3D Image and Video Generation
9.1 3D Image and Video Generation Applications
9.1.1 3D Image Generation Applications
9.1.2 Video Generation and Prediction Applications
9.2 3D Image Generation Framework
9.2.1 General 3D Image Generation Framework
9.2.2 2D to 3D Prediction Framework
9.3 Video Generation and Prediction Framework
9.3.1 Basic Video-GAN
9.3.2 Multi-Stage MD-GAN
9.3.3 MoCoGAN with Content and Action Separated
References

10 General Image Editing
10.1 Image Depth Editing
10.1.1 Depth and Depth of Field
10.1.2 Image Depth-of-Field Editing Framework
10.2 Image Fusion
10.2.1 Image Fusion Problem
10.2.2 GAN-Based Image Fusion Framework
10.3 Interactive Image Editing
10.3.1 Interactive Image Editing Problem
10.3.2 GAN-Based Interactive Image Editing Framework
10.4 Outlook
References
11 Adversarial Attack
11.1 Adversarial Attack and Defense Algorithm
11.1.1 Adversarial Attack
11.1.2 Commonattack Algorithm
11.1.3 Common Defense Algorithm
11.2 Adversarial Sample Generation Based on GAN
11.2.1 Perceptual-Sensitive GAN
11.2.2 Natural GAN
11.2.3 AdvGAN
11.3 GAN-Based Adversarial Attack Defense
11.3.1 APEGAN
11.3.2 DefenseGAN
11.4 AdvBox
11.4.1 Attacks on Classifiers

11.4.2 Gaussian Noise Adversarial Defense
11.4.3 DataPoison
11.4.4 Face Recognition Model Deception
References
12 Speech Signal Processing
12.1 GAN-Based Speech Enhancement
12.1.1 Project Introduction
12.1.2 SEGAN Model
12.1.3 SEGAN Training and Testing
12.2 GAN-Based Speech Transformation
12.2.1 Project Introduction
12.2.2 WORLD Speech Synthesis Tools
12.2.3 cycleGAN-VC2 Model
12.2.4 cycleGAN-VC Training
12.2.5 cycleGAN-VC Model Testing
12.3 GAN-Based Speech Generation
12.3.1 Project Introduction
12.3.2 waveGAN Model
12.3.3 waveGAN Training
References

P. Long, X. Guo, *Generative Adversarial Network*
https://doi.org/10.1007/978-981-96-9404-4_1

1. Generative Model

Peng Long[1] ✉ and Xiaozhou Guo[2]
(1) Beijing YouSan Educational Technology, Beijing, China
(2) • China Electronics Technology Group Corporation No. 54 Research Institute, Shijiazhuang, China

Abstract
This chapter provides a detailed introduction to unsupervised learning, supervised learning, and semi-supervised learning, including the definitions, essences, common scenarios, and frequently used models of different learning methods. Subsequently, generative models and discriminative models are respectively introduced respectively within the scope of supervised learning, covering their definitions, differences, common models, etc. Then, the concept and learning approach of unsupervised generative models are presented. In the second part of this chapter, we classify generative models into two types, explicit generative models and implicit generative models, according to the way generative models handle the probability density function. For explicit generative models, the principle of the maximum likelihood method is described in detail and is divided into two categories: tractable probability density functions and approximate methods. In the first category, FVBN series models are listed, including PixelRNN, PixelCNN, NADE, and flow models. In the second category, variational autoencoders and restricted Boltzmann machines are introduced. In the third part, the implicit generative model is introduced taking GAN as an example, and GAN is compared with other generative models.

Keywords Generative model – Unsupervised learning – Implicit generative model – GAN

The in-depth research on unsupervised generative models has greatly contributed to the development of deep learning. Nowadays, GAN, VAE, flow model, and other such models are widely discussed as typical representatives of unsupervised generative models. Chapter 1 introduces the scope and core content of this book. Section 1.1.1 provides an overview of unsupervised learning, supervised learning, and semi-supervised learning, including their definitions, essences, common scenarios, and frequently used models. Section 1.1.2 introduces generative model and discriminative model within the context of supervised learning, covering their definitions, differences, and common models. Section 1.1.3 introduces the core concept of this book—unsupervised generative models, along with their learning approaches. Based on how generative models handle probability density function, they can be divided into explicit generative model and implicit generative model. Section 1.2.1 first elaborates on the principle of maximum likelihood estimation, which is the fundamental principle of all explicit model. It then further categorizes explicit generative model into exact method and approximate method, based on whether they use exact inference or approximate inference to calculate the likelihood function. Section 1.2.2 introduces FVBN series models, including PixelRNN, PixelCNN, and WaveNet, as examples of the first category of method. Section 1.2.3 explains flow model, represented by NICE, within the first category of method. Section 1.2.4 provides a detailed explanation of latent variable generative models, represented by Variation Autoencoders, within the second category of methods. Section 1.2.5 introduces the modeling approach of Boltzmann machines. Section 1.3 introduces the modeling characteristics of implicit generative models using GAN as an example and compares GAN with other generative models. While introducing the principles of various models, this chapter also provides related code, which is believed to help readers establish a basic understanding and knowledge of generative model.

- Section 1.1: Unsupervised generative model
- Section 1.2: Explicitly generative model
- Section 1.3: Implicit generative model

1.1 Unsupervised Learning and Generative Model

Before introducing unsupervised learning and generative model, let's first understand what supervised learning and unsupervised learning are.

1.1.1 Supervised Learning and Unsupervised Learning

Supervised learning involves learning a model (or a mapping function) that can generate a corresponding predicted output for any given input. The input to the model is a random variable X, and the output is also a random variable Y. Each specific input is an instance represented by a feature vector x and the corresponding output of the instance is represented by the vector y. The set of all possible input vectors is called the feature space (input space), and the set of all possible output vectors make up the output space. Generally, the size of the output space is much smaller than the input space. The essence of supervised learning is to learn the statistical laws of the mapping from input to output.

We list three common supervised learning tasks: regression, classification, and labeling, which mainly differ in the types of variable values.

(1) When both the input and output variables are continuous, it corresponds to a regression task, which is primarily used to learn the numerical mapping relationship between the input and output variables. Common regression tasks include price prediction, trend prediction, etc. Common machine learning models for handling regression tasks include least squares regression, nonlinear regression, etc.

(2) Regardless of whether the input variable is discrete or continuous, when the output variable is a finite number of discrete values, it corresponds to the classification task. Classification task is the most widely discussed and applied task by people. Common classification tasks include image category recognition, audio classification, text classification, etc. Common machine learning models for handling classification tasks include k-nearest neighbors, naive Bayes, decision trees, logistic regression, support vector machines, neural networks, etc.

(3) When both the input and output variables are variable sequences, it corresponds to a labeling task, which is an extension of classification problems and is used to learn the mapping relationship between input sequences and output sequences. Typical labeling tasks include part-of-speech labeling and information extraction in natural language processing. Common machine learning models for handling labeling tasks include hidden Markov models and conditional random fields.

The biggest difference between unsupervised learning and supervised learning is the presence or absence of labels. In supervised learning, the task of training the model is to learn the mapping from input feature x to the label y. While in unsupervised learning, only the feature vector of the sample x exists. Therefore, the task of unsupervised learning is to "dig" deeper into the data, which is essentially to learn the statistical patterns or underlying structures in the data. In-depth research on unsupervised learning has played a crucial role in the revival of deep learning.

We list three common unsupervised learning tasks: dimensionality reduction, clustering, and probability model estimation.

(1) The dimensionality reduction task is mainly used to deal with the problem of high dimensionality of data. Excessively large feature dimensions of real data can easily reduce the fit and usability of the model. We can use dimensionality reduction algorithms to "compress" high-dimensional data into low-dimensional vectors, thereby improving data usability. Common algorithms include principal component analysis, factor analysis, latent Dirichlet, etc. Early autoencoders can also be used for data dimensionality reduction.

(2) Clustering task mainly assigns samples to categories based on certain rules, i.e., by measuring the distance, density, and other indicators between samples, samples that are "close" in relation are grouped into the same category, thereby achieving automatic classification of samples. Common algorithms include hierarchical clustering, k-means clustering, spectral clustering, etc.

(3) In the probability model estimation task, for a probability model that can generate samples, we use the samples to learn the structure and parameters of the probability model, so that the samples generated by the probability model are most similar and realistic to the training samples. One of the probability

by the probability model are most similar and realistic to the training samples. One of the probability model estimation tasks is to estimate the probability density function $p(X)$ of the random variable X.

The common algorithms used are maximum likelihood estimation, adversarial generative networks, variation autoencoder, etc. This part is very rich and is the core content of this book.

Compared to unsupervised learning, supervised learning not only possesses additional label information but also requires test samples. In other words, the machine learning model learns "patterns" from the training set and then applies these "patterns" to the test set to evaluate the model's effectiveness. Furthermore, unsupervised learning offers better scalability than supervised learning, as it can achieve the training objectives while additionally learning representations of samples that can be directly utilized for other tasks.

Semi-supervised learning falls between supervised and unsupervised learning, where only a small portion of the training samples have label information, while the majority lack label information. Semi-supervised learning includes two types of models, transductive and inductive. Transductive semi-supervised learning only processes the given training data, it uses the samples with and without category labels in the training dataset for training and predicts the label information of unlabeled samples among them; inductive semi-supervised learning not only predicts the labels of unlabeled samples in the training dataset, but mainly predicts the labels of unknown samples, the difference between the two is needed to predict whether the labeled samples appear in the training dataset. Semi-supervised learning is generally used in four types of learning scenarios: semi-supervised classification, semi-supervised regression, semi-supervised clustering, semi-supervised dimensionality reduction, etc.

1.1.2 Discriminative and Generative Model

To avoid confusion among readers regarding several common concepts, this section will introduce discriminative model and generative model within the scope of supervised learning only [1]. According to Sect. 1.1.1, supervised learning is learning a model and then using that model to predict the corresponding output for a given input, and we can write the model in the form of a function $Y = f(X)$ or in the form of a conditional probability distribution $p(Y|X)$ and classify it into discriminative and generative model according to how the conditional probability distribution is computed.

In the discriminative model, we directly model $p(Y|X)$, which attempts to describe the distribution of label information Y given the input feature X. The typical discriminative models include: k-nearest neighbor, perceptron machine, decision tree, logistic regression, and conditional random field, etc. Discriminative models directly model conditional probability, which cannot reflect the probability characteristics of the training data itself. However, taking classification problems as an example, discriminative models learn the differences between data from different classes while searching for the optimal classification hyperplane. Additionally, discriminative models can abstract and reduce the dimensionality of data to various degrees, thereby simplifying the learning problem and improving learning accuracy.

In the generative model, the data feature X and label Y of the joint distribution $p(X, Y)$ are modeled together, and then using the conditional probability formula can calculate $p(Y|X)$, which is shown as below:

$$p(Y|X) = \frac{p(X,Y)}{p(X)} \tag{1.1}$$

In practice, we usually transform the joint distribution into a form that is easy to solve for:

$$p(Y|X) = \frac{p(X|Y)P(Y)}{p(X)} \tag{1.2}$$

Where $p(Y)$ is the prior probability distribution of label information Y, which describes the probability in the absence of X. The probability distribution $p(Y|X)$ is the posterior probability of the label Y, which describes the probability distribution in the presence of explicit sample features X. Typical generative models include the naive Bayes and the hidden Markov model, etc. In the naive Bayes, we learn the prior probability distribution $p(Y)$ and the conditional probability distribution $p(X|Y)$ through the training dataset, then we can obtain the joint probability distribution $p(X, Y)$. In the hidden Markov model, we learn the initial probability distribution, state transition probability matrix, and observation probability matrix from the training set to obtain a model representing the joint distribution of state sequences and observation sequences.

Generative model learn the joint distribution directly, which allows them to better represent the data distribution and reflect the similarity between data from the same class. When the sample size is relatively large, the generative model tends to converge better to the true model and its convergence is faster. In addition, generative model can handle cases containing hidden variables, which is beyond the capability of discriminative model. Generative model can also detect certain outliers by computing the edge distribution $P(X)$. However, in practice, the computational overhead of the generative model is generally high and in most cases it is not as effective as the discriminative model.

1.1.3 Unsupervised Generative Models

According to the previous two sections, generating a model implies that the joint distribution of input feature X and label information Y is established, while unsupervised learning implies that there is no label information. In the unsupervised generative model, it is desired to model the probability density function $p(X)$. Suppose there exists a model consisting of N training samples $\{x^{(1)}, x^{(2)}, \ldots, x^{(N)}\}$, then a probability model $\widehat{p}(X)$ can be trained using the training dataset. After the training process is completed, the probability model $\widehat{p}(X)$ should be close to the probability density function $p(X)$, and we can then "generate" samples from the probability model $\widehat{p}(X)$.

Unsupervised generative models have been a popular direction in deep learning in recent years, with a long history of development [2]. In classical statistical machine learning, here has been extensive discussion on the main problem of generative models—estimating probability density functions. Methods for estimating probability density functions are mainly divided into parametric estimation and nonparametric estimation. Parametric estimation typically involves a known mathematical model for the problem being studied (such as a mixture Gaussian distribution and Bernoulli distribution) and then uses samples to estimate the unknown parameters in the model. Common estimation methods include maximum likelihood estimation, Bayesian estimation, maximum posteriori estimation, etc. Nonparametric estimation has no prior knowledge of mathematical models and directly uses sample estimation to estimate mathematical models. Common methods include histogram estimation, kernel probability density estimation (Parzen window), k-nearest neighbor estimation, etc.

Similarly, generative models based on neural network methods have also been studied for a long time. For example, in the 1980s, Hinton already used Boltzmann machine [3] to learn arbitrary probability distributions of binary vectors. In neural network methods, many very good models have emerged, such as deep belief networks [4], neural autoregressive networks [5, 6], deep Boltzmann machines [7], and flow model. Among them, the Variational Autoencoder (VAE) proposed in 2013 and the GAN proposed in 2014 are two of the most outstanding representatives. It should be noted that most deep generative models, including GAN, still fall into the category of parametric estimation, i.e., using samples to estimate the weight parameters of the neural network model.

The research on generative models is of great importance to the development of artificial intelligence technology. It can not only produce realistic images, videos, text or speech, etc., but also achieve satisfactory results in areas such as image conversion, super-resolution images, target detection, and text-to-image. Generative models are closely related to reinforcement learning, semi-supervised learning, multimodal output problems, etc. In addition, the training and sampling of generative models are excellent tests of our ability to express and process high-dimensional probability distribution problems. This book will focus on GAN.

1.2 Explicit and Implicit Generative Model

The realm of generative model is diverse and rich, and we can classify them based on how they handle probability density function. Broadly speaking, models that explicitly handle probability density functions are referred to as explicit generative model, while those that implicitly handle them are called implicit generative model, as illustrated in Fig. 1.1. Explicit generative model, due to the need for training, require precise or approximate expressions for the likelihood function of the samples. In contrast, implicit generative model indirectly control the probability distribution through samples, without explicitly involving the likelihood function during the training process， which is an indirect way of controlling the probability density. Both explicit and implicit generative models aim to model $p(x)$, but explicit model directly optimize $p(x)$, which can be challenging. Implicit models, on the other hand, circumvent the difficulty of directly facing $p(x)$ by optimizing $p(x)$ indirectly through the samples generated by $p(x)$. In this

section, we will start with the most basic method, maximum likelihood estimation, and delve deeply and comprehensively into some representative explicit and implicit generative model, such as fully visible belief network, variation autoencoders, and generative adversarial network.

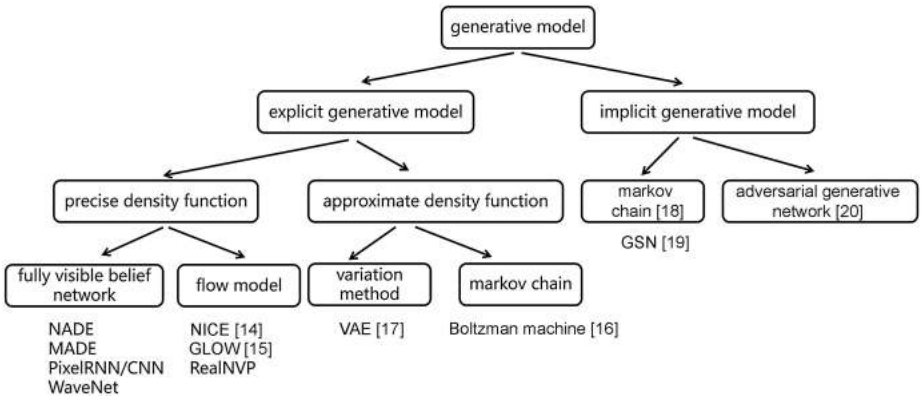


Fig. 1.1 Generative model classification

In generative model, the probability density function $p(x)$ has always played a central position. For a batch of training samples $\{x^{(1)}, x^{(2)}, \ldots, x^{(N)}\}$ obtained from $p_{data}(x)$ independently (note that we require the data in the training sample set to be independently and identically distributed), our goal is to use these training data to train a generative model $p_g(x)$, and the generative model can learn the distribution of the data $p_{data}(x)$ either explicitly or implicitly, or obtain $p_{data}(x)$ approximate expression ($p_{data}(x) \approx p_g(x)$). Then, during the forward inference process, we can obtain a batch of samples by explicitly or implicitly sampling from $p_g(x)$, and the obtained samples (approximately) match the probability distribution $p_{data}(x)$.

1.2.1 Maximum Likelihood Estimation

Since the core of generative model is to solve p (x), let's consider a simple question: for a collection of samples, can we estimate $p(x)$ by directly counting the number of samples? This sounds feasible. We only need to count the frequency of each sample, then normalize the probability by dividing by the total number of samples in the collection, and finally obtain a histogram representation of $p(x)$, as shown in Fig. 1.2. But when the sample dimension is large, the curse of dimensionality arises. For the images in the MNIST dataset, the dimension is 28×28=784, and the value of each pixel position can be taken as 0 or 1, which means that the probability distribution contains a total of $2^{784} \approx 10^{236}$ sample points, corresponding to approximately 10^{236} probability values that need to be estimated. In fact, any training dataset can only contain a very small portion of the sample points in the entire sample space, and each image can only affect the probability of one sample point, meanwhile having no impact on the probability of other similar sample points. Therefore, this counting statistical model does not have generalization performance. In practical operation, it is impossible to store the probability value for every sample point, so we instead use a parameterized probability density function $p_\theta(x)$, where θ is the parameter of the model.

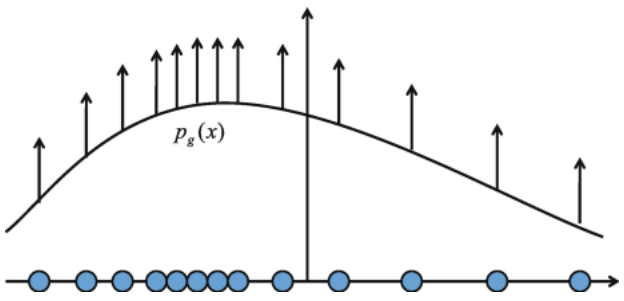


Fig. 1.2 Principle of the maximum likelihood method

We first introduce generative model that use maximum likelihood estimation method. A thorough understanding of the maximum likelihood principle is crucial for understanding generative model. Note that not all generative models employ maximum likelihood estimation. Some generative models do not use it by default, but modifications can be made to enable its use (GAN belong to this category).

Maximum likelihood estimation is a method of estimating the parameters of a probability model [8]. For example, with a dataset containing N samples $\{x^{(1)}, x^{(2)}, ..., x^{(N)}\}$, each sample in the dataset is obtained from some unknown probability distribution $p_{data}(x)$ independently, and if we already know the expression form of $p_g(x)$, but still contains the unknown parameter θ, then the problem becomes: how to use the dataset to estimate the unknown parameter θ in $p_g(x)$. For example, $p_g(x)$ is a Gaussian distribution with undetermined mean and variance parameters, how can the sample be used to estimate the exact values of the mean and variance?

In the maximum likelihood method, the likelihood function $L(\theta)$ is first calculated using all samples:

$$L(\theta) = \prod_{i=1}^{N} p_g\left(x^{(i)}; \theta\right) \tag{1.3}$$

The likelihood function $L(\theta)$ is a function of the model parameter θ, and when the different parameters θ are chosen, the value of the likelihood function is different. It describes the value of the probability of all samples in the dataset given the current parameters. A plain idea is that the probability of generating all samples in the dataset is maximum under the best model parameters, i.e.,

$$\theta_{ML} = \operatorname{argmax}_{\theta} L(\theta) \tag{1.4}$$

However in practice, in computers, the results of multiplying multiple probabilities are not easy to store, for example, the problem of numerical underflow may occur during the calculation, i.e., a relatively small number that is close to 0 is rounded to 0. We can mitigate this problem by taking the logarithm of the likelihood function, i.e., $\log[L(\theta)]$ and still solve for the best model parameters θ_{ML}, so that the log-likelihood function is maximized, i.e.,

$$\theta_{ML} = \operatorname{argmax} \log[L(\theta)] \tag{1.5}$$

It can be shown that the two are equivalent, but taking the likelihood function logarithmically converts the probability product form into a logarithmic summation form, which greatly facilitates the calculation. Expanding it, we have

$$\theta_{ML} = \operatorname{argmax} \sum_{i=1}^{N} \log p_g\left(x^{(i)}; \theta\right) \tag{1.6}$$

It can be found that when using the maximum likelihood estimation, each sample x_i is expected to pull up the corresponding model probability value $p_g(x^{(i)}; \theta)$, as shown in Fig. 1.2, but since the density function of all samples $p_g(x^{(i)}; \theta)$ of all samples must sum to 1, it is not possible to raise all sample points to the maximum probability. The probability density function of one sample point being raised will inevitably cause the function values of other points to be lowered, eventually reaching an equilibrium state.

We can also divide the above equation by N and we can see that the objective of the maximum likelihood method is to maximize the probability of the sample under the empirical distribution $\hat{p}_{\text{data}}$, that is

$$\theta_{ML} = \operatorname{argmax} \mathbb{E}_{\hat{p}_{\text{data}}}[\log p_g(x; \theta)] \tag{1.7}$$

Another understanding of maximum likelihood estimation is that the essence of maximum likelihood estimation is in minimizing the value of KL divergence between the empirical distribution on the training dataset $\hat{p}_{\text{data}}(x)$ and the model distribution $p_g(x; \theta)$

$$\theta_{ML} = \operatorname{argmin} D_{KL}\left(\hat{p}_{\text{data}} \parallel p_g\right) \tag{1.8}$$

The expression for KL dispersion is

$$D_{KL}\left(\widehat{p}_{\text{data}} \parallel p_g\right) = E_{\widehat{p}_{\text{data}}}\left[log\ \widehat{p}_{\text{data}}(x) - \log\ p_g(x;\theta)\right] \tag{1.9}$$

Since the θ value is independent of the first term, then only the second term is considered, that is

$$\theta_{ML} = \text{argmax} \sum_{i=1}^{N} log\ p_g\left(x^{(i)};\theta\right) \tag{1.10}$$

It can be found that above two are identical, which means that the maximum likelihood estimate is the hope $p_g(x;\theta)$ and $p_{\text{data}}(x)$ as similar as possible, preferably so similar that there is no difference (KL value of 0), which is consistent with the idea of generative model. However, it is generally impossible for the generative model to know the form of the expressions $p_g(x;\theta)$ in advance. The actual generative model are very complex and often have no any prior knowledge of $p_g(x;\theta)$, only some formal assumptions or approximations can be used.

Many generative models can be trained using the principle of maximum likelihood. As long as the parameter θ of the likelihood function $L(\theta)$ is obtained, all that is needed is to maximize this function $L(\theta)$. The difference among various models lies in how they express or approximate the likelihood function $L(\theta)$. The left branches of Fig. 1.1 are all explicit generative model, where the fully visible belief network model makes formal assumptions about the $p_g(x;\theta)$, and the flow model is given by defining a nonlinear transformation expression, both of these models actually give the likelihood function $L(\theta)$. The variation autoencoder model, on the other hand, uses an approximate approach and only obtains a lower bound for the log-likelihood function log[$L(\theta)$]. The Boltzmann machine uses a Markov chain to approximate the gradient of the likelihood function. Next, we will introduce each of these models and discuss their advantages and disadvantages.

1.2.2 Fully Visible Belief Network

In a fully visible belief network, there are no unobservable latent variables. The probability expression for high-dimensional observed variables is decomposed dimensionally using the chain rule. That is, for an n-dimensional observed variable x, its probability expression is given by

$$p(x) = \prod_{i=1}^{n} p(x_i \mid x_{i-1}, x_{i-2}, \ldots, x_1) \tag{1.11}$$

An autoregressive network is the simplest type, where each dimension of the observed variable x_i forms a node in the probability model. All these nodes $\{x_1, x_2, \ldots, x_n\}$ collectively constitute a fully directed graph, in which any two nodes are connected, as shown in Fig. 1.3.

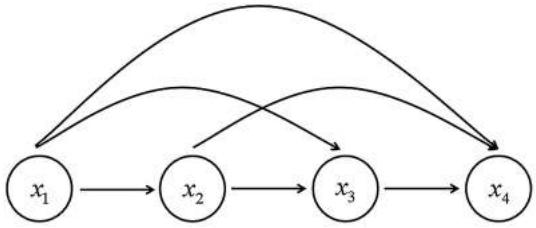


Fig. 1.3 Autoregressive network

Given the chain decomposition relationship of random variables in autoregressive networks, then the core problem becomes how to express the conditional probability $(x_i \mid x_{i-1}, x_{i-2}, \ldots, x_1)$. The simplest model is the linear autoregressive network [9], i.e., each conditional probability is defined as a linear model, using linear regression model for real-valued data, for example

$$p(x_i \mid x_{i-1}, x_{i-2}, \ldots, x_1) = w_1 x_1 + w_2 x_2 + \ldots + w_{i-1} x_{i-1} \tag{1.12}$$

And for binary data, a logistic regression model is used, and for discrete data, a softmax regression model is employed. The specific calculation process is shown in Fig. 1.4. However, the linear model has limited capacity and does not have enough ability to fit the function. In the neural autoregressive network, a neural network is used instead of the linear model, which can increase the capacity arbitrarily and can theoretically fit any joint distribution. Neural autoregressive networks also utilize feature reuse techniques. For example, the hidden abstract features h_i learned by the neural network from the observed variable x_i are not only used when calculating $p(x_{i+1}|\, x_i, x_{i-1}, \ldots, x_1)$ time, but also in the computation of $p(x_{i+2}|\, x_{i+1}, x_i, \ldots, x_1)$, and their computational diagrams are shown in Fig. 1.5. Moreover, the model does not need to use separate neural network representations for each conditional probability calculation and can integrate all neural networks into one, so that as long as the abstract features h_i are designed to depends only on $x_1, x_2, \ldots, x_i$. The current neural autoregressive density estimator [10] is the most representative scheme in neural autoregressive networks, which is a scheme that introduces parameter sharing in neural autoregressive networks, i.e., the weight parameters from the observed variable x_i to any hidden abstract feature $h_{i+1}, h_{i+2}, \ldots$ are shared. Totally, the neural autoregressive density estimator that uses deep learning techniques such as feature reuse and parameter sharing has a very good performance.

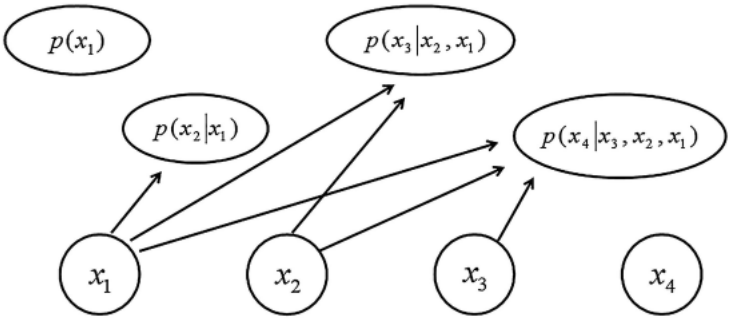


Fig. 1.4 Calculation diagram of linear autoregressive network

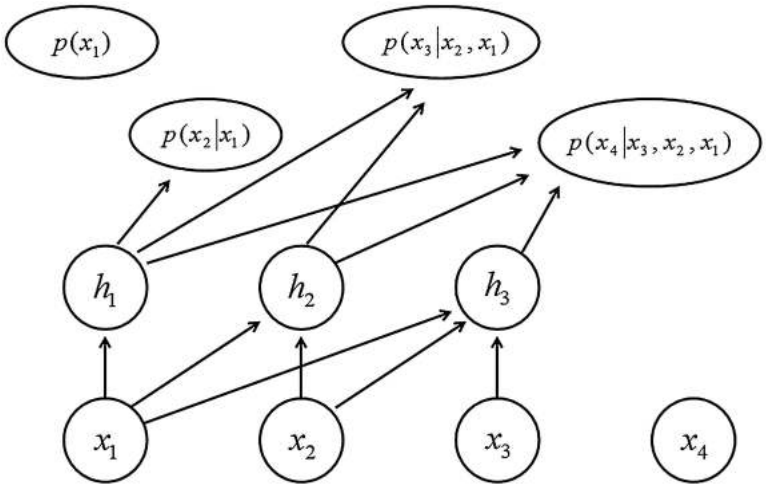


Fig. 1.5 Computational diagram of neural autoregressive network

WaveNet [1] is a speech generative model proposed by Google for autoregressively generating speech sequences. The main computational module it employs is dilated causal convolution, with the core idea being to generate the speech information at the current time step based on the speech information from previous time steps. Dilated causal convolution is illustrated in Fig. 1.6. In each layer of the one-dimensional convolutional neural network, its output depends on the information from the tth and $(t - d)$th time steps of the previous layer, where d is the dilation factor. For example, the dilation factors for the first, second, and third hidden layers are 1, 2, and 4, respectively. It should be noted that the speech at the current time step does not establish a connection with all previous time steps, so the expression for the sample probability differs slightly from equation above. The specific dependency relationship is determined by the

configuration of the convolution layers. When outputting the speech information, a 256-dimensional probability distribution is first obtained through the μ-law quantization method, with each dimension representing a speech signal value. Then, the signal at the current time step is sampled from this probability distribution.

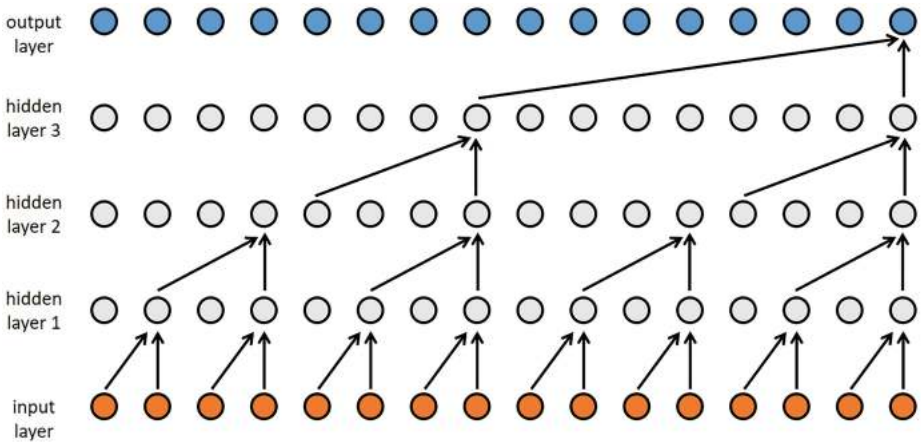


Fig. 1.6 WaveNet dilated causal convolution

PixelRNN and PixelCNN [2] also belong to fully visible belief networks, and as the names show, these two models are generally used for image generation. They decompose the probability $p(x)$ by pixels into n product of conditional probabilities, where n is the number of pixel points of the image. This means that a conditional probability is defined at each pixel point to express the dependencies between pixels, and these conditional probabilities are learned using RNNs or CNNs, respectively. In order to discretize the output, the last layer of RNN or CNN is usually set as a softmax layer to represent the probability of different pixel values in its output. In PixelRNN, the pixels are generally defined to be generated sequentially from the top-left corner along the right and down directions, as shown in Fig. 1.7. Once the dependency order of the nodes is decided, the expression for the log-likelihood of the sample can be obtained, and during subsequent model training, it only needs to be maximized.

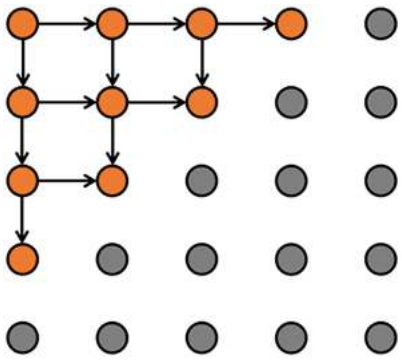


Fig. 1.7 PixelRNN generates pixel order

PixelRNN may have unbounded dependency range within its receptive field because the pixel value of the position to be sought depends on the pixel values of all previously known pixel points, which requires a significant computational cost. PixelCNN uses standard convolutional layers to capture the bounded receptive field, which is faster to train than PixelRNNN. In PixelCNN, the pixel value of each position is only related to its values of the surrounding known pixel points, as shown in Fig. 1.8. The upper part represents known pixels, while the lower part represents unknown pixels. When calculating the pixel value at the current position, all known pixel values within the boxed area are passed to the CNN, and the final softmax output layer of the CNN expresses the probabilities of different pixel values at the central rectangular position. Here, a mask matrix composed of 0 and 1 should be used to erase the gray pixels within the boxed area. PixelRNN and PixelCNN [11, 12] still have very many improved models since their inception. For

example, to eliminate blind spots when generating pixels, GatedPixelCNN [3] splits the receptive field into horizontal and vertical directions, further enhancing the generation quality. However, since these models generate images pixel by pixel, with dependencies between pixels and seriality, sampling efficiency in practical applications is difficult to guarantee. Also, this is a common issue among many FVBN-type models.

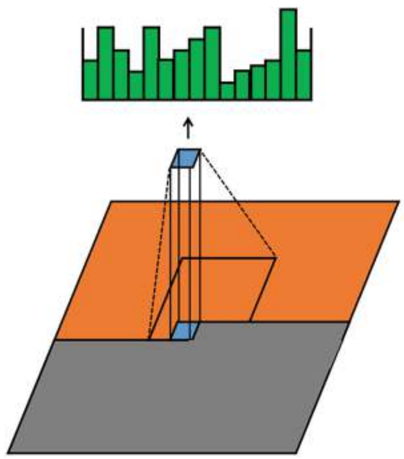


Fig. 1.8 PixelCNN principle

The PixelCNN model is a relatively easy-to-understand model of the fully visible belief network, and the following is the core code of the PixelCNN model:

```
# Two-dimensional mask convolution
class MaskedConv2d(nn.Conv2d).
def __init__(self, mask_type, *args, **kwargs).
super(MaskedConv2d, self). __init__(*args, **kwargs)
assert mask_type in {'A', 'B'}
self.register_buffer('mask', self.weight.data.clone())
bs, o_feature_dim, kH, kW = self.weight.size()
self.mask.fill_(1)
self.mask[:,:, kH // 2, kW // 2 + (mask_type == 'B'):] = 0
self.mask[:,:, kH // 2 + 1:] = 0

def forward(self, x).
self.weight.data *= self.mask
return super(MaskedConv2d, self).forward(x)
# PixelCNN neural network
network = nn.Sequential(
MaskedConv2d('A', 1, feature_dim, 7, 1, 3, bias=False),
nn.BatchNorm2d(feature_dim), nn.ReLU(True).
MaskedConv2d('B', feature_dim, feature_dim, 7, 1, 3, bias=False),
nn.BatchNorm2d(feature_dim), nn.ReLU(True).
MaskedConv2d('B', feature_dim, feature_dim, 7, 1, 3, bias=False),
nn.BatchNorm2d(feature_dim), nn.ReLU(True).
MaskedConv2d('B', feature_dim, feature_dim, 7, 1, 3, bias=False),
nn.BatchNorm2d(feature_dim), nn.ReLU(True).
MaskedConv2d('B', feature_dim, feature_dim, 7, 1, 3, bias=False),
nn.BatchNorm2d(feature_dim), nn.ReLU(True).
MaskedConv2d('B', feature_dim, feature_dim, 7, 1, 3, bias=False),
nn.BatchNorm2d(feature_dim), nn.ReLU(True).
MaskedConv2d('B', feature_dim, feature_dim, 7, 1, 3, bias=False),
nn.BatchNorm2d(feature_dim), nn.ReLU(True).
```

```
MaskedConv2d('B', feature_dim, feature_dim, 7, 1, 3, bias=False),
nn.BatchNorm2d(feature_dim), nn.ReLU(True).
nn.Conv2d(feature_dim, 256, 1))
network.to(device)

train_data = data.DataLoader(datasets.MNIST('data', train=True,
download=True, transform=transforms.ToTensor()).
batch_size=train_batch_size, shuffle=True, num_workers=1, pin_memory=True)
test_data = data.DataLoader(datasets.MNIST('data', train=False,
download=True, transform=transforms.ToTensor()).
batch_size=train_batch_size, shuffle=False, num_workers=1, pin_memory=True)

optimizer = optim.Adam(network.parameters())
if __name__ == "__main__".
for epoch in range(epoch_number).
# Training
cuda.synchronize()
network.train(True)

for input_image, _ in train_data.
time_tr = time.time()

input_image = input_image.to(device)
output_image = network(input_image)
target = (input_image.data[:, 0] * 255).long().to(device)
loss = F.cross_entropy(output_image, target)

optimizer.zero_grad()
loss.backward()
optimizer.step()
print("train: {} epoch, loss: {}, cost time: {}".format(epoch, loss.item(),
time.time() - time_tr))
cuda.synchronize()

# Testing
with torch.no_grad().
cuda.synchronize()
time_te = time.time()
network.train(False)
for input_image, _ in test_data.
input_image = input_image.to(device)
target = (input_image.data[:, 0] * 255).long().to(device)
loss = F.cross_entropy(network(input_image), target)
cuda.synchronize()
time_te = time.time() - time_te
print("test: {} epoch, loss: {}, cost time: {}".format(epoch, loss.item(),
time_te))

# Generate samples
with torch.no_grad().
image = torch.Tensor(generation_batch_size, 1, 28, 28).to(device)
image.fill_(0)
network.train(False)
for i in range(28).
for j in range(28).
out = network(image)
probs = F.softmax(out[:,:, i, j]).data
```

```
image[:,:, i, j] = torch.multinomial(probs, 1).float() / 255.
utils.save_image(image, 'generation-image_{:02d}.png'.format(epoch),
nrow=12, padding=0)
```

1.2.3 Flow Model

Flow model is a generative model with a relatively straightforward idea but are actually not easy to construct. It utilizes techniques such as invertible nonlinear transformations to enable the exact computation of likelihood function. Compared to FVBN, flow model introduces the concept of latent variable and establish a deterministic mapping relationship between latent variables and observed variables.

Let's first introduce the basic idea of flow model. For a latent variable z with a simple distribution (such as a Gaussian distribution) denoted by $p_z(z)$, then if there exists a continuous, differentiable, invertible nonlinear transformation $g(z)$, which converts the simple distribution of the latent variable z into a complex distribution over samples x, and we denote the inverse transformation of $g(z)$ as $f(x)$, this is $x = g(z)$ and $z = f(x)$, then the exact probability density function $p_x(x)$ of the sample x is

$$p_x(x) = p_z(z) \left|\det \left(\frac{\partial f}{\partial x}\right)\right| \tag{1.13}$$

Note that the nonlinear transformation $g(z)$ causes a deformation of the space, i.e., $p_x(x) \neq p_z(f(x))$, and there is $p_z(z)dz = p_x(x)dx$. As to invertible matrix, there are

$$det\left(A^{-1}\right) = \det\left(A\right)^{-1} \tag{1.14}$$

Then, we get

$$p_x(x) = p_z(z) \left|\det \left(\frac{\partial g}{\partial z}\right)\right|^{-1} \tag{1.15}$$

If the above model is successfully constructed, the samples are generated by simply sampling from the simple distribution $p_z(z)$ and then transform it to $x = g(z)$.

In order to train the nonlinear independent component estimation model, we must calculate the probability density function of the sample $p_x(x)$. Analyzing the above equation, the probability density function $p_x(x)$ requires the calculation of $p_z(z)$ and the absolute value of the determinant of the Jacobi matrix. For the former, to construct an $g(z)$ (inverse transform of $f(x)$) so that when given a sample x is given, it must be easy to obtain its corresponding hidden variable by $z = f(x)$, so $p_z(z)$ is usually designed as a simple distribution that is easy to compute; for the latter, it is necessary to design some special form so that the determinant of the Jacobi matrix is easy to compute. In addition, the invertibility of the transformation requires that the samples x and hidden variables z have the same dimensionality. In summary, the generative model needs to be carefully designed as an easy-to-handle and flexible bijection model so that the inverse transform $f(x)$ exists and the determinant of the corresponding Jacobi matrix can be computed.

In practical flow model, the nonlinear mapping $f(\boldsymbol{x})$ is composed of multiple mapping functions $f_1, f_2 ... f_k$ combined together, that is $\boldsymbol{z} = f_k \circ ... \circ f_1(\boldsymbol{x})$ and $x = f_1^{-1} \circ \ldots \circ f_k^{-1}(\boldsymbol{z})$. The meaning of "flow" in a flow model is that a variable continuously flows through multiple transformations and ultimately "forms" another variable. Correspondingly, the determinant of the Jacobian matrix can be decomposed into

$$\left|\det \left(\frac{\partial f}{\partial x}\right)\right| = \left|\det \left(\frac{\partial f_k}{\partial \boldsymbol{f_{k-1}}}\right)\right| \left|\det \left(\frac{\partial f_{k-1}}{\partial \boldsymbol{f_{k-2}}}\right)\right| \ldots \left|\det \left(\frac{\partial f_1}{\partial x}\right)\right| \tag{1.16}$$

Here, we introduce two very basic and simple flows: affine flow and elementwise flow. In affine flow, the nonlinear mapping $f(x) = A^{-1}(x - b)$ maps the sample x to a standard Gaussian distribution, where the learnable parameter A is a non-singular square matrix and b is a bias vector. The sampling process involves first sampling to obtain z, and then obtaining the sample based on $x = Az + b$. The Jacobian matrix in the s affine flow model is A^{-1}, and the difficulty of calculating the determinant depends on the number of

dimensions in the matrix. In the elementwise flow, mapping is done element by element, i.e., $f(x_1, \ldots, x_d) = (f(x_1), \ldots, f(x_d))$, then the Jacobian matrix is a diagonal matrix:

$$\begin{bmatrix} f'(\boldsymbol{x}_1) & \cdots & \\ \vdots & \ddots & \vdots \\ & \cdots & f'(\boldsymbol{x}_d) \end{bmatrix}$$

Its determinant is also easy to calculate, that is:

$$\left| det\left(\frac{\partial f}{\partial \boldsymbol{x}}\right)\right| = \prod_{i=1}^{d} f'(\boldsymbol{x}_i) \tag{1.17}$$

In order to provide the reader with a deeper understanding of such models, we provide a detailed description of the NICE model [13]. The inverse transform of the NICE model $f(x)$ consists of multiple additive coupling layers and a scale transformation layer, as shown in Fig. 1.9.

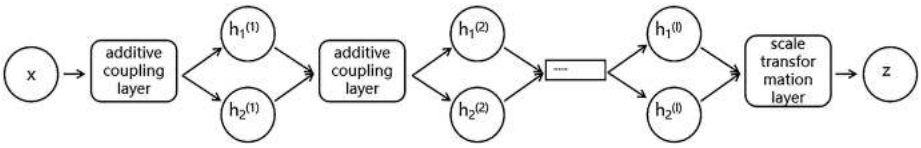


Fig. 1.9 NICE model structure

In each additive coupling layer, the n-dimensional sample x is first decomposed into two parts x_1 and x_2, for example, the first 1, 3, 5... element is assigned to the x_1 part and the first element 2, 4, 6... element into the x_2 part, each of which has dimension n/2. It is also possible to assign the x use other divisions. The two parts are then transformed:

$$h_1 = x_1 \tag{1.18}$$

$$h_2 = x_2 + m(x_1) \tag{1.19}$$

Where $m()$ is an arbitrary function, note that it is important here to ensure the dimensionality of the $m()$ output is consistent with x_2, the NICE model is constructed using a multilayer fully connected network and the ReLU activation function. It is easy to find that the additive coupling layer is used as part of the inverse transform $f(x)$, which is invertible and the determinant of the Jacobi matrix is easy to compute. When known h_1 and h_2, the inverse transform can be obtained.

$$x_1 = h_1 \tag{1.20}$$

$$x_2 = h_2 - m(h_1) \tag{1.21}$$

Its Jacobi matrix is:

$$\begin{bmatrix} I & \\ \frac{\partial m}{\partial h_1} & I \end{bmatrix}$$

According to the nature of the triangular matrix, its determinant is the product of the diagonal elements, so the determinant of the additive coupling layer Jacobi matrix has absolute value 1. Since the inversion of each layer is easy to calculate, the inversion after series coupling is still easy to calculate. The Jacobi matrix is:

(1.22)

$$\frac{\partial f}{\partial x}=\frac{\partial h^{(l)}}{\partial h^{(l-1)}}\frac{\partial h^{(l-1)}}{\partial h^{(l-2)}}\bullet\bullet\bullet\frac{\partial h^{(1)}}{\partial h^{(0)}}$$

According to the properties of matrix determinant, there are:

$$\det\left|\frac{\partial f}{\partial x}\right|=\det\left|\frac{\partial h^{(l)}}{\partial h^{(l-1)}}\right|\bullet\det\left|\frac{\partial h^{(l-1)}}{\partial h^{(l-2)}}\right|\bullet\bullet\bullet\det\left|\frac{\partial h^{(1)}}{\partial h^{(0)}}\right|=1 \tag{1.23}$$

It is important that different division strategies should be used in the different additive coupling layers, allowing sufficient confusion of information in the different dimensions of the sample. In the scale transformation layer, a vector is defined containing *n* non-negative parameters $s = [s_1, s_2, ..., s_n]$ that yields the corresponding hidden variable *z* by multipling the output of the additive coupling layer $h^{(l)}$ with *s* element by element. Here *s* is used to control the feature transformation of each dimension, which can characterize the importance of the dimension, and a larger value of the corresponding dimension indicates a lower importance of this dimension, because the hidden variable needs to go through the scale transformation layer first when generating the sample, and the hidden variable needs to be multiplied element by element in the scale transformation layer $1/s$. To calculate the Jacobi matrix, the scale transformation is written in the form of a diagonal matrix:

$$\begin{bmatrix}s_1 & \cdots & \\ \vdots & \ddots & \vdots \\ & \cdots & s_n\end{bmatrix}$$

Then the determinant of its Jacobi matrix is $s_1 s_2 \cdot \cdot \cdot s_n$. Now, we construct the invertible, easy-to-compute absolute values of the determinant of the Jacobi matrix inverse transformation *f*(*x*). For the hidden variable, the NICE model assumes that its *n* dimensions are independent of each other, i.e.,

$$p_z(z)=\prod_{i=1}^{n}p_z(z_i) \tag{1.24}$$

If random variable *z* obeys Gaussian distribution, the likelihood function of the sample *x* is

$$\log p(x)=\log\left(\prod_{i=1}^{n}\frac{1}{\sqrt{2\pi}}e^{-\frac{f(x)_i^2}{2}}\prod_{i=1}^{n}s_i\right)=-\frac{n}{2}\log 2\pi-\sum_{i=1}^{n}\frac{f(x)_i^2}{2}+\sum_{i=1}^{n}\log\left(s_i\right) \tag{1.25}$$

$$p_z(z)=\prod_{i=1}^{n}\frac{1}{(1+e^{z_i})(1+e^{-z_i})} \tag{1.26}$$

Then the likelihood function of the sample is

$$\log p(x)=\log\left(\prod_{i=1}^{n}\frac{1}{\left(1+e^{f(x)_i}\right)\left(1+e^{-f(x)_i}\right)}\prod_{i=1}^{n}s_i\right)=-\sum_{i=1}^{n}\log\left(1+e^{f(x)_i}\right)-\sum_{i=1}^{n}\log\left(1+\right.$$

Now, we can use the maximum likelihood method to train the NICE model, and after the training is completed, we also get the generated model (*z*). If *z* obeys a Gaussian distribution, then sampling directly from the Gaussian distribution would yield *z*; if we choose *z* is a logistic distribution, we can now sample from a uniform distribution between 0 and 1 to obtain ϵ and then use the transformation $z = t(\epsilon)$ to obtain the hidden variable. According to the mapping relationship of two random variables,

$$p(z)=p(\epsilon)\left|\det\frac{\partial t^{-1}}{\partial z}\right| \tag{1.28}$$

there are $t(\epsilon) = \log \epsilon - \log(1-\epsilon)$. Using the nonlinear transformation $g(z)$ to the hidden variable z, i.e., through the inverse transformation of the scale transform layer and the inverse transformation of multiple additive coupling layers, we obtain the generated samples x.

The core code of the NICE model is shown below:

```
# Additive coupling layer
class Coupling(nn.Module).
def __init__(self, in_out_dim, mid_dim, hidden, mask_config).
super(Coupling, self). __init__()
self.mask_config = mask_config

self.in_block = nn.Sequential(
nn.Linear(in_out_dim//2, mid_dim).
nn.ReLU())
self.mid_block = nn.ModuleList([
nn.Sequential(
nn.Linear(mid_dim, mid_dim).
nn.ReLU()) for _ in range(hidden - 1)])
self.out_block = nn.Linear(mid_dim, in_out_dim//2)

def forward(self, x, reverse=False).
[B, W] = list(x.size())
x = x.reshape((B, W//2, 2))
if self.mask_config.
on, off = x[:,:, 0], x[:,:, 1]
else.
off, on = x[:,:, 0], x[:,:, 1]
off_ = self.in_block(off)
for i in range(len(self.mid_block)).
off_ = self.mid_block[i](off_)
shift = self.out_block(off_)
if reverse.
on = on - shift
else.
on = on + shift
if self.mask_config.
x = torch.stack((on, off), dim=2)
else.
x = torch.stack((off, on), dim=2)
return x.reshape((B, W))

# Scale transformation layer
class Scaling(nn.Module).
def __init__(self, dim).
super(Scaling, self). __init__()
self.scale = nn.Parameter(
torch.zeros((1, dim)), requires_grad=True)

def forward(self, x, reverse=False).
log_det_J = torch.sum(self.scale)
if reverse.
x = x * torch.exp(-self.scale)
else.
x = x * torch.exp(self.scale)
return x, log_det_J

# NICE model
```

```
class NICE(nn.Module).
def __init__(self, prior, coupling.
in_out_dim, mid_dim, hidden, mask_config).
self.prior = prior
self.in_out_dim = in_out_dim

self.coupling = nn.ModuleList([
Coupling(in_out_dim=in_out_dim.
mid_dim=mid_dim.
hidden=hidden.
mask_config=(mask_config+i)%2) \
for i in range(coupling)])
self.scaling = Scaling(in_out_dim)

def g(self, z).
x, _ = self.scaling(z, reverse=True)
for i in reversed(range(len(self.coupling))).
x = self.coupling[i](x, reverse=True)
return x

def f(self, x).
for i in range(len(self.coupling)).
x = self.coupling[i](x)
return self.scaling(x)

def log_prob(self, x).
z, log_det_J = self.f(x)
log_ll = torch.sum(self.prior.log_prob(z), dim=1)
return log_ll + log_det_J

def sample(self, size).
z = self.prior.sample((size, self.in_out_dim)).cuda()
return self.g(z)

def forward(self, x).
return self.log_prob(x)
```

The Real NVP [5] model and Glow [6] model have improved upon the NICE model by introducing innovations such as convolutional operations in the coupling layers and adding multi-scale structures, further enhancing the quality of generated samples. Additionally, autoregressive model with the incorporation of nonlinear mappings allows for the construction of autoregressive flow model, which primarily include two categories: Masked Autoregressive Flow (MAF) and Inverse Autoregressive Flow (IAF). Due to significant differences in their design methodologies, they each possess distinct advantages in terms of the speed of computing the likelihood function. Overall, flow model, through their ingenious design, enables the precise calculation of the probability density function of samples and possess a very elegant theoretical foundation. However, their drawbacks lie in the complexity of the computational process and excessively long training times, resulting in a performance gap compared to models such as GAN in practical applications.

1.2.4 Variation Autoencoder

Autoencoders occupy an important position in deep learning, initially being used solely for dimensionality reduction or feature learning. A typical autoencoder consists of two neural networks: an encoder and a decoder, as shown in Fig. 1.10.

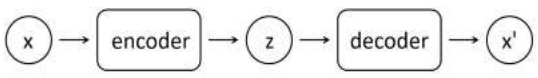


Fig. 1.10 Self-encoder structure

The sample x goes through the encoder to get some sort of encoding representation z. The dimensionality of z is generally less than x. Then the encoding vector z is fed to the decoder can obtain the reconstructed of the sample x. If the reconstruction x' is of good quality, it is considered that the encoder has successfully learned the abstract features of the sample, which can also be interpreted as achieving dimensionality reduction. Once the abstract features of the data z are learned, they can not only be used for sample reconstruction but also for classification tasks by simply attaching a classifier to the encoder, as shown in Fig. 1.11. Latent variables have important and widespread applications in generative models. On the one hand, they can provide a "meaningful" representation of the samples; on the other hand, models based on latent variables have faster sampling speeds compared to Fully Visible Belief Networks (FVBNs). Since FVBNs only contain observed variables and establish dependencies among these observed variables, some of the observed variables can be modified into latent variables, and a conditional probability relationship between them can be established to achieve faster sampling speeds.

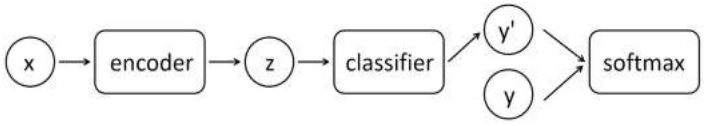


Fig. 1.11 Using self-encoders for classification tasks

VAE [7] (variation autoencoder) constructs the autoencoder as a generative model. It treats z as a hidden variable for generating samples and makes some modifications to the encoder and decoder, ultimately realizing a highly performant generative model. Unlike other generative models such as FVBNs and GAN, VAE aims to define a generative model that generates samples through latent variables:

$$p_\theta(x) = \int_z p_\theta(x|z)p_\theta(z)dz \tag{1.29}$$

The way this generative model generates samples is very concise and elegant: first, sample z from the distribution of latent variables $p_\theta(x)$, and then sample from the conditional distribution $p_\theta(x|\, z)$ to get x. However, this generative model cannot be directly constructed! This is because training generative models usually requires maximizing the log-likelihood function to solve for the model parameters θ, which means that for N independently and identically distributed training samples $\{x^{(1)},x^{(2)},\ldots,x^{(N)}\}$, we need to:

$$\max_\theta \sum_{i=1}^{N} \log \left[p_\theta\left(x^{(i)}\right)\right] \tag{1.30}$$

We need to calculate $p_\theta(x)$. Analysis the computational equation of $p_\theta(x)$, the interior of the integral sign is relatively easy to solve. For the hidden variable z, the prior distribution of $p_\theta(z)$ can be designed as a simple Gaussian distribution. $p_\theta(x|\, z)$ can be learned using a neural network, then the hard part is traversing all the hidden variables z to calculate the integral. In addition, the posterior distribution of the hidden variable z is

$$p_\theta(z|x) = \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(x)} \tag{1.31}$$

It is also difficult to solve. Training a generative model involves first solving for the log-likelihood function (i.e., using the likelihood function as the loss function) and then maximizing it. The idea behind VAE is that, although the exact log-likelihood function cannot be solved, we can obtain a lower bound of the log-likelihood function and maximize this lower bound, which approximates maximizing the log-likelihood function. Specifically, VAE introduces a new probability distribution $q_\phi(z|\, x)$ to approximate the posterior distribution $p_\theta(z|\, x)$. The log-likelihood function in this case is:

$$\log\left[p_\theta\left(x^{(i)}\right)\right]=\mathbb{E}_{z\sim q_\phi(z|x)}\log\left[p_\theta\left(x^{(i)}\right)\right]=\mathbb{E}_z\left[\log\frac{p_\theta\left(x^{(i)}|z\right)p_\theta(z)}{p_\theta\left(z|x^{(i)}\right)}\right]=\mathbb{E}_z\left[\log\frac{p_\theta\left(x^{(i)}|z\right)p_\theta(z)}{p_\theta\left(z|x^{(i)}\right)}\frac{q_\phi\left(z|x^{(}\right.\right.$$

The final equation consists of three terms, where the first two terms are computable, but the third term is not. However, based on the properties of the KL divergence, we know that the third term is necessarily greater than or equal to 0. In other words

$$\log\left[p_\theta\left(x^{(i)}\right)\right]\geq\mathbb{E}_z\left[\log p_\theta\left(x^{(i)}|z\right)\right]-\mathrm{D}_{\mathrm{KL}}\left(q_\phi\left(z|x^{(i)}\right)\parallel p_\theta(z)\right)\tag{1.33}$$

We refer to the right-hand side of the above inequality as variational lower bound (ELBO), and denote as $l(x^{(i)};\theta,\phi)$. Then, the variational lower bound is simply need to be maximized, i.e., regard the variational lower bound as the loss function of the model:

$$\max_{\theta,\phi}\ l\left(x^{(i)};\theta,\phi\right)\tag{1.34}$$

At this point, the core idea of VAE has been realized. Next, we will describe some details, such as how to transform the mathematical model into a neural network, i.e., how to compute the variational lower bound ELBO? Let's first look at the second term of EBLO, where means computing KL divergence between $q_\phi(z|x^{(i)})$ (the approximate distribution of the posterior distribution to the hidden variable z) with $p_\theta(z)$ (and the prior distribution of the hidden variable). Based on the experience in practice, two basic assumptions are made: 1. The prior distribution of the hidden variable $p_\theta(z)$ is a D-dimensional standard Gaussian distribution $N(0,I)$. Noticing that $p_\theta(z)$ not contain any unknown parameters, so it is rewritten as $p(z)$; 2. the approximate distribution of the posterior distribution of the hidden variables $q_\phi(z|x^{(i)})$ is a Gaussian distribution $N(\mu,\Sigma;x^{(i)})$ in which the components are independent of each other. Each sample $x^{(i)}$ matches a D-dimensional Gaussian distribution $N(\mu,\Sigma;x^{(i)})$. Now it is necessary to just know about $\mu(x^{(i)})$ and $\Sigma(x^{(i)})$, we could calculate the KL divergence.

We use two neural networks (i.e., encoders with parameters) to solve for the mean and logarithm of the variance (because the value domain of the logarithm of the variance is all real numbers, and the value domain of the variance is all positive real numbers, it is relatively convenient to use neural networks to fit the logarithm of the variance without precisely designing the activation function). Since the D-dimensional hidden variables z are independent of each other, the mean is a D-dimensional vector, and the variance is a D-dimensional diagonal matrix, i.e.,

$$\begin{bmatrix}\sigma_1^2 & & & \\ & \sigma_2^2 & & \\ & & \cdots & \\ & & & \sigma_D^2\end{bmatrix}$$

The variance is actually contain D parameters required to learn, instead of D^2. Then the input of the so-called encoder here is the sample $x^{(i)}$, the first encoder output is a D-dimensional vector[$\mu_1,\mu_2,...,\mu_D$], and the output of the second encoder is also a D-dimensional vector, $\left[\log\sigma_1^2,\log\sigma_2^2,\ldots,\log\sigma_D^2,\right]$, that is

$$\mu\left(x^{(i)}\right)=enc_{1,\phi}\left(x^{(i)}\right)\tag{1.35}$$

$$log\ \sigma^2\left(x^{(i)}\right)=enc_{2,\phi}\left(x^{(i)}\right)\tag{1.36}$$

Since each dimension of the two Gaussian distributions is independent of each other, the KL divergence can be calculated separately, where the KL divergence value of the dth dimension is

(1.37)

$$\mathrm{D}_{\mathrm{KL}}\left(q_\phi\left(z\middle|x^{(i)}\right)_d \parallel p_\theta(z)_d\right) = \mathrm{D}_{\mathrm{KL}}\left(N\left(\mu_d, \sigma_d^2\right) \parallel N(0,1)\right) = \frac{1}{2}\left(-\log \sigma_d^2 + \mu_d^2 + \sigma_d^2 - 1\right)$$

The above calculation process is relatively simple and will not be expanded here. The total KL divergence is easily computed as

$$\mathrm{D}_{\mathrm{KL}}\left(q_\phi\left(z\middle|x^{(i)}\right) \parallel p_\theta(z)\right) = \sum_{d=1}^{D} \mathrm{D}_{\mathrm{KL}}\left(q_\phi\left(z\middle|x^{(i)}\right)_d \parallel p_\theta(z)_d\right) \tag{1.38}$$

Computationally, by having the encoder learn the mean and variance of the approximate distribution of the latent variable's posterior distribution, we obtain the probability density function of the approximate posterior distribution of the latent variable, allowing us to calculate the KL divergence. Essentially, during VAE training, the encoder is expected to minimize the KL divergence, which means making the approximate posterior distribution converge toward the standard Gaussian distribution. That is, for each sample $x^{(i)}$, the $q_\phi(z|x^{(i)})$ should converge to a Gaussian distribution.

Now focusing on the first term of ELBO $\mathbb{E}_z\left[\log p_\theta\left(x^{(i)}\middle|z\right)\right]$, in order to calculate this term, an empirical approximation needs to be used

$$\mathbb{E}_z\left[\log p_\theta\left(x^{(i)}\middle|z\right)\right] \approx \log p_\theta\left(x^{(i)}\middle|z\right) \tag{1.39}$$

That means it is not necessary to sample all the different z and then calculate $\log p_\theta(x^{(i)}|z)$ when calculating this term. Instead, we only need to sample from it once. This may seem unreasonable, but the actual effect proves that the approximately equal relationship holds. In addition to the fact that in general autoencoder are mapped one-to-one, i.e., one sample x corresponds to a hidden variable z, so it can be imagined that $q_\phi(z|x^{(i)})$ is a very sharp single-peaked distribution, then the difference of the mean value calculated by multiple sampling or one sampling is not much. Next, in order to calculate $\log p_\theta(x^{(i)}|z)$, we again make the assumption that $p_\theta(x|z)$ is a Bernoulli or Gaussian distribution. When the Bernoulli distribution is assumed, the corresponding x is a binary vectors with Q dimensions independent of each other. The Q parameters of the Bernoulli distribution $[\rho_1, \rho_2, ..., \rho_Q,]$ are handed over to the neural network to learn, and this neural network decoder, parameterized by θ, input the hidden variable z and the output is $[\rho_1, \rho_2, ..., \rho_Q,]$, that is

$$\rho(z) = dec_\theta(z) \tag{1.40}$$

The likelihood of the sample can now be calculated as

$$p_\theta\left(x^{(i)}\middle|z\right) = \prod_{q=1}^{Q} \left(\rho_q(z)\right)^{x_q^{(i)}} \left(1 - \rho_q(z)\right)^{1 - x_q^{(i)}} \tag{1.41}$$

The corresponding log-likelihood function is:

$$\log p_\theta\left(x^{(i)}\middle|z\right) = \sum_{q=1}^{Q} x_q^{(i)} \log\left(\rho_q(z)\right) + \left(1 - x_q^{(i)}\right) \log\left(1 - \rho_q(z)\right) \tag{1.42}$$

So it is only necessary to design the last layer of the activation function of the encoder as a sigmoid function and use the binary cross-entropy as the loss function of the decoder. If we assume that $p_\theta(x^{(i)}|z)$ is a Gaussian distribution, corresponding sample x is a real-valued vector with Q dimensions independent of each other, and the variance of each dimension of this Gaussian distribution is fixed as some constant σ^2. Q parameters $[\mu_1, \mu_2, ..., \mu_Q]$ is given to the neural network, the decoder, which is parameterized by θ, input hidden variable z and the output $[\mu_1, \mu_2, ..., \mu_Q]$, that is

$$\mu(z) = dec_\theta(z) \tag{1.43}$$

The likelihood function of the sample can now be calculated as

$$p_\theta\left(x^{(i)}\middle|z\right) = \frac{1}{\prod_{q=1}^{Q}\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2} \parallel x^{(i)} - \mu(z) \parallel_2^2\right) \tag{1.44}$$

The corresponding log-likelihood is:

$$\log p_\theta\left(x^{(i)}\middle|z\right) \sim -\frac{1}{2\sigma^2} \parallel x^{(i)} - \mu(z) \parallel_2^2 \tag{1.45}$$

Therefore, the activation function of the last layer of the encoder needs to be designed to have a range of all real values, and the Mean Squared Error (MSE) is used as the loss function. Computationally, an approximation operation based on empirical knowledge is used for one time sample and we rely on the encoder learning parameters of $p_\theta(x|z)$, and finally the likelihood of the sample with conditional probability is calculated. VAE aims to maximize the loss function corresponding to the decoder part, essentially wanting to minimize the reconstruction error of the sample, which is very obvious in Bernoulli distribution, while in Gaussian distribution, MSE loss wants to bring the output of the encoder (mean of Gaussian distribution) close to the sample.

To review the process above, the training process is as follows: sending sample $x^{(i)}$ into the encoder can obtain the parameters of the approximation posterior distribution of the hidden variable (i.e., the mean and variance of the Gaussian distribution), at which point a hidden variable needs to be sampled from the distribution z and then send to decoder. In fact, a small problem here is that the process of sampling from the distribution is not derivable, i.e., the mean and variance parameters calculated by the encoder are "flooded" after the hidden variable is sampled, and the decoder is only facing an isolated z that can not know from which Gaussian distribution. We need to combine the $\mu(x^{(i)})$ and $\sigma(x^{(i)})$ with the encoder, otherwise the gradient propagation will break of sample z in backpropagation. Reparameterization trick does a simple treatment by sampling directly in the standard Gaussian distribution $N(0, I)$ to obtain ϵ, and then let $z = \mu + \epsilon \times \sigma$, so that the backpropagation link is connected, as shown in Fig. 1.12.

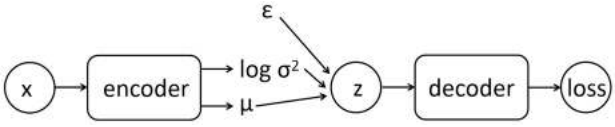


Fig. 1.12 Forward calculation of variation autoencoder

After the training is completed, we can directly sample the latent variable z from $p(z)$ and feed it into the decoder. In the case of the Bernoulli distribution, the decoder outputs the probability of each dimension of the sample taking a certain value. In the case of the Gaussian distribution, the decoder outputs the mean, which is the generated sample.

VAE is often compared with GAN, which is an implicit probability generative model where the likelihood function does not appear explicitly in GAN, while VAE is an explicit probability generative model that also tries to maximize the likelihood function, but unlike the FVBN model where an exact likelihood function exists for maximization, VAE obtains a lower bound on the likelihood function and approximates the maximum likelihood. In the image generation, a relatively obvious drawback of VAE is that the generated images tend to be blurry. This may be a common issue with models that use maximum likelihood since the essence of maximum likelihood is minimization $D_{KL}(p_{\text{data}} \parallel p_{\text{model}})$. The explanation of this problem involves the nature of KL divergence and would not be expanded here.

The core code of the VAE model is as follows:

```
# VAE model
class VAE(nn.Module).
def __init__(self, encoder_layer_sizes, latent_size, decoder_layer_sizes.
conditional=False, num_labels=0).
super(). __init__()
if conditional.
```

```
assert num_labels > 0
assert type(encoder_layer_sizes) == list
assert type(latent_size) == int
assert type(decoder_layer_sizes) == list
self.latent_size = latent_size
self.encoder = Encoder(
encoder_layer_sizes, latent_size, conditional, num_labels)
self.decoder = Decoder(
decoder_layer_sizes, latent_size, conditional, num_labels)

def forward(self, x, c=None).
if x.dim() > 2.
x = x.view(-1, 28*28)
means, log_var = self.encoder(x, c)
z = self.reparameterize(means, log_var)
recon_x = self.decoder(z, c)
return recon_x, means, log_var, z

def reparameterize(self, mu, log_var).
std = torch.exp(0.5 * log_var)
eps = torch.randn_like(std)

return mu + eps * std

def inference(self, z, c=None).
recon_x = self.decoder(z, c)

return recon_x

# Encoder
class Encoder(nn.Module).
def __init__(self, layer_sizes, latent_size, conditional, num_labels).
super(). __init__()
self.conditional = conditional
if self.conditional.
layer_sizes[0] += num_labels
self.MLP = nn.Sequential()

for i, (in_size, out_size) in enumerate(zip(layer_sizes[:-1],
layer_sizes[1:])).
self.MLP.add_module(
name="L{:d}".format(i), module=nn.Linear(in_size, out_size))
self.MLP.add_module(name="A{:d}".format(i), module=nn.ReLU())

self.linear_means = nn.Linear(layer_sizes[-1], latent_size)
self.linear_log_var = nn.Linear(layer_sizes[-1], latent_size)

def forward(self, x, c=None).
if self.conditional.
c = idx2onehot(c, n=10)
x = torch.cat((x, c), dim=-1)
x = self.MLP(x)
means = self.linear_means(x)
log_vars = self.linear_log_var(x)

return means, log_vars
```

```
# Decoder
class Decoder(nn.Module).
def __init__(self, layer_sizes, latent_size, conditional, num_labels).
super(). __init__()

self.MLP = nn.Sequential()
self.conditional = conditional
if self.conditional.
input_size = latent_size + num_labels
else.
input_size = latent_size

for i, (in_size, out_size) in enumerate(zip([input_size]+layer_sizes[:-1],
layer_sizes)).
self.MLP.add_module(
name="L{:d}".format(i), module=nn.Linear(in_size, out_size))
if i+1 < len(layer_sizes).
self.MLP.add_module(name="A{:d}".format(i), module=nn.ReLU())
else.
self.MLP.add_module(name="sigmoid", module=nn.Sigmoid())

def forward(self, z, c).
if self.conditional.
c = idx2onehot(c, n=10)
z = torch.cat((z, c), dim=-1)
x = self.MLP(z)

return x
```

1.2.5 Boltzmann Machine

Boltzmann machine belongs to another explicit probability model, which is an energy-based model. Training Boltzmann machines also requires the same idea based on the maximum likelihood, but when calculating the gradient of the maximum likelihood, a different approximation algorithm from variation methods is used. Boltzmann machines have received less attention lately, so we will only briefly describe them here.

In energy models, the probability of a sample $p(x)$ is usually modeled in the following form:

$$p(x) = \frac{e^{-E(x)}}{Z} \tag{1.46}$$

where $Z = \sum_x e^{-E(x)}$ is the collocation function. To enhance the expressiveness of the model, it is common to add visible variables v except the hidden variable h. The most simple restricted Boltzmann machine, RBM, for example, has both visible and hidden variables as binary discrete random variables (which of course could be extended to real values). It defines an bipartite undirected probability graph, where the visible variables v form one part and the hidden variables h form another part. There is no connection between the visible variables and no connection between the hidden variables (hence the term "restricted"), but full connections exist between visible variables and hidden variables, as shown in Fig. 1.13.

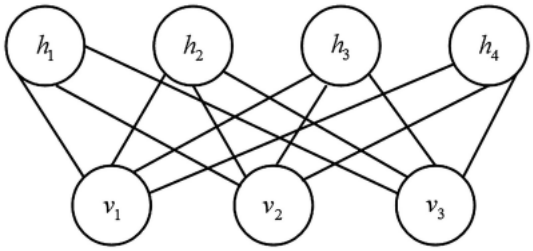


Fig. 1.13 Restricted Boltzmann machine structure diagram

In RBM, the joint probability distribution of the visible and hidden variables is given by the energy function, that is

$$p(v,h)=\frac{\exp\left(-E(v,h)\right)}{Z} \tag{1.47}$$

where the expression of the energy function is

$$E(v,h)=-h^{T}Wv-b^{T}v-c^{T}h \tag{1.48}$$

Distribution function *Z* can be written as

$$Z=\sum_{v}\sum_{h}\exp\left\{-E(v,h)\right\} \tag{1.49}$$

Considering the special structure of the dichotomous graph, it is found that the visible variables are independent of each other when the hidden variables are known, and the hidden variables are also independent of each other when the visible variables are known, i.e., there are

$$p(h|v)=\prod_{i}p(h_{i}|v) \tag{1.50}$$

$$p(v|h)=\prod_{i}p(v_{i}|h) \tag{1.51}$$

Furthermore, a specific expression for the discrete probability can be obtained as

$$p(h_{i}=1|v)=\mathrm{sigmoid}(c_{i}+W_{i}v) \tag{1.52}$$

$$p(v_{j}=1|h)=\mathrm{sigmoid}\left(b_{j}+W_{j}^{T}h\right) \tag{1.53}$$

In order to make the RBM and the energy model have consistent expressions, define the free energy of the visible variable *f*(*v*) as

$$f(v)=-b^{T}v-\sum_{i}log\left(1+e^{(c_{i}+W_{i}v)}\right) \tag{1.54}$$

where h_i is the first *i* hidden variable, the probability of the visible variable at this point is

$$p(v)=\frac{e^{-f(v)}}{Z} \tag{1.55}$$

Distribution function *Z* is $Z=\sum_{v}e^{-f(v)}$. When training the RBM model using the maximum likelihood method, the gradient of the likelihood function needs to be calculated, and the parameters of the model are θ, then

$$\frac{\partial\ log\ p(v)}{\partial\theta}=-\frac{\partial f(v)}{\partial\theta}+\mathbb{E}_{v}\left[\frac{\partial f(v)}{\partial\theta}\right] \tag{1.56}$$

It can be seen that the RBM explicitly defines the probability density function of the visible variables, but it is not easy to solve because the calculation of the collocation function *Z* requires integrating all visible variables *v* and hidden variables *h*, so the log-likelihood log*p*(*v*) cannot be solved directly, so the model cannot be trained directly using the idea of maximum likelihood. However, training the model can still be accomplished by bypassing the solution of the log-likelihood function and directly solving for the gradient of the log-likelihood function. For the weight and bias parameters θ, there has

(1.57)

$$\frac{\partial \log p(v)}{\partial W_{ij}} = -\mathbb{E}_v[p(h_i|v) \bullet v_j] + v_j^{(i)} \bullet \text{sigmoid}\Big(W_i \bullet v^{(i)} + c_i\Big)$$

$$\frac{\partial \log p(v)}{\partial c_i} = -\mathbb{E}_v[p(h_i|v)] + \text{sigmoid}\Big(W_i \bullet v^{(i)}\Big) \tag{1.58}$$

$$\frac{\partial \log p(v)}{\partial b_j} = -\mathbb{E}_v[p(h_i|v)] + v_j^{(i)} \tag{1.59}$$

Analyzing the gradient expression, the hard part of the computation lies in the calculation of the expectation of the visible variable *v*. RBM uses sampling methods to approximate the gradient and then updates the weights using the approximated gradient. In order to sample the visible variables *v*, the RBM constructs a Markov chain that eventually converges to *p*(*v*), i.e., the smooth distribution of the Markov chain is *p*(*v*). The samples are initially randomly given, and the smooth distribution is reached after a sufficient number of iterative runs, at which point the samples are obtained by continuous sampling from the transfer matrix *p*(*v*). We can use Gibbs sampling method to complete the process, due to the independence of the two parts of the variables, when fixing the visible variables (or hidden variables), the distribution of the hidden variables (visible variables) are

$$h^{(n+1)} \sim \text{sigmoid}\Big(W^T v^{(n)} + c\Big) \tag{1.60}$$

$$v^{(n+1)} \sim \text{sigmoid}\Big(W h^{(n+1)} + b\Big) \tag{1.61}$$

In other words, we first sample the hidden variables and then sample the visible variables. By doing so, we can use the random maximum likelihood method to complete the training of the generative model.

Boltzmann machines rely on Markov chain for training models or generating samples, but this technique is rarely used nowadays, most likely because Markov chain approximation techniques cannot be applied to high-dimensional generation problems like ImageNet. Moreover, even though Markov chain methods can be used well for training, generating samples using a model based on Markov chain requires significant computational cost.

The core code for Restricted Boltzmann Machines [8] is shown below:

```
# Constrained Boltzmann machine model
class RBM(nn.Module).
def __init__(self, n_vis=784, n_hin=500, k=5).
super(RBM, self). __init__()
self.W = nn.Parameter(torch.randn(n_hin, n_vis) * 1e-2)
self.v_bias = nn.Parameter(torch.zeros(n_vis))
self.h_bias = nn.Parameter(torch.zeros(n_hin))
self.k = k

def sample_from_p(self, p).
return F.relu(torch.sign(p - Variable(torch.rand(p.size()))))

def v_to_h(self, v).
p_h = F.sigmoid(F.linear(v, self.W, self.h_bias))
sample_h = self.sample_from_p(p_h)
return p_h, sample_h

def h_to_v(self, h).
p_v = F.sigmoid(F.linear(h, self.W.t(), self.v_bias))
sample_v = self.sample_from_p(p_v)
```

```
return p_v, sample_v

def forward(self, v).
pre_h1, h1 = self.v_to_h(v)

h_ = h1
for _ in range(self.k).
pre_v_, v_ = self.h_to_v(h_)
pre_h_, h_ = self.v_to_h(v_)

return v, v_

def free_energy(self, v).
vbias_term = v.mv(self.v_bias)
wx_b = F.linear(v, self.W, self.h_bias)
hidden_term = wx_b.exp().add(1).log().sum(1)
return (-hidden_term - vbias_term).mean()

rbm = RBM(k=1)

train_op = optim.SGD(rbm.parameters(),0.1)

# Training
for epoch in range(10).
loss_ = []
for _, (data, target) in enumerate(train_loader).
data = Variable(data.view(-1, 784))
sample_data = data.bernoulli()
v, v1 = rbm(sample_data)
loss = rbm.free_energy(v) - rbm.free_energy(v1)
train_op.zero_grad()
loss.backward()
train_op.step()
```

1.3 Implicit Generative Model

Implicit generative models are also typically constructed with the aid of latent variables z. This involves first sampling noise z from a fixed probability distribution $p_z(z)$ (such as a Gaussian distribution, uniform distribution), and then using a neural network to map it to a sample x. This idea is consistent with flow model and VAE, but the difference lies in the training methods. For instance, flow model use maximum likelihood methods and require some form of processing of the probability density function $p(x)$ or the likelihood function $\log p(x)$. The core objective of implicit generative model is also to make $p_g(x)$ approximate $p_{\text{data}}(x)$. They do not explicitly model or approximate the probability density function or likelihood function, but they can still indirectly interact with $p_{\text{data}}(x)$ through training data. Typical representatives of implicit generative models include GSN [9] (Generative Stochastic Networks) and GAN [10] (Generative Adversarial Networks). In implicit generative models, we cannot obtain an (approximate) expression for $p(x)$, as the expression is hidden within the neural network, and the model can only generate samples.

Practice has shown that even if a model has a very high likelihood function value, it may still produce low-quality samples. An extreme example is a model that merely memorizes the samples in the training set, demonstrating that explicit generative models are not always reliable. An alternative approach to obtaining generative models without relying on likelihood functions is to use two-sample tests. For instance, the generative model can generate a large number of samples S_1, and the training dataset contains a large number of samples S_2. We can check whether $S_1$and S_2 come from the same probability distribution and continuously optimize the generative model so that the two sets of samples pass the test. Similar to the above idea, implicit generative model actually first calculate and compare some kind of difference between

$p_g(x)$ with $p_{\text{data}}(x)$, and then minimize this difference as much as possible. When choosing this comparison method, it can be divided into two categories based on density ratio $p(x)/q(x)$ and density difference $p(x) - q(x)$.

The first category can be further divided into three methods: probability classification estimation (standard GAN), divergence minimization (f-GAN), and proportion matching (b-GAN). The second category is mainly represented by Integral Probability Metrics (IPM) methods, including WGAN, MMDGAN, and other models. Different comparison methods correspond to different loss functions, such as Maximum Mean Discrepancy (MMD), Jensen-Shannon Divergence (JS), ànd Optimal Transport Distance. It should be noted that when choosing KL Divergence, the optimization objective becomes the previous maximum likelihood estimation goal $\mathbb{E}_x[\log p_\theta(x)]$. For the calculation of this optimization objective, explicit generative models perform decomposition, approximation, and other computational processing, while implicit generative models directly fit the numerical value of the distance between the two sets of samples using neural networks.

Taking GAN as an example, since there is no explicit probability density function $p_g(x)$, GAN cannot directly write out the likelihood function and use the maximum likelihood method. Instead, it directly uses a generator to sample from noise and output samples, and then leverages a discriminator to learn the distance between $p_{\text{data}}(x)$ and $p_g(x)$. The generator is then trained to minimize this distance. It can be seen that GAN indirectly controls $p_g(x)$ by controlling the generator. Compared to fully visible belief networks, GAN can produce samples in parallel, making them more efficient. As a comparison, Boltzmann machines require the use of Markov chains for approximation, which places certain requirements on the probability distribution. Similarly, flow model also require the transformation function $x = g(z)$ to be invertible, with the noise and sample dimensions being the same. Although the determinant of the Jacobian matrix is easy to solve, GAN use a generator as the generative function without the constraints of invertibility, dimensionality, or probability distribution. Additionally, and GAN do not require the participation of Markov chains and do not require much effort to generate a single sample, making them more efficient compared to Boltzmann machines and Generative Stochastic Networks. Compared to VAE, GAN also do not need to deal with the variational lower bound of the likelihood function. Overall, GAN are a very streamlined and efficient model. However, their drawback lies in their difficulty in training. Theoretically, achieving a global Nash equilibrium is almost impossible. Furthermore, they are prone to mode collapse, resulting in poor diversity in the generated samples.

References

1. Zhou, C. H. Machine learning [J]. 2016.
2. Goodfellow I, Bengio Y, Courville A. Deep learning [M]. MIT press, 2016.
3. Hinton GE, Sejnowski TJ, Ackley DH. Boltzmann machines: Constraint satisfaction networks that learn [M]. Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science, 1984.
4. Hinton GE, Osindero S, Teh YW. A fast learning algorithm for deep belief nets [J]. Neural computation, 2006, 18(7): 1527-1554. [MathSciNet][Crossref]
5. Bengio S, Bengio Y. Taking on the curse of dimensionality in joint distributions using neural networks [J]. IEEE Transactions on Neural Networks, 2000, 11(3): 550-557. [Crossref]
6. Bengio S. Modeling High-Dimensional Discrete Data with [C]// Advances in Neural Information Processing Systems 12: Proceedings of the 1999 Conference. MIT Press, 2000, vol. 1, p. 400.
7. Salakhutdinov R, Hinton G. Deep boltzmann machines[C]// Artificial intelligence and statistics. PMLR, 2009: 448-455.
8. Wang, Jing-Long, Professor of Statistics. Multivariate statistical analysis [M]. Science Press, 2008.
9. Frey BJ, Brendan JF, Frey BJ. Graphical models for machine learning and digital communication [M]. MIT press, 1998. [Crossref]
10. Larochelle H, Murray I. The neural autoregressive distribution estimator [C]//Proceedings of the fourteenth international conference on artificial intelligence and statistics. jMLR Workshop and Conference Proceedings, 2011: 29-37.
11. Oord A, Kalchbrenner N, Vinyals O, et al. Conditional image generation with pixelcnn decoders [J]. arXiv preprint arXiv:1606.05328, 2016.
12. Salimans T, Karpathy A, Chen X, et al. Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications[J]. arXiv preprint arXiv:1701.05517, 2017.

13. Dinh L, Krueger D, Bengio Y. Nice: Non-linear independent components estimation [J]. arXiv preprint arXiv:1410.8516, 2014.

14. Dinh L, Sohl-Dickstein J, Bengio S. Density estimation using real nvp [J]. arXiv preprint arXiv:1605.08803, 2016.

15. Kingma DP, Dhariwal P. Glow: Generative flow with invertible 1x1 convolutions [J]. arXiv preprint arXiv:1807.03039, 2018.

16. Hinton G E, Zemel R S. Autoencoders, minimum description length, and Helmholtz free energy [J]. Advances in neural information processing systems, 1994, 6: 3-10.

17. Kingma DP, Welling M. Auto-encoding variational bayes [J]. arXiv preprint arXiv:1312.6114, 2013.

18. Smolensky P. Information processing in dynamical systems: Foundations of harmony theory [R]. Colorado Univ at Boulder Dept of Computer Science, 1986.

19. Bengio Y, Laufer E, Alain G, et al. Deep generative stochastic networks trainable by backprop[C]//International Conference on Machine Learning. PMLR, 2014: 226-234.

20. Goodfellow I, Pouget-Abadie J, Mirza M, et al. Generative adversarial nets [J]. Advances in neural information processing systems, 2014, 27.

P. Long, X. Guo, *Generative Adversarial Network*
https://doi.org/10.1007/978-981-96-9404-4_2

2. Objective Function of GAN

Peng Long[1] ✉ and Xiaozhou Guo[2]
(1) Beijing YouSan Educational Technology, Beijing, China
(2) • China Electronics Technology Group Corporation No. 54 Research Institute, Shijiazhuang, China

Abstract
This chapter mainly introduces the objective functions of GAN. Firstly, a detailed introduction to the standard GAN is provided, including its basic idea, mathematical principles, and algorithm flow, etc. It explains two different objective functions based on f-divergence, namely LSGAN and EBGAN based on the energy model, and also derives and summarizes fGAN based on arbitrary f-divergence. Among another major category of objective functions based on IPM, a very detailed introduction to the Wasserstein distance and the derivation of the objective function of WassersteinGAN is given. Then, a Loss-Sensitive GAN that achieves the same goal as WGAN in a different way is derived. Subsequently, a method WGAN-GP for handling the Lipschitz constraint through a regularization term is introduced. Finally, a detailed explanation of McGAN, MMDGAN, etc. based on the IPM mode will be given. In addition, we explain other types of objective functions, including the reconstruction loss function and the relative loss function.

Keywords f-divergence – IPM – WGAN – Objective function

Chapter 2 mainly introduces the objective function of GAN. Section 2.1 starts with a detailed introduction of the standard GAN, including its basic ideas, fundamentals, and algorithmic flow. Sections 2.2 and 2.3 introduce Least Squares GAN (LSGAN) and Energy-Based GAN (EBGAN) as solution to the vanishing gradients problem. Section 2.4 summarizes the fGAN based on arbitrary f-divergencet. We offer a very detailed explanation of the Wasserstein distance and Wasserstein GAN (WGAN) in Sect. 2.5 and introduce two methods for addressing the Lipschitz constraint in Sects. 2.6–2.8: weight clipping and WGAN with Gradient Penalty (WGAN-GP). In fact, WGAN belongs to the family of Integral Probability Metrics (IPM), and we will introduce IPM and briefly discuss related GAN such as McGAN, MMDGAN, and others in Sect. 2.9. The last section explains other types of objective functions, including reconstruction loss functions, relative loss functions, etc. Through the study of Chap. 2, we should have a deeper understanding of the principle of GAN and the function and nature of the objective function.

Section 2.1	GAN
Section 2.2	LSGAN
Section 2.3	EBGAN
Section 2.4	fGAN
Section 2.5	WGAN
Section 2.6	LS-GAN
Section 2.7	WGAN-GP
Section 2.8	IPM
Section 2.9	RGAN

2.1 GAN

GAN is a deep generative model, first proposed by Goodfellow in 2014 [1], which has developed into one of the hottest models nowadays. Inspired by game theory, it generally consists of two neural networks: the generator (G) and the discriminator (D). As shown in Fig. 2.1, these two neural networks are trained in an adversarial manner, ultimately resulting in a generator with excellent performance.

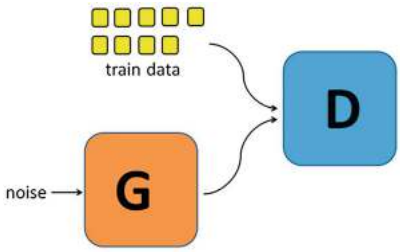


Fig. 2.1 Basic structure of GAN

2.1.1 General Understanding of GAN

Let us first describe the basic principle of GAN in layman's terms from the perspective of game theory. The function of the generator is to generate some random samples, while the function of the discriminator is to determine the authenticity of the given input samples, i.e., to determine whether the input samples of the discriminator come from the training dataset. For example, if we have a training dataset of apple images and the generator generates a banana image at this point, it is unlikely that this banana image comes from the apple dataset. Therefore, a well-trained discriminator should be able to identify that this banana image is fake. During the training process, the generator and the discriminator engage in a game with each other. The generator continuously improves its generative capabilities, generating samples that increasingly resemble those in the training dataset in order to deceive the discriminator. Meanwhile, the discriminator continuously enhances its discriminative abilities to distinguish as accurately as possible whether the input samples come from the training dataset.

For instance, when the generator initially generates banana images, due to the significant shape differences between bananas and apples, the discriminator, after learning the shape difference information, can determine that the banana images are fakes created by the generator, rather than coming from the training dataset. Subsequently, the generator self-improves based on the feedback from the discriminator and must generate images with the same shape as apples. The improved generator will then produce apple-shaped images. In response, the discriminator begins to self-improve by seeking new classification features from both the apple images in the training dataset and the orange images generated by the generator. The discriminator may choose to add color as a new feature for classification. At this point, the discriminator can once again distinguish between oranges and apples, prompting the generator to continue improving. This process alternates until the generator can produce apple images that are identical to those in the training dataset.

Initially, both the generator and the discriminator are relatively weak. However, through continuous self-learning and competition, they will ultimately reach a Nash equilibrium state. In this state, the samples generated by the generator are identical to those in the training dataset, while the discriminator possesses the strongest discriminative ability, capable of detecting any "subtle" differences between the input samples and those in the training dataset. The discriminator is able to detect any "subtle" differences between the input samples and the training dataset.

In the Nash equilibrium state, neither the generator nor the discriminator can make any further improvements. Any change in the generator would indicate a decline in its generative capabilities, and any change in the discriminator would signify a decrease in its discriminative abilities. Regardless of how the generator (or the discriminator) changes, the discriminator (or the generator) will not make corresponding adjustments. The game training process comes to an end. Obviously, since the samples generated by the generator are exactly the same as those in the training dataset, the discriminator will not be able to determine the authenticity of the samples generated by the generator, and we have a perfect generator at this point.

2.1.2 GAN Model

The above description can first generate a basic impression of GAN, but it is not rigorous, we need to describe the working principle of GAN in mathematics completely.

Assume that both the generator and discriminator are fully connected networks with parameters denoted as θ and ϕ, assume that the training dataset $\{x^{(1)}, x^{(2)}, \ldots, x^{(N)}\}$ is sampled from the probability

distribution $p_{data}(x)$, and the sample dataset $\left\{x_G^{(1)}, x_G^{(2)}, \ldots, x_G^{(N)}\right\}$ is generated by the generator which satisfies a probability distribution $p_g(x)$.

The input to the discriminator is a sample x, and the output is a probability value between 0 and 1, indicating the probability of the sample x originated from the training dataset distribution p_{data}, and $1 - p$ denotes the probability that the sample x from the distribution p_g. So, $D(x) = 1$ indicates that the sample$x$ is derived entirely from the training dataset, while $D(x) = 0$ indicates that the sample x does not originate from the training dataset at all, i.e., it originates entirely from the generating sample distribution. Note that the output of the discriminator is a "soft" result, not a "hard" result as described before. The activation function of the last layer of the discriminator mostly uses the Sigmoid function.

When training the discriminator, we face a supervised learning binary classification problem, where the discriminator should output 1 for the samples in the training dataset and 0 for the samples generated by the generator, as shown in Fig. 2.2. The objective function of the discriminator can be obtained using the binary cross-entropy as the loss function:

$$\max_{\theta} \ \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log (1 - D(G(z)))] \tag{2.1}$$

The training data should be:

$$\left\{\left(x^{(1)}, 1\right), \left(x^{(2)}, 1\right), \ldots, \left(x^{(N)}, 1\right), \left(G\left(z^{(1)}\right), 0\right), \left(G\left(z^{(2)}\right)\right), 0\right), \ldots \left(G\left(\mathrm{z}^{(N)}\right), 0\right)\right\}$$

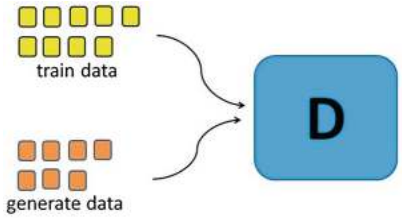


Fig. 2.2 Principle of discriminator

The objective function when training with two types of samples is

$$\max_{\theta} \frac{1}{N} \sum_{i=1}^{N} \left[\log D\left(x^{(i)}\right) + \log \left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right] \tag{2.2}$$

When training the generator, the training data are: $\{z^{(1)}, z^{(2)}, \ldots, \mathrm{z}^{(N)}\}$, as shown in Fig. 2.3

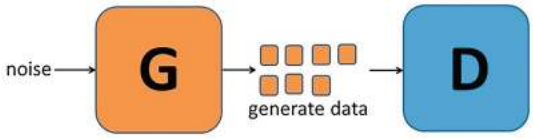


Fig. 2.3 Generator principle

For the generator, the objective function is:

$$\min_{\phi} \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log (1 - D(G(z)))] \tag{2.3}$$

The first term is constant with respect to the generator, so it can be simplified as

$$\min_{\phi} \mathbb{E}_{z\sim p_z} \log (1 - D(G(z))) \tag{2.4}$$

The actual objective function when using sample training is

(2.5)

$$\min_{\phi} \frac{1}{N} \sum_{i=1}^{N} \log \left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$$

The widely used method for alternately training a GAN using a discriminator and a generator is to first train the discriminator for *k* iterations and then train the generator for 1 iteration, repeating this process until the objective functions converge. The entire algorithm is shown as follows:

GAN training algorithm
1. while not converge
2. for *k* iterations
3. sample $\{z^{(1)}, z^{(2)}, \ldots z^{(N)}\}$ from $p_Z(z)$
4. sample $\{x^{(1)}, x^{(2)}, \ldots, x^{(N)}\}$ from $p_{data}(x)$
5. train discriminator according to $\max_{\theta} \frac{1}{N} \sum_{i=1}^{N} \left[\log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right]$
6. end for
7. sample $\{z^{(1)}, z^{(2)}, \ldots z^{(N)}\}$ from $p_Z(z)$
8. train generator from $\min_{\phi} \frac{1}{N} \sum_{i=1}^{N} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$
9. end while

In practice, it is found that in the early stage of training the generator, the generating ability of the generator is generally poor, while the discriminator's discriminating ability tends to be stronger, so the $D(G(z))$ values are generally small, which in turn leads to a relatively small gradient of the generator (as shown in Fig. 2.4). Consequently, sometimes the objective function that can provide a larger gradient used could be used during the initial stages.

$$\min_{\phi} \mathbb{E}_{z \sim p_z} - \log D(G(z)) \tag{2.6}$$

We call this the unsaturated form (the discriminant loss function used above is called the saturated form). When actually using the sample training, the objective function is

$$\min_{\phi} \frac{1}{N} \sum_{i=1}^{N} - \log D\left(G\left(z^{(i)}\right)\right) \tag{2.7}$$

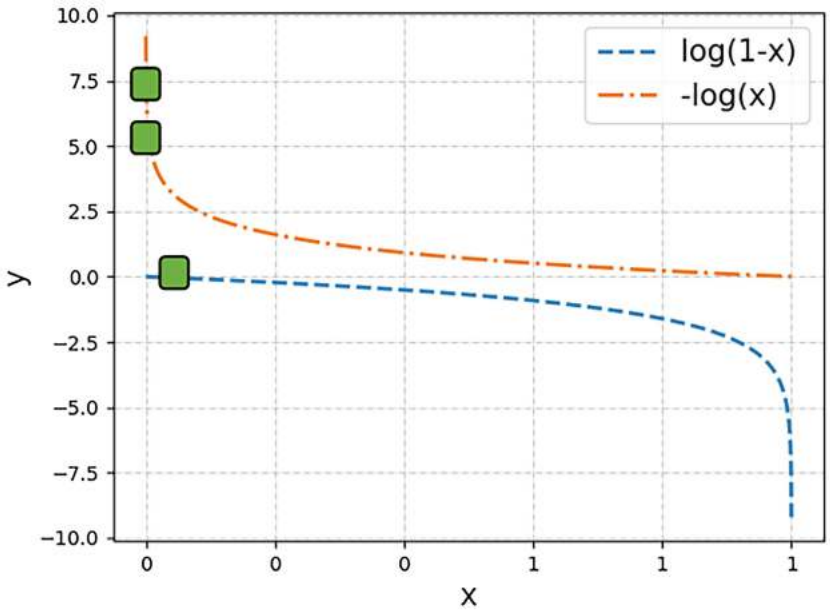


Fig. 2.4 Saturated form and unsaturated form function curves

2.1.3 Nature of GAN

In order to delve into the essence of GAN, we conducted theoretical analysis on it. Firstly, during each iteration process, the optimal discriminant D^* can be computed, simply by making the first-order derivative of the objective function equal 0, then

$$D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)} \tag{2.8}$$

Proof: First, we use a transformation without proof:

$$p_g(G(z))dx = p_z(z)dz \tag{2.9}$$

The objective function of the discriminator becomes:

$$\max_{\theta} \mathbb{E}_{x\sim p_{data}}[\log D(x)] + \mathbb{E}_{x\sim p_g}\left[\log\left(1 - D(x)\right]\right. \tag{2.10}$$

such that its first-order derivative is zero, then:

$$\frac{\partial \mathbb{E}_{x\sim p_{data}}[\log D(x)] + \mathbb{E}_{x\sim p_g}[\log\,(1 - D(x)]}{\partial\theta} = 0 \Leftrightarrow \frac{\partial\{\int_x p_{data}(x)[\log D(x)]dx + \int_x p_g(x)[\log\,(1 - D(x)]}{\partial\theta}$$

When the discriminator reaches optimality, the objective function of the generator

$$\min_{\phi} \mathbb{E}_{x\sim p_{data}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log\,(1 - D(G(z)))] \tag{2.12}$$

can be rewritten as:

$$\min_{\phi} 2D_{JS}(p_{data}(x) \parallel p_g(x)) - \log 4 \tag{2.13}$$

Calculation steps:

$$\min_{\phi} \mathbb{E}_{x\sim p_{data}}[\log D^*(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D^*(G(z)))] = \min_{\phi} \mathbb{E}_{x\sim p_{data}}[\log D^*(x)] + \mathbb{E}_{x\sim p_g}[\log(1 - D^*(x))]$$

JS divergence is a common way to measure the difference between two probability distributions. To explain JS divergence a little, let's start with a familiar fact: on a two-dimensional plane, each point represents an element, and the distance between points (the Euclidean distance) can be calculated by the Pythagorean theorem. The distance between(1, 0) and (3, 0) is definitely greater than the distance between (0, 1) and (1, 0). In fact, element is an abstract concept, points on the plane can be regarded as elements, and matrices, polynomials, functions can also be regarded as elements. Similar to the example just now, if each probability distribution $p(x)$ is also considered as an element (as shown in Fig. 2.5), the distance between probability distributions can be calculated using the JS divergence:

$$JS(p_1(x) \parallel p_2(x)) = \frac{1}{2}\int p_1(x)\log\frac{2p_1(x)}{p_1(x)+p_2(x)}dx + \frac{1}{2}\int p_2(x)\log\frac{2p_2(x)}{p_1(x)+p_2(x)}dx \tag{2.15}$$

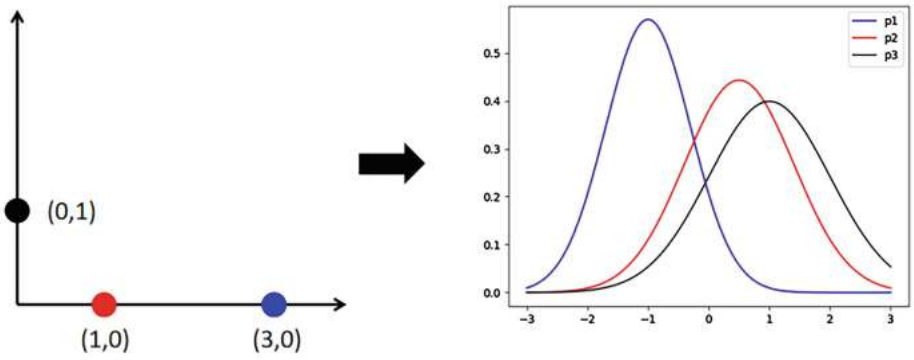


Fig. 2.5 KL dispersion interpretation

Correspondingly, the smaller the JS divergence, the more similar the two probability distributions are, while the larger the JS divergence, the more the two probability distributions differ. The JS divergence gets 0 when the two distributions are identical. As shown in Fig. 2.5, the calculation shows that $JS(p_1(x) \parallel p_2(x)) > JS(p_2(x) \parallel p_3(x))$.

It can be seen that the GAN is essentially trained discriminators first to get the JS divergence between p_{data} and p_g, and then train the generator to minimize the JS divergence. The generator reaches the global optimum when the JS divergence is 0, i.e., $p_{data} = p_g$. Theoretically, it can also be proved that GAN can achieve the global optimum when the generator and discriminator have sufficient capacity and the discriminator can reach the optimal solution for a given generator, which is of course almost impossible in practice.

In addition, for the non-saturated form objective function of the generator, again under the optimal discriminator D^* condition, the objective function becomes

$$\min_{\phi} D_{KL}(p_g(x) \parallel p_{data}(x)) - 2D_{JS}(p_{data}(x) \parallel p_g(x)) \tag{2.16}$$

Calculation steps:

$$\min_{\phi} -\mathbb{E}_{z\sim p_z}\log[D^*(G(z))] = \min_{\phi} -\mathbb{E}_{x\sim p_g}\log[D^*(x)] = \min_{\phi} -\mathbb{E}_{x\sim p_g}\log\left[\frac{p_g(x)}{p_{data}(x)+p_g(x)}\right] + \mathbb{E}_{x\sim p_g}\log$$

There is a point of contradiction: while the non-saturating form of the generator aims to maximize the JS divergence, it also minimizes the KL divergence simultaneously. These are two opposite optimization directions. However, from a practical perspective, this approach does indeed avoid the problem of training gradient saturation to a certain extent.

2.2 LSGAN

2.2.1 Gradient Disappearance

During the training of GAN, the problem of vanishing gradients in the generator often arises. Generally speaking, in deep learning, the error calculated based on the loss function is needed to guide the update optimization of deep network parameters by backpropagation. For example, for a simple neural network containing three hidden layers, when the gradient disappearance occurs, the hidden layers close to the output layer are relatively normal when the weights are updated because their gradients are relatively normal. But when the closer to the input layer, as the gradient vanishes, the weights of the hidden layers close to the input layer are updated slowly or stagnantly due to the gradient vanishing phenomenon. This results in only the later layers effectively learning during the training process, while the shallow layers fail to learn adequately.

The issue of gradient vanishing in GAN is a little different from the concern of the gradient vanishing problem mentioned above. Because the GAN includes two neural networks, the generator G and the discriminator D. The GAN is trained by alternating the generator and the discriminator. When the parameters of the discriminator are fixed and the generator is being trained, the gradients of the generator's parameters are nearly zero, which means that the discriminator is not providing any useful information for improving the generator. In this case, the generator fails to improve, and the training process stagnates, which is the gradient vanishing problem in GAN. The general gradient vanishing problem discusses that the shallow network near the input layer cannot obtain the gradient information (as shown in Fig. 2.6), while GAN mostly refers to the discriminator's inability to provide gradient information to the generator (as shown in Fig. 2.7).

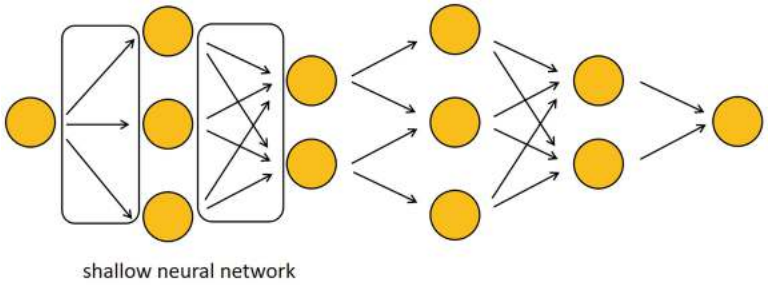


Fig. 2.6 Gradient disappearance of the depth network

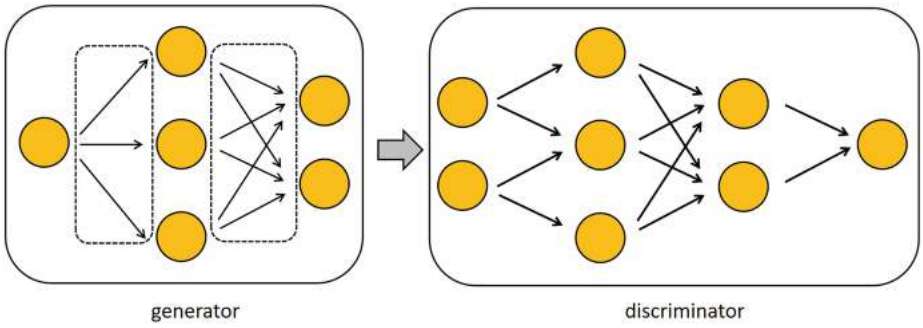


Fig. 2.7 Gradient disappearance of GAN

Why does the generator produce gradient disappearance? LSGAN argues that this is because for those samples that are correctly classified but far from the true distribution p_{data} but do not impose any penalty. Here, we try to describe this idea from two perspectives [2].

As shown in Fig. 2.8, when the discriminator is fixed, the decision surface $D(x) = 0.5$ is fixed, and for simplicity, we use a straight line to represent the decision surface. For the samples in the upper left of the decision surface, the $D(x)$ is <0.5, while for the samples on the lower right side of the decision surface, the $D(x)$ is >0.5, and the further away from the decision surface, the greater the deviation of the value from 0.5. Therefore, for the current discriminator, the more the sample is on the upper left, the lower the probability

that the discriminator thinks the sample is from the training dataset; and the further the sample is to the bottom right, the closer the value is to 1, i.e., the higher the probability that the discriminator thinks the sample is from the training dataset. At this point, consider the red triangular samples in the lower right corner, which are generated by the generator and are far away from the decision surface, so the generator has a high degree of confidence in them. However, these samples are also significantly distant from the training dataset. When we train the generator with the red triangular samples, these samples can successfully deceive the discriminator, resulting in the discriminator not conveying any improvement information to the generator, which generates gradient disappearance, and the worst is $p_g(x)$ and $p_{data}(x)$ are still far apart.

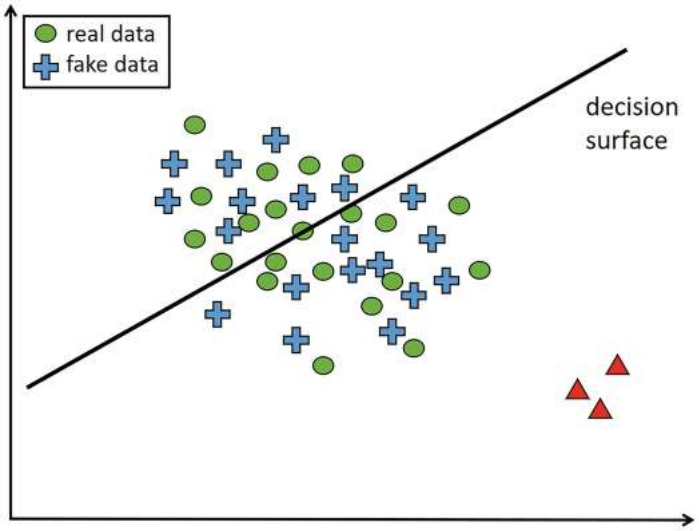


Fig. 2.8 Sample distribution when discriminator is fixed

Alternatively, we can describe the problem from the point of view of the objective function. In the standard form of GAN, the last layer of the discriminator uses the Sigmoid activation function is

$$f(y) = \frac{1}{1 + e^{-y}} \tag{2.18}$$

As shown in Fig. 2.9, here we need to split the linear operation and activation function operation of the last neuron of discriminator *D*. For the input sample *x*, after neural network of the last layer (the last layer has only one neuron), we get the feature after the linear operation of this neuron *y*. Then, the activation function Sigmoid operation is used to obtain $\sigma(y)$. Meanwhile, the objective function of the generator in the standard form GAN is

$$\min_{G_\theta} \mathbb{E}_{z \sim p_z} - \log D(G(z)) \tag{2.19}$$

For any characteristic *y*, its function curve is

$$-\log \frac{1}{1 + e^{-y}} \tag{2.20}$$

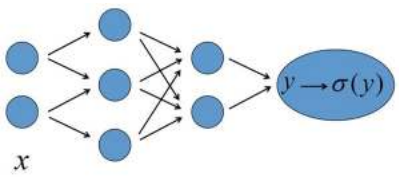


Fig. 2.9 Internal structure of the discriminator

As shown by the red triangular samples in Fig. 2.10, when the value of $\boldsymbol{\sigma(y)}$ is large, it indicates that the discriminator believes there is a high probability that the input sample comes from the training dataset. When training the generator using these samples, the objective function plateaus numerically, resulting in very small gradients. This easily leads to gradient vanishing, which means that the samples generated by the generator have high confidence for the discriminator, regardless of whether the samples truly conform to the probability distribution of the training set. Consequently, the generator does not evolve through its gradients, irrespective of whether the samples generated have high confidence for the discriminator.

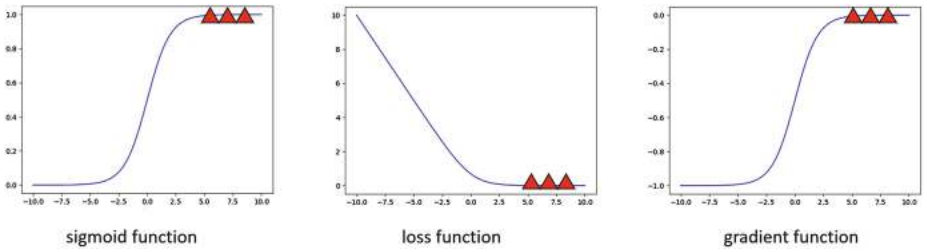


Fig. 2.10 Training with triangular samples and the corresponding gradients

Alternatively, if another objective function of the generator is used:

$$\min_{G_\theta} \mathbb{E}_{z\sim p_z} \log\left(1 - D(G(z))\right) \tag{2.21}$$

As above, for an arbitrary sample, the curve of

$$\log\left(1 - \frac{1}{1+e^{-y}}\right) \tag{2.22}$$

When the value of $\sigma(y)$ is very small, it means that the discriminator considers that the input samples have a small probability to come from the training dataset, as shown in the purple diamond samples in Fig. 2.11. When training the generator using these samples, the objective function may reach a numerical plateau, causing the gradients to become very small, which can easily lead to gradient vanishing. This means that even if the generator produces samples with low confidence, the discriminator will not provide gradients to "drive" the generator to avoid generating these poor-quality samples.

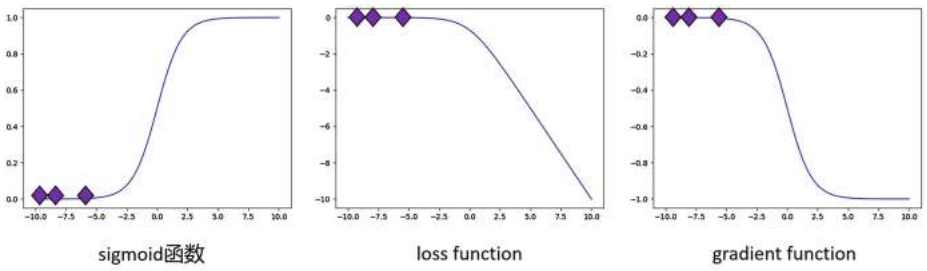


Fig. 2.11 Training with diamond-shaped samples and the corresponding gradients

2.2.2 LSGAN Design

At this point, it can be found that the gradient disappearance problem limits the self-evolution of the generator and prevents the discriminator and generator from achieving the optimal solution. To address this issue, LSGAN proposes a new loss function: the least squares loss function, which can be used to penalize the samples generated by the generator that are far away from the decision surface, essentially encouraging the samples to move closer to the decision boundary. By doing so, the LSGAN aims to avoid the gradient vanishing problem, as illustrated by the red samples in Fig. 2.8.

In LSGAN, for discriminator, the labels of the samples from training dataset are b, the generator generates samples with the labels a, then its objective function is

$$\min_{\theta} \frac{1}{2}\mathbb{E}_{x\sim p_{data}}\left[(D(x)-b)^2\right]+\mathbb{E}_{z\sim p_z}\left[(D(G(z))-a)^2\right] \tag{2.23}$$

For the generator, the objective function is

$$\min_{\phi} \frac{1}{2}\mathbb{E}_{z\sim p_z}\left[(D(G(z))-c)^2\right] \tag{2.24}$$

where c represents the value that the generator wants the discriminator to believe. Usually for the a, b, c value, we have two proposals. In the first scheme, let $b = c = 1$, and $a = 0$, which using a 0-1 encoding scheme to make the samples generated by the generator as realistic as possible. In the second scheme, set $a = -1$, $b = 1$, $c = 0$. It can be shown that when $b - c = 1$ and $b - a = 2$, the generator optimizes the Pearson chi-squared divergence (a kind of f-divergence) between the $p_{data} + p_g$ and $2p_g$. In practice, the ReLU function and the LeakyReLU function are usually chosen for the activation function of LSGAN, and it is found that these two schemes exhibit similar performance.

2.3 EBGAN

In 2006, Yang Lecun first proposed the concept of energy-based model in machine learning [3]. Essentially, an energy model is an energy function $U(x)$ that maps each sample in the sample space to an energy scalar value that is related to probability density function of the sample, as follows:

$$p(x) = \frac{e^{-U(x)}}{Z} \tag{2.25}$$

Where Z is the normalization constant. We all know that in physics, low-energy matter tends to be stable, while high-energy matter tends to be unstable, and electrons spontaneously jump from high-energy states to low-energy states. Inspired by this, for example, in supervised learning, for a sample x in the training set, if the label y of (x, y) is correct, then assign (x, y) with lower energy, and if the label y is incorrect, then a higher energy is assigned to (x, y). Therefore, the energy model learning task is to learn a "good" energy function $U(x)$. Similarly, in unsupervised learning, low energy should be placed on the training data p_{data} and higher energy should be placed elsewhere, as shown in Fig. 2.12.

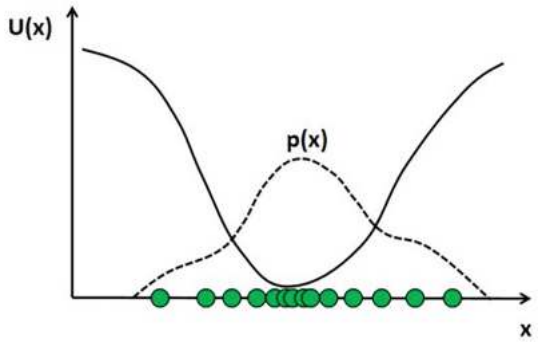


Fig. 2.12 Schematic diagram of the energy function

EBGAN is a successful attempt to apply an energy model to GAN. Let us first describe in layman's terms how to apply an energy model to a generative model [4]. The first step is to create an energy field $U(x)$, then place low-energy values in regions of high probability density and high-energy values in other regions, as shown in Fig. 2.13 on the left. Adjust the generator according to the energy field $U(x)$, making energy of generative samples reduce, as shown in Fig. 2.13 right.

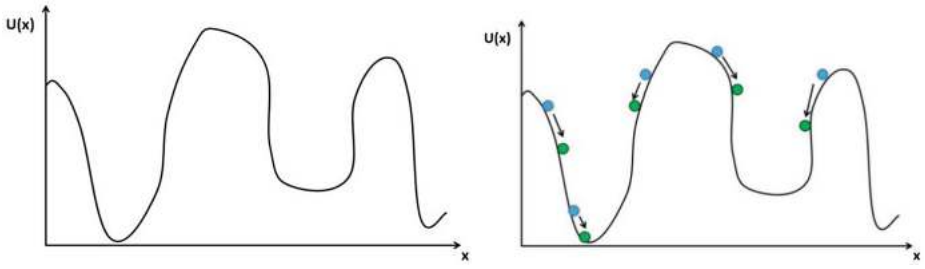


Fig. 2.13 Schematic diagram of EBGAN principle

In EBGAN, the discriminator plays the role of an energy function $U(x)$. For each input sample, the discriminator assigns the energy value $D(x)$, and the function of the generator is still to generate samples randomly. During the iterative training process, it is expected that the discriminator will assign as low an energy as possible (with the minimum energy being 0) to the samples in the training dataset, and as high an energy as possible to the samples from the generator. It should be noted that the convergence speed will be slowed down if the maximum assigned energy of the discriminator is not numerically limited. To avoid assigning infinite energy to the generated samples from the generator, the upper energy limit is set to m, i.e., the discriminator objective function is

$$\min_D \mathbb{E}_{x\sim p_{data}}[D(x)] + \mathbb{E}_{z\sim p_z}[m - D(G(z))]^+ \tag{2.26}$$

Among them $[\cdot] = \max(0, \cdot)$. So the discriminator is actually "shaping" the energy function $U(x)$. Naturally, the training goal of the generator is to generate samples with as small an energy value as possible, i.e., the objective function is:

$$\min_G \mathbb{E}_{z\sim p_z}[D(G(z))] \tag{2.27}$$

In EBGAN, the discriminator is not a simple fully connected network or a convolutional network, but an autoencoder consisting of an encoder and a decoder, plus an MSE error calculation layer, as shown in Fig. 2.14:

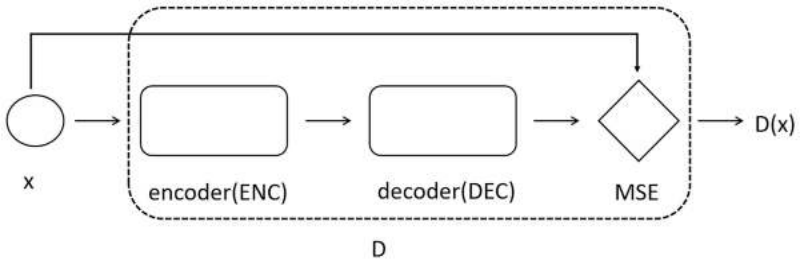


Fig. 2.14 Schematic diagram of EBGAN discriminator

For the input sample, it first pass through an encoder (ENC) to obtain the coded representation of the sample $ENC(x)$ and then the coded representation is fed to the decoder (DEC) to obtain the reconstruction of sample $DEC(ENC(x))$. Calculate the reconstruction error (EBGAN selects the mean square error), and use this error as the energy value of the sample, i.e.,

$$D(x) = \| x - DEC(ENC(x)) \|^2$$

In fact, EBGAN is very closely related to regularization autoencoder. Putting aside EBGAN for a moment, just for a autoencoder, it is important to avoid it to be a simple constant mapping, i.e., for any sample x, all have $DEC(ENC(x)) = x$, which means that the autoencoder does not learn the hidden variable representation of the sample or extract the features of the sample, but only copies the sample exactly. Therefore, it is often necessary to add some constraint regularization terms to the autoencoder to ensure that it can only approximate and replicate inputs similar to the training data, rather than all inputs. These constraints force

the autoencoder to decide which parts of the input data should be prioritized for replication. In EBGAN, the generator serves as a regularization term for the autoencoder since the autoencoder is very explicitly instructed to replicate as many samples as possible from the training dataset and as few samples as possible from the generator. And using a trainable generator neural network as a regularization term provides more flexibility than a manually set regularization term.

Further, for the discriminator's autoencoder model, in order to prevent it from producing samples clustered in only one or a few patterns of p_{data}, EBGAN adds a PT (Pulling Away) regularization term to the encoder. For samples in a batch that are expected to be different from each other after pass through the encoder, this term is quantified using the cosine similarity, i.e.:

$$f_{PT}(S) = \frac{1}{N(N-1)} \sum_{i} \sum_{i \neq j} \left(\frac{S_i^T S_j}{\| S_i \| \| S_j \|} \right)^2 \tag{2.28}$$

Where S denotes the sample set of a batch. The EBGAN that uses the PT regularization term is called EBGAN-PT.

Finally, when training EBGAN, here are a few hints:

(1) For the discriminator's objective function, which consists of two terms, the first term for the samples of the training dataset and the second term for the samples generated by the generator, then the value domain of the second term is restricted to $[0, m]$, and so as to the first term. But it is not restricted by the $[\cdot]^+$ function and may exceed m. In theory, the upper bound on the value domain of the first term depends on the capacity of the neural network and the complexity of the dataset.

(2) In practice, when training EBGAN, the discriminator (autoencoder model) can be trained on the training dataset alone first, and when the loss function of the discriminator converges, the value of the loss function probably indicates how well the autoencoder model fits the dataset, at which point begin to search for the hyperparameters m.

(3) Hyperparameter m needs to be chosen carefully. If the value is too large, it is likely to cause difficulties and instability in training, while if the value is too small, it is likely to cause distortion and ambiguity in the final generated samples.

(4) At the start of the training process, we can choose a relatively large value for m and then gradually decrease it until it eventually decays to 0.

2.4 fGAN

GAN essentially learns the distance metric between probability distribution of training dataset p_{data} and generated dataset p_g, and then minimize the distance metric to achieve the final result $p_{data} = p_g$. GAN can not only use JS divergence as the measure of distance between two probability distributions, but also can be replaced by KL divergence, total variance distance, Wasserstein distance, etc., as long as it can reasonably measure the distance of the distribution. Some of these distance measures are included in the framework of f-divergence.

In the fGAN [5], the f-divergence expression is defined as

$$D_f(p_{data} \parallel p_g) = \int_x p_g(x) f\left(\frac{p_{data}(x)}{p_g(x)} \right) dx \tag{2.29}$$

In this framework, it is possible to choose different $f(x)$ to obtain the corresponding different metrics, where the requirements:

(1) function $f(x)$ must be the mapping from positive real numbers to real numbers.

$$f(1) = 0.$$

(2) function $f(x)$ is convex, and the value of f-divergence reaches a minimum of 0 when the two distributions coincide exactly.

For example, in order to obtain the JS divergence, make

$$f(u) = -(u+1)\log\frac{1+u}{2} + u\log u \tag{2.30}$$

$$u = \frac{p_{data}(x)}{p_g(x)} \tag{2.31}$$

The correspondence relationships between the other metrics and $f(u)$ are shown in Table 2.1.

Table 2.1 Various metrics and corresponding expressions

Metric	Expression	$f(u)$
Total variance	$\frac{1}{2}\int \mid p_{data}(x) - p_g(x) \mid dx$	$\frac{1}{2} \mid u-1 \mid$
KL divergence	$\int p_{data}(x)\log\left[\frac{p_{data}(x)}{p_g(x)}\right]dx$	$u\log u$
Inverse KL divergence	$\int p_g(x)\log\left[\frac{p_g(x)}{p_{data}(x)}\right]dx$	$-\log u$
Pearson χ^2	$\int \frac{(p_g(x)-p_{data}(x))^2}{p_{data}(x)}dx$	$(u-1)^2$
Neyman χ^2	$\int \frac{(p_g(x)-p_{data}(x))^2}{p_g(x)}dx$	$\frac{(u-1)^2}{u}$
Hellinger distance	$\int\left(\sqrt{p_{data}(x)} - \sqrt{p_g(x)}\right)^2 dx$	$\left(\sqrt{u}-1\right)^2$
Jeffrey distance	$\int(p_{data}(x) - p_g(x))\log\left[\frac{p_{data}(x)}{p_g(x)}\right]dx$	$(u-1)\log u$
JS divergence	$\frac{1}{2}\int p_{data}(x)\log\left[\frac{2p_{data}(x)}{p_g(x)+p_{data}(x)}\right]dx + \frac{1}{2}\int p_g(x)\log\left[\frac{2p_g(x)}{p_g(x)+p_{data}(x)}\right]dx$	$-(u+1)\log\frac{1+u}{2} + u\log u$
α divergence	$\frac{1}{\alpha(\alpha-1)}\int p_{data}(x)\left[\left(\frac{p_g(x)}{p_{data}(x)}\right)^{\alpha} - 1\right] - \alpha(p_g(x) - p_{data}(x))dx$	$\frac{1}{\alpha(\alpha-1)}(u^{\alpha} - 1 - \alpha(u-1))$

Since it is not known about expressions (or approximate expressions) of p_{data} and p_g, so it is not possible to calculate the f-divergence directly. Based on the training dataset $\{x^{(1)}, x^{(2)}, ..., x^{(N)}\}$ and the sample set $\left\{x_G^{(1)}, x_G^{(2)}, \ldots, x_G^{(N)}\right\}$ generated by the generator, using the conjugate function, we can obtain the f-divergence estimate by training a neural network $T(x)$.

Define $g(t)$ as the conjugate function of function $f(u)$:

$$g(t) = \max_{u\in \mathrm{dom}\ f}\{ut - f(u)\} \tag{2.32}$$

Where definition domain of the u is $f(u)$, and $g(t)$ is proved to be a convex function. A lower bound on the estimated f -divergence can be obtained as

$$\max_T \mathbb{E}_{p_{data}}[T(x)] - \mathbb{E}_{p_g}[g(T(x))] \tag{2.33}$$

Proof: According to

$$g(t) = \max_{u\in \mathrm{dom}\ f}\{ut - f(u)\} \tag{2.34}$$

There are

$$f(u) = \max_{t\in \mathrm{dom}\ g}\{tu - g(t)\} \tag{2.35}$$

Combining these two, there can be:

$$D_f(p_{data} \parallel p_g) = \int_x p_g(x) f\left(\frac{p_{data}(x)}{p_g(x)}\right) dx = \int_x p_g(x) \left\{ \max_{u \in \mathrm{dom}\, g} \left\{ t \frac{p_{data}(x)}{p_g(x)} - g(t) \right\} \right\} dx = \int_x \max_{u \in \mathrm{dom}\, g} \{ t p_d$$

For any, it is necessary to find the optimal t, that is

$$t^* = f'(u) \tag{2.37}$$

Therefore, for any, there is also an optimal t^* corresponding to which the two have a complex analytic relationship. We can fit this relationship using a neural network:

$$t = T(x) \tag{2.38}$$

In this way, when calculating f-divergence, we convert the process of solving for maximization into a training process for the neural network. Since the neural network is parameterized, its output values should be $g(t)$. The inequality sign is used since the neural network is parameterized and its output values should be a subset of the value domain. By once again swapping the order of the maximum and integral, and using the inequality sign again here, f-divergence can be expressed as:

$$D_f(p_{data} \parallel p_g) \geq \int_x \max_T \{ T(x) p_{data}(x) - g(T(x)) p_g(x) \} dx = \max_T \mathbb{E}_{p_{data}}[T(x)] - \mathbb{E}_{p_g}[g(T(x))] \tag{2.39}$$

Furthermore, according to the definition of conjugate function, it is obtained that

$$t = f'(u) \tag{2.40}$$

It is necessary to limit the output of the neural network $T(x)$ so that keep it within the range of the first derivative of $f(u)$.

The expression of the conjugate function $g(t)$ is determined by $f(u)$, and the derivation process has some limitations on the range of $T(x)$ values. For example, when choosing inverse KL divergence as the metric, $f(u) = -\log u$, and we can calculate $g(t) = -1 - \log(-t)$, and the range of $T(x)$ is correspondingly limited to $(-\infty, 0)$. Simply set the activation function of $T(x)$ to $-e^x$, The operation of selecting other metrics is similar and is summarized in Table 2.2.

Table 2.2 Various forms of metrics

Metric	$f(u)$	$g(t)$	$f'(\mathbb{D})$	Activation function
Total variance	$\frac{1}{2} \mid u-1 \mid$	t	$-\frac{1}{2} \leq t \leq \frac{1}{2}$	$\frac{1}{2} \tanh(x)$
KL divergence	$u \log u$	$e^t - 1$	$\mathbb{R}$	x
Inverse KL divergence	$-\log u$	$-1 - \log(-t)$	$\mathbb{R}_-$	$-e^x$
Pearson χ^2	$(u-1)^2$	$\frac{1}{4}t^2 + t$	$\mathbb{R}$	x
Neyman χ^2	$\frac{(u-1)^2}{u}$	$2 - 2\sqrt{1-t}$	$t < 1$	$1 - e^x$
Hellinger distance	$\left(\sqrt{u}-1\right)^2$	$\frac{t}{1-t}$	$t < 1$	$1 - e^x$
Jeffrey distance	$(u-1)\log u$	$t - 2 + W\left(e^{1-t}\right) + \frac{1}{W(e^{1-t})}$	$\mathbb{R}$	x
JS divergence	$-(u+1)\log\frac{1+u}{2} + u\log u$	$-\log(2-e^t)$	$t < \log 2$	$-\log(1+e^{-x}) + \log 2$

At this point, the discriminator $T(x)$ (we still use the name discriminator, but in fact it no longer has an intuitive interpretation of discriminating truth, it is just a neural network with a specific activation function) has an objective function of

$$\max_T \mathbb{E}_{x \sim p_{data}}[T(x)] - \mathbb{E}_{z \sim p(z)}[g(T(G(z)))] \tag{2.41}$$

Generator's training objective is naturally to minimize the learned f-divergence, so the objective function is

$$\min_G \mathbb{E}_{x\sim p_{data}}[T(x)] - \mathbb{E}_{z\sim p(z)}[g(T(G(z)))] \tag{2.42}$$

Compared with the original GAN, the principle of fGAN is similar: both learn the distance between distributions and then training the generator with the distance as the objective function, except that it is more generalized and can choose different metrics to derive different GAN. For example, under some specific conditions, the generator objective function of LSGAN is Pearson χ^2, and the generator objective function of EBGAN is Total Variance.

2.5 WGAN

2.5.1 Wasserstein Distance

WGAN is a typical representative based on IPM, which uses Wasserstein distance with better mathematical properties to solve the gradient vanishing problem in standard GAN, and also has excellent performance in practice. This section will first explain the Wasserstein distance and discuss its performance differences with f-divergence, and then use the duality method to derive the final form of the objective function of WGAN.

For the training dataset $p_{data}(x)$ and the generated dataset $p_g(x)$, the distance between two probability distributions can be measured using KL divergence, JS divergence, total variance, etc. Naturally, we expect the magnitude of the distance value to accurately reflect the degree of difference between two distributions. When the two distributions are far apart and dissimilar, the distance value should be larger; conversely, when the two distributions are close and relatively similar, the distance value should be smaller. Only by training the discriminator to get accurate distance information, the distance between distributions can be reduced by the process of continuously optimizing the generator, and finally reach $p_{data} = p_g$.

However, metrics such as KL divergence and JS divergence do not satisfy the above requirements, and it cannot accurately indicate the difference between two distributions. For example, for two one-dimensional uniform distributions in the plane, $P(x,y)$ is a uniform distribution from $(0,0)$ to $(0,1)$, and $Q(x,y)$ is a uniform distribution between $(\theta,0)$ and $(\theta,1)$, as shown in Fig. 2.15 below:

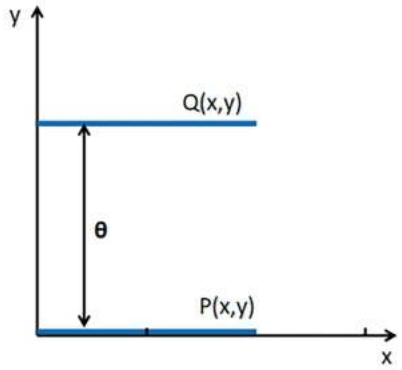


Fig. 2.15 Distribution of P and Q

Calculate the JS divergence with

$$JS(P \parallel Q) = \begin{cases} \log 2 & \theta \neq 0 \\ 0 & \theta = 0 \end{cases} \tag{2.43}$$

It can be found that when θ is not zero, the value of JS divergence is always log2, i.e., regardless of the distance between $P(x,y)$ and $Q(x,y)$, the JS divergence remains constant. Only when the two distributions completely overlap, does the value of JS divergence suddenly become 0. In the standard GAN, the generator learning target is min $JS(p_{data} \parallel p_g)$. At this time, the generator cannot learn effectively, and no matter how

to adjust the weights, it cannot reduce the JS divergence value, specifically manifested as the gradient vanishing.

The root causes of the above problems are $P(x,y)$ and $Q(x,y)$ do not have a cross section, and if the $P(x,y)$ and $Q(x,y)$ produce a cross section of length θ of the cross section, as shown in Fig. 2.16.

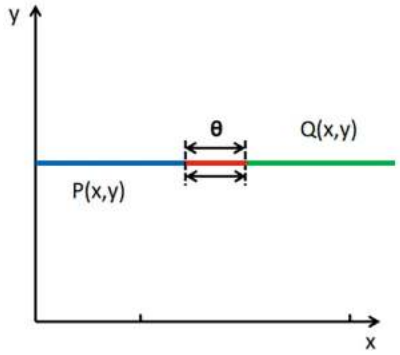


Fig. 2.16 P and Q crossover

Calculating the JS divergence, we have $JS(P \parallel Q) = (1 - \theta) \log 2$. The JS divergence at this time is the function of θ, which can serve as a good objective function for learning to generator. Unfortunately, the above bad situation happens widely in GAN. According to the law of manifold distribution, high-dimensional data of the same class in nature tend to be concentrated near some low-dimensional manifold, i.e., the set of p_{data} and p_g are just low-dimensional manifolds in a high-dimensional space, and the intersection of the two is almost non-existent, both of which will face the problem of "unlearned" generators. To take a less rigorous example, in a three-dimensional space, two squares of dimension two are low-dimensional manifolds, and in most cases, the two squares do not have a non-negligible intersection (as shown in the left image of Fig. 2.17). If the intersection of the two squares is a line segment, the line segment can be ignored because it is small enough compared to the whole square (as shown in the middle image of Fig. 2.17). There is a certain possibility that the two squares produce a non-negligible intersection part (as shown in the right image of Fig. 2.17), but the probability of this situation becomes smaller and smaller as the space dimension increases, so the situation can also be ignored when facing the actual problem. In summary, there is almost no non-negligible intersection between p_{data} and p_g.

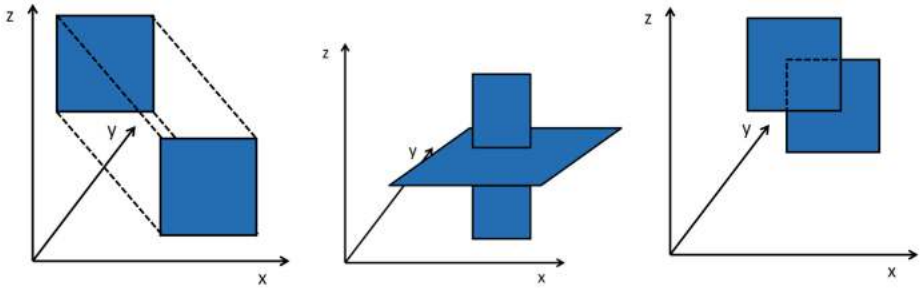


Fig. 2.17 Schematic diagram of several distributions of flow patterns

Similarly, KL divergence, total variance distance also have similar problems. One solution to solve this problem is adding Gaussian noise on p_{data} and p_g during training, so that diffuse them to the whole high-dimensional space and produce a non-negligible cross section, and gradually reduce the variance of Gaussian noise as training proceeds, but the fundamental approach makes choosing a more reasonable metric.WGAN elegantly solves the above problem using Wasserstein distance instead of JS divergence [6]. For probability distribution $p(x)$ and $q(y)$, define

$$W[p(x), q(y)] = \inf_{\gamma \sim \Pi(p,q)} \iint \gamma(x,y) \mid x - y \mid dxdy \tag{2.44}$$

Among them, the $\gamma(x,y)$ is the joint distribution of $p(x)$ and $q(y)$, and

$$\int \gamma(x,y)dy = p(x) \tag{2.45}$$

$$\int \gamma(x,y)dx = q(y) \tag{2.46}$$

And $\Pi(p,q)$ indicates the set of all possible joint distributions constituted of $p(x)$ and $q(y)$. Above calculation process is choosing the optimal one among all possible joint distributions of $\gamma(x,y)$, such that $\iint \gamma(x,y) \mid x-y \mid dxdy$ is the minimum value, and this value is the Wasserstein distance of $p(x)$ and $q(y)$.Let's take a simple example of a discrete distribution to demonstrate the calculation process in detail, assuming that the random variables x, y can only take on values of $\{1, 2, 3, 4\}$, the probability distributions of $p(x)$ and $q(y)$ are as follows:

	1	**2**	**3**	**4**
***p*(*x*)**	0.25	0.25	0.5	0
***q*(*y*)**	0.5	0.5	0	0

There are various possibilities for their joint distribution, such as

p(x)	***q(y)***			
	1	**2**	**3**	**4**
1	0.25	0	0.25	0
2	0	0.25	0.25	0
3	0	0	0	0
4	0	0	0	0

or

p(x)	***q*(*y*)**			
	1	**2**	**3**	**4**
1	0.25	0.25	0	0
2	0	0	0.5	0
3	0	0	0	0
4	0	0	0	0

If we consider the probability values as "goods," each of the joint distributions here actually represents a certain transportation scheme, i.e., how to transport $p(x)$ into $q(y)$. Taking the second joint distribution as an example, the first column represents keeping the probability value of $p(x = 1) = 0.25$ at the position of $x = 1$, the second column represents moving the probability value of $p\ (x = 2) = 0.25$ to the position of $x = 1$, and the third column represents moving the probability value of $p\ (x = 3) = 0.5$ to the position of $x = 2$. After the above transportation, $p\ (x)$ is exactly the same as $q\ (y)$. Meanwhile, considering that the distance between any two positions x and y is $|x - y|$, there is a "unit price table":

x	***y***			
	1	**2**	**3**	**4**
1	0	1	2	3
2	1	0	1	2
3	2	1	0	1
4	3	2	1	0

Multiplying the unit price table with the corresponding elements of the transportation scheme table and then summing them up, we get all the costs of this transportation scheme. Since different transportation options lead to different costs, among all transportation options, the least costly option is selected and its cost value is the Wasserstein distance of $p(x)$ and $q(y)$.

The Wasserstein distance has better mathematical properties than measures such as JS divergence and KL divergence. It is continuous everywhere and is derivable almost everywhere. Using the example at the beginning of this section, using the Wasserstein distance we have $W(P \parallel Q) = \theta$. It can be seen that the Wasserstein distance clearly indicates the distance between P and Q. Furthermore, it can provide good distance information to the generator, which can give the exact gradient direction of reducing distance of P and Q, and elegantly avoids the problem of unclear distance indication of JS divergence.

2.5.2 WGAN Design

For p_{data} and p_g, using some mathematical tricks, their Wasserstein distances can be written as

$$W[p_{data}(x), p_g(x)] = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim p_{data}}[f(x)] - \mathbb{E}_{x \sim p_g}[f(x)] \tag{2.47}$$

where $\|f\|_L \leq 1$ denotes $f(x)$ satisfies the 1-Lipschitz limit, i.e., for any x and y, all have $|f(x) - f(y)| \leq |x - y|$.

To compute the Wasserstein distance, we need to iterate through all joint probability distributions that satisfy the conditions, then calculate the total cost under each joint probability distribution, and finally take the smallest total cost value. Otherwise, it is almost unsolvable when the dimensionality is high. Somewhat similar to the previous fGAN, when an optimization problem is difficult to solve, it can be considered to be transformed into a dual problem that is easier to solve. Each linear programming problem has a corresponding dual problem, which is constructed based on the constraints and objective function of the original problem. Accordingly, we first represent the Wasserstein distance as a linear programming form by defining the vector Γ (i.e., "discretizing" the joint probability distribution and pulling it into column vectors on a position-by-position basis)

$$\Gamma = \begin{bmatrix} \gamma(x_{data1}, x_{g1}) \\ \gamma(x_{data1}, x_{g2}) \\ \cdots \\ \gamma(x_{data2}, x_{g1}) \\ \gamma(x_{data2}, x_{g2}) \\ \cdots \\ \gamma(x_{data3}, x_{g1}) \\ \gamma(x_{data3}, x_{g2}) \\ \cdots \\ \cdots \end{bmatrix} \tag{2.48}$$

Define the vector D is

(2.49)

$$D = \begin{bmatrix} d(x_{data1}, x_{g1}) \\ d(x_{data1}, x_{g2}) \\ \cdots \\ d(x_{data2}, x_{g1}) \\ d(x_{data2}, x_{g2}) \\ \cdots \\ d(x_{data3}, x_{g1}) \\ d(x_{data3}, x_{g2}) \\ \cdots \\ \cdots \end{bmatrix}$$

For two constraints, define the matrix A.

$$A = \begin{bmatrix} 1 & 1 & \cdots & | & 0 & 0 & \cdots & | & \cdots & | & 0 & 0 & \cdots & | & \cdots \\ 0 & 0 & \cdots & | & 1 & 1 & \cdots & | & \cdots & | & 0 & 0 & \cdots & | & \cdots \\ \cdots & \cdots & \cdots & | & \cdots & \cdots & \cdots & | & \cdots & | & \cdots & \cdots & \cdots & | & \cdots \\ \cdots & \cdots & \cdots & | & \cdots & \cdots & \cdots & | & \cdots & | & 1 & 1 & \cdots & | & \cdots \\ \cdots & \cdots & \cdots & | & \cdots & \cdots & \cdots & | & \cdots & | & \cdots & \cdots & \cdots & | & \cdots \\ 1 & 0 & \cdots & | & 1 & 0 & \cdots & | & \cdots & | & 1 & 0 & \cdots & | & \cdots \\ 0 & 1 & \cdots & | & 0 & 1 & \cdots & | & \cdots & | & 0 & 1 & \cdots & | & \cdots \\ \cdots & \cdots & \cdots & | & \cdots & \cdots & \cdots & | & \cdots & | & \cdots & \cdots & \cdots & | & \cdots \\ \cdots & \cdots & \cdots & | & \cdots & \cdots & \cdots & | & \cdots & | & 0 & 0 & \cdots & | & \cdots \\ \cdots & \cdots & \cdots & | & \cdots & \cdots & \cdots & | & \cdots & | & \cdots & \cdots & \cdots & | & \cdots \end{bmatrix} \tag{2.50}$$

Define vector b

$$b = \begin{bmatrix} p_{data}(x_{data1}) \\ p_{data}(x_{data2}) \\ \cdots \\ p_g(x_{g1}) \\ p_g(x_{g2}) \\ \cdots \\ \cdots \end{bmatrix} \tag{2.51}$$

Having defined these complex matrices and vectors, our Wasserstein distance can then be expressed in the form of the following linear programming

$$\min_{\Gamma} \{< \Gamma, D > | A\Gamma = b, \Gamma \geq 0\} \tag{2.52}$$

Dual theory is a very beautiful theory, especially for strong dual problems with:

$$\min_{x} \left\{ c^T x | Ax = b, x \geq 0 \right\} = \max_{y} \left\{ b^T y \middle| A^T y \leq c \right\} \tag{2.53}$$

In other words, by solving the dual problem of the original problem, we can obtain the solution to the dual problem simultaneously with solving the original problem. Even for the weak dual problem, the lower bound of the original problem is given although it cannot be solved exactly:

(2.54)

$$\min_{x}\left\{c^{T}x|Ax=b,x\geq 0\right\}\geq\max_{y}\left\{b^{T}y|A^{T}y\leq c\right\}$$

In the fGAN, we give a lower bound on the f-divergence, but fortunately, this time facing a strong dyadic problem:

$$\min_{\Gamma}\left\{<\Gamma,D>|A\Gamma=b,\Gamma\geq 0\right\}=\max_{F}\left\{<b,F>\left|A^{T}F\leq D\right.\right\} \tag{2.55}$$

For the dual problem of the original problem, we define the vector *F*:

$$F=\begin{bmatrix}f_{1}(x_{data1})\\f_{1}(x_{data2})\\\cdots\\f_{2}(x_{g1})\\f_{2}(x_{g2})\\\cdots\\\cdots\end{bmatrix} \tag{2.56}$$

According to its constraints, the requirement for any x_{datai} and any x_{gj}, both have:

$$f_{1}(x_{datai})+f_{2}(x_{gj})\leq d(x_{datai},x_{gj}) \tag{2.57}$$

Namely, there are

$$f_{1}(x)+f_{2}(x)\leq d(x,x)=0 \tag{2.58}$$

Further,

$$f_{2}(x)\leq f_{1}(x) \tag{2.59}$$

In summary, we have

$$W[p_{data}(x),p_{g}(x)]=\min_{\Gamma}\left\{<\Gamma,D>|A\Gamma=b,\Gamma\geq 0\right\}=\max_{F}\left\{<b,F>\left|A^{T}F\leq D\right.\right\}=\max_{f_{1},f_{2}}\left\{\int[p_{data}(x)\right.$$

Now, define a neural network *F*(*x*) to fit the previous equation of *f*(*x*), and using the sampling calculation, the loss function of discriminator (now called critic) in WGAN is

$$W[p_{data}(x),p_{g}(x)]=\sup_{\|f\|_{L}\leq 1}\mathbb{E}_{x\sim p_{data}}[f(x)]-\mathbb{E}_{x\sim p_{g}}[f(x)] \tag{2.61}$$

We use the neural network critic (which we previously called discriminator) to learn (*x*) , then the objective function of critic is

$$\max_{f_{w}}\mathbb{E}_{x\sim p_{data}}[f_{w}(x)]-\mathbb{E}_{z\sim p_{z}}[f_{w}(G(z))] \tag{2.62}$$

Among them $\|f_{w}\|_{L}\leq 1$, the objective function of the generator is:

$$\min_{G}-\mathbb{E}_{z\sim p_{z}}[f_{w}(G(z))] \tag{2.63}$$

We have utilized the Wasserstein distance to obtain the superior performance of WGAN, but at the same time introduced the problem of 1-Lipschitz constraint of the discriminator, which is certainly a very big challenge.

2.6 LS-GAN

Loss-Sensitive GAN (hereinafter referred to as LS-GAN) is proposed almost simultaneously with WGAN. Although they originate from different perspectives, both ultimately incorporate Lipschitz constraint to the discriminator [7]. This section will give a brief introduction of LS-GAN and compare its similarities and differences with WGAN, so that readers can have a more diversified understanding of Lipschitz constraint.

LS-GAN also consists of two parts, the generator and the discriminator. The generator, like most GAN, accepts uniform or normal noise input z and outputs samples x, and the parameters of this neural network are given by ϕ, which is denoted by $G_\phi(z)$. In LS-GAN, the input of the discriminator (denoted by θ parameterization) is a sample and output a loss function value. Note that although the term $L_\theta(x)$ loss function, it does not refer to the objective function used to train the neural network, but rather to a "scoring" function on the samples, where the loss function $L_\theta(x)$ should have a small loss value for samples from the training dataset and a larger loss value for samples from the generator$G_\phi(z)$. It also needs to satisfy the restriction that

$$L_\theta(G_\phi(z)) - L_\theta(x) \geq \Delta(x, G_\phi(z)) \tag{2.64}$$

where $\Delta(x, G_\phi(z))$ denotes the intervals between x and $G_\phi(z)$, that is $L_\theta(G_\phi(z))$ and $L_\theta(x)$ must have at least $\Delta(x, G_\phi(z))$ size interval. As shown in Fig. 2.18, the left part satisfies the restriction while the right part does not.

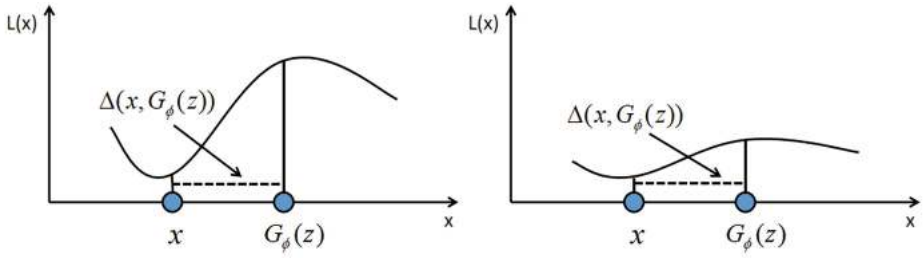


Fig. 2.18 Loss function L interval schematic

This constraint allows the two types of samples to be separated by the loss function. The training objective is giving small loss value for the samples from the training dataset and making two types of samples are separated by a certain interval. The hard interval constraint is written as a soft constraint to obtain the objective function:

$$\min_\theta \mathbb{E}_{x\sim p_{data}} L_\theta(x) + \lambda \mathbb{E}_{x\sim p_{data}, z\sim p_z}(\Delta(x, G_\phi(z)) + L_\theta(x) - L_\theta(G_\phi(z)))_+ \tag{2.65}$$

Among them $(a)_+ = \max(0, a)$. The goal of the generator is to hopefully generate the samples at $L_\theta(x)$ smaller positions, i.e.,

$$\min_\phi \mathbb{E}_{z\sim p_z} L_\theta(G_\phi(z)) \tag{2.66}$$

For the actual calculation, the above expression is approximated using empirical mean. For n individual noise $\{z^{(1)}, z^{(2)}, .., z^{(n)}\}$ and n samples from the training dataset $\{x^{(1)}, x^{(2)}, ..., x^{(n)}\}$, the empirical objective function of the discriminator is

$$\min_\theta \frac{1}{n}\sum_{i=1}^{n} L_\theta\left(x^{(i)}\right) + \frac{\lambda}{n}\sum_{i=1}^{n}\left(\Delta\left(x^{(i)}, G_\phi\left(z^{(i)}\right)\right) + L_\theta\left(x^{(i)}\right) - L_\theta\left(G_\phi\left(z^{(i)}\right)\right)\right)_+ \tag{2.67}$$

The empirical objective function of the generator is

(2.68)

$$\min_{\phi} \sum_{i=1}^{n} L_{\theta}\left(G_{\phi}\left(z^{(i)}\right)\right)$$

We discuss the second term of the discriminant's objective function. When $L_{\theta}(G_{\phi}(z)) - L_{\theta}(x) \geq \Delta(x, G_{\phi}(z))$, the second term is 0 and does not participate in the minimization of the objective function. When $L_{\theta}(G_{\phi}(z)) - L_{\theta}(x) \leq \Delta(x, G_{\phi}(z))$, the second term will appear in the objective function to minimize, i.e., pull down the loss value of x and pull up the loss value of $G_{\phi}(z)$, as shown in Fig. 2.19.

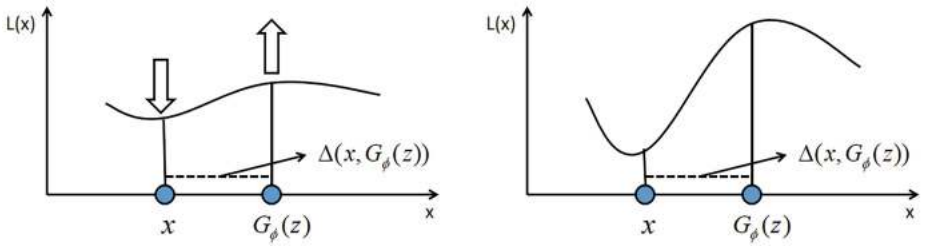


Fig. 2.19 Principle of LS-GAN

It should be noted that LS-GAN imposes the Lipschitz constraint in the same way as WGAN-GP, both using the soft method of gradient penalty, where LS-GAN adds a regularization term to the loss function $L_{\theta}(x)$. The addition of the regularization term to the objective function is

$$\frac{1}{2}\mathbb{E}_{x \sim p_{data}} \| \nabla_x L_{\theta}(x) \|^2 \tag{2.69}$$

However, in details, WGAN requires a 1-Lipschitz constraint across the entire space, while WGAN-GP simplifies this by making the gradient of the discriminator's output relative to its input close to 1 at intermediate samples. In contrast, the Lipschitz constraint of LS-GAN is imposed on the manifold of the real data, so it is only computed on p_{data}. The regularization term is not centered on 1 as in WGAN, but the gradient of the loss function respect to input is expected to be close to 0. This is due to the fact that the number of samples required for convergence is proportional to the gradient in the generalizability proof of LS-GAN.

LS-GAN training has some "on-demand" capability. For example, there is one real sample x_r from the training set and two samples x_{g-f} and x_{g-n} generated by the generator, where x_{g-n} are close to x_r, but x_{g-f} and x_r are farther away. After the completion of training loss function, the $L_{\theta}(x_{g-f})$ is at least greater than $\Delta(x_{g-f}, x_r) + L_{\theta}(x_r)$ and $L_{\theta}(x_{g-n})$ is approximately equal to $L_{\theta}(x_r)$, the generator will focus more on x_{g-f} so that make it lean toward x_r.

We give a brief introduction to the idea of LS-GAN. In the original GAN, the distribution of the training dataset p_{data} does not have any restrictions and there is no prior knowledge, then when the distribution p_{data} is very complex, in order to achieve the optimal in $D(x)$

$$D(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)} \tag{2.70}$$

It is required that the discriminator possesses infinite modeling capacity, allowing the model to fit arbitrarily complex functions. With infinite modeling capacity, when the overlap between the two manifolds is negligible (which is highly plausible in practice), the JS divergence becomes constant, leading to vanishing gradients. WGAN chooses to use Wasserstein distance to improve the problem, but LS-GAN directly improves on the infinite modeling capacity of the discriminator. LS-GAN assumes that the distribution of the training dataset $p_{data}(x)$ is satisfying k-Lipschitz, and the designed loss function L_{θ} is also required to satisfy k-Lipschitz, then it can be shown that $p_g(x)$ is converge to $p_{data}(x)$. The first Lipschitz constraint refers to the Lipschitz density, which requires that the true probability distribution density cannot change too fast and the change in density with the sample cannot be infinitely large, and this condition that can be satisfied for

most distributions and is a natural restriction. For example, if a slight adjustment is made to the pixels of an image, it should still be the true image, and the density in the true image should not change abruptly and dramatically under the Lipschitz assumption. The second Lipschitz constraint is for the loss function $L_\theta(x)$ which restricts the loss function $L_\theta(x)$ ability to model infinitely, by controlling it in the space of functions that satisfy the Lipschitz constraint, which is to prove the assumptions of convergence of the distribution and increased generalization ability. Additionally, LS-GAN effectively addresses the issue of vanishing gradients and would not elaborate on this further here.

Now make a comparison between LS-GAN and WGAN, in WGAN, the discriminator (critic) objective function is

$$\max_{\theta} \mathbb{E}_{x\sim p_{data}}[f_\theta(x)] - \mathbb{E}_{z\sim p_z}[f_\theta(G(z))] \tag{2.71}$$

where $\|f_\theta(x)\|_L \leq 1$. 1-Lipschitz arises as a result of solving the dual problem. Now think of 1-Lipschitz in a different way, where the discriminator tries to maximize the difference of the first-order moments between training sample and the generated sample $f_\theta(x)$. If the 1-Lipschitz constraint is not imposed, it will keep making the generated sample's $f_\theta(x)$ values to smaller values while the discriminator's objective function is unbounded; in LS-GAN, it processes pairs of samples. Due to $(a)_+$ function in the objective, when $L_\theta(G(z))$ is greater than $L_\theta(x) + \Delta(x, G(z))$, this term in the objective function is 0, and it is no longer optimized $L_\theta(G(z))$. Therefore, LS-GAN does not also make the value of $L_\theta(G(z))$ to be too large, resulting in unbounded values of the objective function.

WGAN and LS-GAN add 1-Lipschitz constraints from different perspectives, with the former relying more on "technology" and the latter more on intuition, both suggesting that some degree of constraint on the discriminator is necessary.

2.7 GAN-GP

2.7.1 Weights Clipping

To solve the 1-Lipschitz problem of the discriminator, we illustrate both weight clipping and gradient penalty in this section.

Weight clipping is the most straightforward way, i.e., the weights of the discriminators are restricted to a certain range $[-c, c]$. When training the discriminator, in each iteration, the gradient of the weights is calculated based on the batch samples and updated to get new weights. The weights that are outside the range are finally clipped to c or $-c$.

The weight clipping method is simple and fast to compute, but it also has some issues. First, it is a crude approximation and cannot strictly guarantee the 1-Lipschitz constraint. Then, if the threshold c is chosen too large, it takes a long time for discriminator to converge and reach the optimum, and it is easy to cause gradient explosion. While the threshold c is selected too small, it is easy to produce the problem of gradient vanishing. Finally, experimental observations have shown that weight clipping can cause the discriminator's function to become overly simple, ignoring the higher order moments of the data distribution. In other words, the discriminator only focuses on the mean and variance of the data distribution, while neglecting skewness and kurtosis. This issue arises regardless of whether weight clipping, L2 parameter clipping, or L1 and L2 weight decay are used.

2.7.2 WGAN-GP

A derivable function satisfies the 1-Lipschitz constraint when and only the norm of the gradient of the function at any point is less than or equal to 1. Furthermore, when the discriminator of WGAN reaches optimality, the norm of the gradient of $f(x)$ is almost always 1 on p_g and p_{data}. Considering these two points, WGAN-GP [8] adds a regularization term to the discriminator's objective function so that the norm of the gradient at any point is close to 1, i.e.,

$$\max_{f_w} \mathbb{E}_{x\sim p_{data}}[f_w(x)] - \mathbb{E}_{z\sim p_z}[f_w(G(z))] - \lambda \mathbb{E}_{x\sim p_x}\left[(\| \nabla_x f_w(x) \|_2 - 1)^2\right] \tag{2.72}$$

Where λ is the penalty factor greater than 0. It should be noted that the gradient penalty term only imposes a "soft" constraint, which does not strictly require $\|\nabla_x f_w(x)\|_2$ equal to 1 everywhere, and is allowed to fluctuate slightly up and down, so it does not strictly satisfy the 1-Lipschitz constraint.

In addition, in practical training, we also need to consider how to obtain samples for the penalty term. Since it is impossible to traverse all samples in the full space to make the gradient criterion close to 1, we usually use linear interpolation to construct the samples for the penalty term. For example, for a sample x_{data} from p_{data}, a sample x_g from p_g and a random number $\epsilon \sim U[0, 1]$, we get a sample of the penalty term x_{gp} as shown in Fig. 2.20.

$$x_{gp} = \epsilon x_{data} + (1 - \epsilon) x_g \tag{2.73}$$

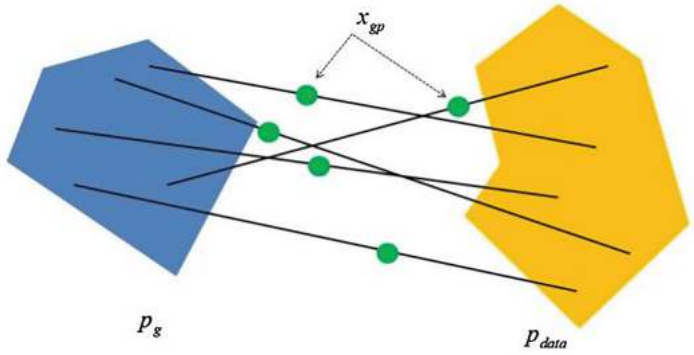


Fig. 2.20 Schematic diagram of the penalty sample

2.8 IPM

We know that the original form GAN is only a special case of fGAN. fGAN proposes to use f-divergence as a metric of the distance between two probability distributions, and the JS divergence used in the original form GAN is one of the f-divergence. LSGAN, EBGAN, etc. can also be considered as special cases of fGAN. In fact, a similar situation exists in WGAN. IPM defines a large class of metrics to define the distance between two probability distributions, and the Wasserstein distance defined in WGAN can be considered as a special case in the framework of IPM. Understanding IPM helps us to have a more comprehensive understanding of GAN.

2.8.1 IPM Definition

Similar to the f-divergence, the IPM (Integrated Probability Metric) also measures the distance between two probability distributions. For the implicit probability distribution defined by the generator p_g and the probability distribution of the training dataset p_{data}, the IPM is defined as

$$d_F(p_{data}, p_g) = \sup_{f \in F} \mathbb{E}_{x \sim p_{data}}[f(x)] - \mathbb{E}_{x \sim p_g}[f(x)] \tag{2.74}$$

where F is a set of bounded, measurable, numerical functions, and f is a function in the set of F. The optimal f^* in these f make the entire equation achieves a maximum value, which is some distance of the p_{data} and p_g distribution. By choosing different set F, we get different form of the metric.

It should also be noted that the IPM discussed in this section needs to satisfy an additional condition: for any function f in the set F, the $-f$ should also be in the set F, which means F is symmetric. At this point, the metric under the IPM framework $d_F(p_{data}, p_g)$ satisfies positivity, symmetry and trigonometric inequality, but is a pseudo-metric, i.e., $d_F(p_{data}, p_g) = 0$ does not imply $p_{data} = p_g$.

In GAN, distance is generally obtained by optimizing the discriminator, and the discriminator is usually the last fully connected layer. Its input is a vector, and its output value is a scalar value. The calculation process of this layer can be regarded as calculating the inner product of two vectors (one vector is the input of the fully connected layer, and the other vector is the weight of the fully connected layer). Therefore, in IPM, we usually consider functions in the form of $f(x) = < v, \Phi_w(x) \mid v \in \mathbb{R}^m >$, where v is an m-dimensional

vector and $\Phi_w(x)$ maps sample x to an m-dimensional vector. Φ_w can be understood as a neural network, and IPM can be written as:

$$d_F(p_{data}, p_g) = \sup_{f \in F} \mathbb{E}_{\boldsymbol{x} \sim p_{data}}[f(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{x} \sim p_g}[f(\boldsymbol{x})] = \max_{w, \|v\|_p \leq 1} < v, \mathbb{E}_{\boldsymbol{x} \sim p_{data}}[\boldsymbol{\Phi}_w(x)] - \mathbb{E}_{\boldsymbol{x} \sim p_g}[\boldsymbol{\Phi}_w(x)] > = \max_{w} \left[\|$$

Among them, there are

$$\mu_w(p_g) = \mathbb{E}_{\boldsymbol{x} \sim p_g}[\boldsymbol{\Phi}_w(x)] \tag{2.76}$$

$$\mu_w(p_{data}) = \mathbb{E}_{\boldsymbol{x} \sim p_{data}}[\boldsymbol{\Phi}_w(x)] \tag{2.77}$$

It can be seen that IPM maximizes the difference between the mean values of two probability distribution features. When calculating the difference, the neural network Φ_w first maps the sample x to a certain feature and then calculates the mean of the feature under the probability distribution. The difference is finally obtained by the q-norm of the subtract between the feature mean vectors, and the maximum difference value is obtained by finding the optimal neural network Φ_w. When the IPM is calculated, the optimization generator just needs to optimize p_g to minimize the IPM as before.

2.8.2 GAN-Based IPM

In fact, many GAN can be derived from or have close connection with IPM, and we list a few representatives here without derivation. In IPM, when the set of functions F is restricted to the set of all functions satisfying 1-Lipschitz, there are

$$d_F(p_{data}, p_g) = \sup_{|f|_L \leq 1} \mathbb{E}_{x \sim p_{data}}[f(x)] - \mathbb{E}_{x \sim p_g}[f(x)] \tag{2.78}$$

At this point, the objective function of the discriminator of WGAN is obtained. In the original WGAN, a weight clipping strategy is used to adjust the weights of the neural network to fall within the range of [−1, 1]. It can be seen that it actually applies ∞ norm constraints (p = 1) to the weights w of ϕ and the vectors v in the final fully connected layer, where q = 1. Therefore, its essence is to minimize the difference in feature mean based on L1 norm.

McGAN [9] extends the concept of feature mean differences, considering not only the mean but also second-order statistical feature variance. The selection function set F is

$$\left\{ f(x) = < U^T \boldsymbol{\Phi}_w(x), V^T \boldsymbol{\Phi}_w(x) > \middle| U, V \in \mathbb{R}^{m \times k}, U^T U = I_k, V^T V = I_k \right\} \tag{2.79}$$

Both $\{u_1, u_2, ..., u_k\} \in \mathbb{R}^m$ and $\{v_1, v_2, ..., v_k\} \in \mathbb{R}^m$ are orthogonal bases, and the IPM distance exported under this selection is:

$$d_F(p_{data}, p_g) = \max_{w} \| \left[\Sigma_w(p_{data}) - \Sigma_w(p_g) \right]_k \|_* \tag{2.80}$$

Among them, $[A]_k$ represents the rank k approximation of matrix A, $\|x\|_*$ is the KyFan norm of the vector, and it can be seen that the IPM distance corresponding to McGAN is obtained by maximizing the variance difference between the embedded features of the two distributions.

In the Maximum Mean Discrepancy (MMD), the set of functions F in IPM is selected as $\{\{f| \parallel f \parallel_{H_k \leq 1}\}$, which restricts it to the unit sphere of Hilbert space H_k. Because for any $f(x)$ in Hilbert space, $f(x) = \sum_{i=1}^{n} a_i \boldsymbol{\Phi}_{x_i}(x)$, where $\boldsymbol{\Phi}_{x_i}(x)$ is determined by a defined kernel function, i.e., $k(x, x_i) = \boldsymbol{\Phi}_{x_i}(x)$, and because $f(x) = < f, k(\cdot, x) >_{H_k}$, the IPM distance can be written as:

$$d_F(p_{data}, p_g) = \sup_{\|f\|_{H_k \leq 1}} \mathbb{E}_{\boldsymbol{x} \sim p_{data}}[f(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{x} \sim p_g}[f(\boldsymbol{x})] = \sup_{\|f\|_{H_k \leq 1}} \left\{ < f, \mathbb{E}_{\boldsymbol{x} \sim p_{data}} \boldsymbol{\Phi}_x(\cdot) - \mathbb{E}_{\boldsymbol{x} \sim p_g} \boldsymbol{\Phi}_x(\cdot) >_{H_k} \right\} = \| \mu$$

Among them, $\mu(p) = \mathbb{E}_{x\sim p}\Phi_x(\cdot)$. MMD first maps the sample x to an infinite dimensional Hilbert space through $\Phi_x(\cdot)$, and then calculates the difference between two probability distributions with respect to the first-order moment (mean), which is the MMD distance. When two distributions are completely equal, their MMD distance is 0. The GMMN generative moment matching network directly trains the generative model by minimizing the MMD distance, with a Gaussian kernel $k(x, x') = \exp(-\| x - x'\|^2)$ selected as the kernel function. MMDGAN replaces the fixed Gaussian kernel function in GMMN with a combination of injective function f_w and Gaussian kernel function, that is, $k(x, x') = \exp(-\| f_w(x) - f_w(x')\|^2)$. Then, a discriminator is used to learn a kernel function to obtain the MMD distance, and the generator is optimized using the MMD distance. GMMN and MMDGAN are both generative models within the IPM framework, with the main difference being that the kernel function is fixed or contains learnable parameters. Their advantage lies in the ability to map samples to different feature spaces by selecting different kernel functions, but their disadvantage is that when there are a large number of samples, calculating the MMD distance requires significant computational resources.

In linear discriminant analysis, we want to project the samples onto a straight line so that the projection points of similar samples are as close as possible and the projection points of different samples are as far away as possible. FisherGAN [10] is inspired by this. In FisherGAN, the set of functions F is restricted to:

$$\frac{1}{2}\left[\mathbb{E}_{x\sim p_{data}} f^2(x) + \mathbb{E}_{x\sim p_g} f^2(x)\right] = 1 \tag{2.82}$$

There are also MCGAN, MMDGAN, etc. [9, 11] and are considered as a special case under the IPM framework and will not be discussed here.

Furthermore, when the set of functions F is limited to all continuous functions between -1 and 1, there is

$$d_F(p_{data}, p_g) = \delta(p_{data}, p_g) \tag{2.83}$$

At this point, the form of IPM is the total variational distance, and the metrics learned by EBGANs for p_g and p_{data} are exactly the total variational distance. The discriminator is used as $f(x)$ to maximize the above equation. When the discriminator reaches its optimum, the objective function of the generator will be very close to the total variation distance of p_g and p_{data}, indicating a very close relationship between IPM and EBGANs.

2.8.3 IPM and f-Divergence

The metric defined by f-divergence usually faces several problems. First, as the dimensionality of the data space increases, the value of f-divergence will become increasingly difficult to estimate, and the support sets of the two distributions will tend to be unaligned, which will lead to infinity in the value of f-divergence. For example, using KL divergence to calculate the distance between two distributions p_g and p_{data}, if there is $p_g(x_0) = 0$ and $p_{data}(x_0) \neq 0$ at a certain point x_0, according to the KL divergence calculation formulation:

$$KL(p_{data} \parallel p_g) = \int p(x) \log \frac{p_{data}(x)}{p_g(x)} dx \tag{2.84}$$

At least at the x_0 point, infinity occurs within the log logarithm. Also, considering that

$$\mathrm{D}_f(p_{data} \parallel p_g) \geq \sup_T \left\{\mathbb{E}_{x\sim p_{data}}[T(x)] - \mathbb{E}_{x\sim p_g}[f^*(T(x))]\right\} \tag{2.85}$$

In other words, the metric learned by the discriminator in the GAN is not the true f-divergence, but only a variational lower bound of it. In practice, it is difficult to guarantee whether the equality in the above equation holds, leading to inaccurate metric estimates.

IPM overcomes the problem of f-divergence. The convergence of f-divergence is highly dependent on the distribution of the data, while the convergence of IPM is not affected by the dimensionality of the sample data or the choice of samples. It can converge to the probability distribution of the training dataset p_{data} that exhibits stronger consistency.

2.9 Other Objective Functions

In addition to the many objective functions mentioned earlier, there are many other types and variations of GAN's objective functions. Their construction is not necessarily based on some distance function such as f-divergence or IPM. For example, some objective functions are designed to enable the model to distinguish between true and false and improve the quality of generated samples (RGAN); some are used to calculate the reconstruction loss function (BEGAN, MAGAN, etc.); some are designed for classification tasks that exist in GANs (TripleGAN, cGAN, etc.). Due to space constraints, we will introduce two types of objective functions using RGAN and BEGAN.

2.9.1 RGAN

The standard GAN uses JS divergence to measure the distance between p_{data} and p_g, and from the perspective of JS divergence

$$D_{JS}(p_{data} \parallel p_g) = \frac{1}{2}\{\log 4 + \max \mathbb{E}_{\boldsymbol{x}\sim p_{data}}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x}\sim p_g}[\log (1 - D(\boldsymbol{x}))]\} \tag{2.86}$$

The training of GAN is a process of minimizing JS divergence. When $D(x_{data}) = 1$ and $D(x_g) = 0$, the JS divergence value is maximized. When $D(x_{data}) = 0.5$ and $D(x_g) = 0.5$, the JS divergence value is minimized. In summary, training GANs should generally involve a process where $D(x_{data})$ decreases from 1 to 0.5, while $D(x_g)$ increases from 0 to 0.5. Consider an unsaturated generator objective function that maximizes the value of $D(x_g)$, and it can be imagined that if the training level is good enough, the value of $D(x_{data})$ will always remain at 1, while the value of $D(x_g)$ will continue to grow and even reach 1. However, there is no process of simultaneously reducing the value of $D(x_{data})$ here, which is inconsistent with the objective function optimization process of standard GAN.

In response, RGAN [12] redefined the discriminator, which takes a pair of samples x_{data} and x_g as inputs $\widehat{x}$ each time, i.e., $\widehat{x} = (x_{data}, x_g)$, and its output is $D(\widehat{x}) = sigmoid(C(x_{data}) - C(x_g))$. The discriminator no longer estimates the probability that sample x comes from the training dataset, but evaluates the probability that x_{data} is more realistic than x_g. The objective function of the discriminator is:

$$\min_{\boldsymbol{\theta}} -\mathbb{E}_{\boldsymbol{x}\sim p_{data}, \boldsymbol{z}\sim p_z}[\log (sigmoid(C(\boldsymbol{x}) - C(G(\boldsymbol{z}))))] \tag{2.87}$$

The objective function of the generator is:

$$\min_{\boldsymbol{\theta}} -\mathbb{E}_{\boldsymbol{x}\sim p_{data}, \boldsymbol{z}\sim p_z}[\log (sigmoid(C(G(\boldsymbol{z})) - C(\boldsymbol{x})))] \tag{2.88}$$

RGAN uses a relative discriminator to make it more stable compared to standard GAN training, resulting in improved generation quality.

2.9.2 BEGAN

Reconstruction loss functions often appear in GANs, which make the output of the neural network as close as possible to the input results. The reconstruction loss function may appear in the generator, such as CycleGAN and VAEGAN, or in the discriminator, such as EBGAN and MAGAN. The BEGAN introduced in this section is also set in the discriminator to reconstruct the loss function.

In BEGAN, the structure of discriminator D is an autoencoder, which takes sample x as input and outputs the reconstructed $D(x)$ of the sample. Therefore, the autoencoder loss $\mathscr{L}(x)$ of the sample can be defined as:

$$\mathscr{L}(x) = \parallel x - D(x) \parallel \tag{2.89}$$

The general design idea of GAN is to make the probability distributions of p_{data} and p_g as close as possible, while the design idea of BEGAN no longer considers the distribution of samples, but makes the probability distributions of the two autoencoder losses as close as possible. Specifically, for samples from the training

set distribution $p_{data}(x)$, the corresponding autoencoder loss $\mathscr{L}(x)$ will also correspond to a certain probability distribution $\mu_1(x)$; correspondingly, for samples generated by the generator with the distribution $p_g(x)$, the autoencoder loss also corresponds to the probability distribution $\mu_2(x)$. So, BEGAN expects to achieve the effect of $\mu_2(x)$ approaching $\mu_1(x)$ by optimizing the generator.

BEGAN uses Wasserstein distance to measure the difference $W(\mu_1, \mu_2)$ between two probability distributions μ_1 and μ_2, i.e.,

$$W(\mu_1, \mu_2) = \inf_{\gamma \in \Gamma(\mu_1, \mu_2)} \mathbb{E}_{(x_1, x_2)}[|x_1 - x_2|] \tag{2.90}$$

Among them, γ is a certain joint probability distribution of μ_1 and μ_2, and γ is the set of all possible joint probability distributions. BEGAN does not perform dual transformation on it like WGAN to obtain a form that can be estimated through sampling, but attempts to obtain its lower bound. According to Johnson's inequality, there is

$$\inf_{\gamma \in \Gamma(\mu_1, \mu_2)} \mathbb{E}_{(x_1, x_2)}[|x_1 - x_2|] \geq \inf_{\gamma \in \Gamma(\mu_1, \mu_2)} |\ \mathbb{E}_{(x_1, x_2)}[x_1 - x_2]\ | = |\ m_1 - m_2\ | \tag{2.91}$$

Among them, m_1 and m_2 are the mean values of μ_1 and μ_2, respectively. In order to approximate $W(\mu_1, \mu_2)$, it is necessary to obtain the maximum value of $|m_1 - m_2|$. Furthermore, for p_{data} and p_g, we can only change the distribution of μ_1 and μ_2 by optimizing the autoencoder to obtain the maximum difference in mean. Therefore, the optimization objective of the discriminator can be set as follows:

$$\min_{\theta_D} \mathbb{E}_{x \sim p_{data}}[\|\ x - D(x)\ \|] - \mathbb{E}_{x \sim p_g}[\|\ x - D(x)\ \|] \tag{2.92}$$

The objective function of the discriminator has formal similarities with WGAN, but its consideration is not the difference in sample distribution but the difference in reconstruction error distribution. In addition, BEGAN calculates the lower bound of Wasserstein distance, thus avoiding the Lipschitz constraint imposed on the discriminator.

After obtaining the Wasserstein approximation distance (lower bound) of the autoencoder loss distribution, it is natural to optimize the generator by optimizing this distance, i.e., the objective function is:

$$\min_{\theta_G} \mathbb{E}_{z \sim p_z}[\|\ x - D(G(z))\ \|] \tag{2.93}$$

On this basis, BEGAN considers the balance problem between the discriminator and generator during training. $\mathbb{E}[\mathscr{L}(x)] = \mathbb{E}[\mathscr{L}(G(z))]$ is defined as an equilibrium point by BEGAN; there exists a parameter of $p_{data}(x) = p_g(x)$, which means that the samples generated by the generator make it impossible for the discriminator to distinguish between true and false. But in practice, it is usually necessary to add a relaxation factor α to adjust the equilibrium point, that is, $\alpha\mathbb{E}[\mathscr{L}(\boldsymbol{x})] = \mathbb{E}[\mathscr{L}(G(\boldsymbol{z}))]$, where $\alpha \in [0, 1]$. In order to maintain balance, BEGAN borrowed relevant algorithms from control theory, and the loss function of its final discriminator is:

$$\min_{\theta_D} \mathbb{E}_{x \sim p_{data}}[\|\ x - D(x)\ \|] - k_t \mathbb{E}_{x \sim p_g}\Big[\|\ x - D(x)\ \| \tag{2.94}$$

Among them, $k_{t+1} = k_t + \lambda_k(\alpha\mathbb{E}[\mathscr{L}(x)] - \mathbb{E}[\mathscr{L}(G(z))])$. k_t is a continuously updated parameter, and λ_k is a hyperparameter. The objective function constructed in this way forms a negative feedback system, allowing the two values of the loss function to remain balanced.

References

1. Goodfellow, Ian, et al. Generative adversarial nets. Advances in neural information processing systems 27 (2014).

2. Mao, Xudong, et al. Least squares generative adversarial networks. Proceedings of the IEEE international conference on computer vision. 2017.
3.

LeCun, Yann, et al. A tutorial on energy-based learning. Predicting structured data 1.0 (2006).

4. Zhao, Junbo, Michael Mathieu, and Yann LeCun. Energy-based generative adversarial network. arXiv preprint arXiv:1609.03126 (2016).
5. Nowozin, Sebastian, Botond Cseke, and Ryota Tomioka. f-gan: Training generative neural samplers using variational divergence minimization. Proceedings of the 30th International Conference on Neural Information Processing Systems. 2016.
6. Arjovsky, Martin, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. International conference on machine learning. pmlr, 2017.
7. Qi, Guo-Jun. Loss-sensitive generative adversarial networks on lipschitz densities. International Journal of Computer Vision 128.5 (2020). 1118-1140.
[MathSciNet][Crossref]
8. Gulrajani, Ishaan, et al. Improved training of wasserstein GAN. arXiv preprint arXiv:1704.00028 (2017).
9. Mroueh, Youssef, Tom Sercu, and Vaibhava Goel. Mcgan: Mean and covariance feature matching gan. international conference on machine learning. PMLR, 2017.
10. Mroueh, Youssef, and Tom Sercu. Fisher gan. arXiv preprint arXiv:1705.09675 (2017).
11. Li, Chun-Liang, et al. Mmd gan: Towards deeper understanding of moment matching network. arXiv preprint arXiv:1705.08584 (2017).
12. Jolicoeur-Martineau, Alexia. The relativistic discriminator: a key element missing from standard GAN. arXiv preprint arXiv:1807.00734 (2018).

P. Long, X. Guo, *Generative Adversarial Network*
https://doi.org/10.1007/978-981-96-9404-4_3

3. Training of GAN

Peng Long[1] ✉ and Xiaozhou Guo[2]
(1) Beijing YouSan Educational Technology, Beijing, China
(2) • China Electronics Technology Group Corporation No. 54 Research Institute, Shijiazhuang, China

Abstract
Although GAN has been widely applied in various aspects, training a GAN is not an easy task. During the training process, problems such as mode collapse, non-convergence of the loss, and blurriness of generated samples may occur. This chapter introduces the three most common problems in GAN training, namely gradient vanishing, non-convergence of the objective function, and mode collapse, and analyzes the causes of these problems. Regarding the problem of gradient vanishing, the annealing noise method is introduced. In response to the oscillation and instability of the objective function during GAN training, two methods, spectral regularization SNGAN and consistent optimization, are elaborated in detail. In addition, many GAN training techniques, such as feature matching and historical mean, are also explained. For the problem of mode collapse, from the two perspectives of the objective function and the GAN structure, some algorithms that can effectively alleviate mode collapse are introduced, and specific methods such as unrolledGAN, DRAGAN, MADGAN, VVEGAN, and the minibatch discriminator are provided.

Keywords Mode collapse – Spectral regularization – GAN training techniques

GAN has attracted tremendous attention once it was proposed and has been widely used in various aspects due to its good theoretical support and relatively excellent generation effect. However, training GAN is not an easy task, and problems such as mode collapse, object function non-convergence, and blurred generation samples may occur during the training process, and in this chapter we will introduce some GAN training techniques to solve these problems.

In Sect. 3.1, we introduce the three most common problems in GAN training, including gradient vanishing, non-convergence, and mode collapse; Sect. 3.2 introduces the annealing noise method for the standard form of GAN; Sect. 3.3 introduces the spectral normalization method with very wide influence for the WGAN series, detailing the basic principle and process. In response to the oscillation and instability of the objective function during GAN training, we have explained in detail the spectral normalization SNGAN and consistent optimization methods in Sects. 3.3 and 3.4. In addition, we have presented many training techniques for GANs in Sect. 3.5, such as feature matching and historical mean. For the problem of mode collapse, Sect. 3.6 introduces some algorithms that can effectively alleviate it from the perspectives of objective function and GAN structure design and specifically provides five methods: unrolled GAN, DRAGAN, MADGAN, VVEGAN, and mini-batch discriminator.

- Section 3.1 Several issues of training
- Section 3.2 Annealing noise
- Section 3.3 Spectral normalization
- Section 3.4 Consistent optimization
- Section 3.5 GAN training techniques
- Section 3.6 Mode collapse

3.1 Several Issues of Training

In this section, we present some of the problems that often occur in GAN during training.

3.1.1 Gradient Vanishing Problem

First, regarding the gradient vanishing problem of GAN, we have partially explored it in LSGAN and WGAN in Chap. 2, where LSGAN argues that the gradient conducted by the discriminator to the generator vanishes when the generator is trained with classified correct samples but far from the decision surface; in WGAN, we are able to illustrate how the gradient vanishes. When p_g and p_{data} are both low-dimensional manifolds in a high-dimensional space, the distance or metric scatter between the two distributions is discontinuous, and it becomes constant or infinite at $p_{\text{data}} \neq p_g$, and only at $p_{\text{data}} = p_g$ the distance value is zero, thus giving rise to the problem of gradient vanishing.

Now, the gradient vanishing problem of the original form GAN is re-illustrated in detail. For the probability distribution of training dataset p_{data}, both theory and practice show that the support set of p_{data} is a low-dimensional manifold in a high-dimensional space; for the probability distribution p_g which is implicitly defined by the generator G, since GAN first samples from a simple distribution $z \sim p(z)$, and then uses a generator to obtain the sample $x = G(z)$. If the dimension of z is smaller than the dimension of x, then the support set of p_g is also a low-dimensional manifold, that is, the dimension will not exceed the dimension of z. Let's give an example to illustrate what a low-dimensional manifold in high-dimensional space is. For example, for a circle on a two-dimensional plane with its center at the origin and radius r, although this manifold exists in two-dimensional space, its dimension is only 1, which means that only the angle parameter is needed to describe any point on the circle. Therefore, it can be considered a low-dimensional manifold.

If the support sets of p_g and p_{data} do not intersect or are both low-dimensional manifolds, there exists a perfect discriminator D^* that can completely separate the two manifolds. The perfect discriminator D^* outputs 1 for any sample on the support set of p_{data} and 0 for any sample on the support set of p_g. As shown on the left in Fig. 3.1, when the support sets of p_g and p_{data} do not intersect, continuous training of D will inevitably result in a perfect discriminator. For low-dimensional manifolds, ignoring the intersection of the support sets will also result in a perfect discriminator. Obviously, when the support sets of p_g and p_{data} are not low-dimensional manifolds and intersect, there cannot be a perfect discriminator, as shown in Fig. 3.2. For the samples in the middle dashed line, the discriminator cannot simply provide output values of 0 or 1. Unfortunately, this situation is almost impossible to occur during the actual training of GANs. Considering that in alternating iterations of training GANs, the generator is usually trained only once, while the discriminator is trained multiple times in the hope of achieving the current optimal discriminator, it is likely that a perfect discriminator D^* will appear in practice.

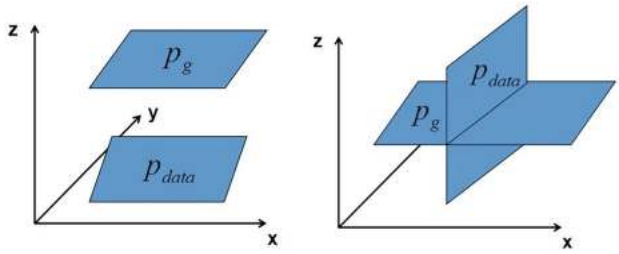


Fig. 3.1 Support set of p_g and p_{data} are (not) intersected

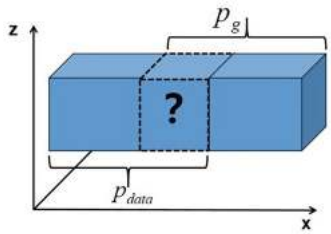


Fig. 3.2 Support set of p_g and p_{data} is not a low-dimensional manifold

The perfect discriminator D^* is a constant of 0 or 1 on both the support sets of p_g and p_{data}, and its gradient $\nabla_x D$ is clearly also 0. When using backpropagation algorithm to train generator G, the discriminator

cannot pass any gradient information to the generator. The generator cannot obtain gradient information, its parameters stop updating, and its object function shows convergence, but it is far from sufficient to illustrate $p_g \rightarrow p_{\text{data}}$. Some experiments have shown that as the discriminator becomes better during training, the gradient of the generator disappears to zero. This creates a contradiction: if the discriminator is trained well, the gradient disappears when training the generator; if the discriminator is not trained well, it will not be able to learn accurate JS divergence, which will not guide the training of the generator in the future. Therefore, when training GANs, it is important to be careful not to train the discriminator too well.

In order to avoid gradient vanishing, while the original form of GAN was proposed, the authors incidentally proposed another objective function for the generator:

$$\min \mathbb{E}_{p_z}[-\log(D(G(z)))] \tag{3.1}$$

It can alleviate the problem of gradient vanishing to some extent, acting mainly in the early stages of training. However, in each round of alternating iterations of training, when the discriminator reaches the current optimum, the learned distance of p_g and p_{data} are

$$KL(p_g \parallel p_{\text{data}}) - 2JS(p_g \parallel p_{\text{data}}) \tag{3.2}$$

Such a distance can confuse the training of the generator, on the one hand, to minimize the KL divergence between two distributions, and at the same time, to maximize the JS divergence between two distributions, resulting in a difficult for the system converge to an equilibrium state. And experimental results show that the training process is unstable using such an objective function.

3.1.2 Non-convergence

In GAN, we represent the generator *G* and the discriminator *D* as a fully connected neural network or a convolutional neural network, and then learn the weight parameters of the network by gradient descent algorithm and backpropagation. In Chap. 1, we have shown that GAN is an implicit generative model that p_g is implicitly defined by the samples generated by the generator, so the whole process is not accessible to p_g directly. However, in the proof of global convergence, the proof is centered around the probability density function p_g and requires that object function is convex; however, this property is not guaranteed when using a neural network representation of *G* and *D* since the optimization is performed in the parameter space rather than the function space.

In addition, the proof of convergence requires that both *G* and *D* have sufficient capacity. The capacity of a model can be simply understood as the number of parameters of the model, which describes the ability of the model to fit various functions. A model with high capacity may cause overfitting, while a model with low capacity may cause underfitting and may lead to non-existence of equilibrium points when the capacity of the discriminator *D* is limited.

The convergence proof also has the condition that in each step of the GAN iterative training, fixing parameter of generator *G* and trains the discriminator *D* to optimality. However, in practice, training the discriminators to optimality requires a huge amount of computation, so only one or more times training are executed at a time, and the generators are trained before the discriminators reach optimality. This leads to the confusion of whether the alternating iterations training algorithm is solving the $\min_G \max_D V(G,D)$ problem or the $\max_D \min_G V(G,D)$ problem? Usually,

$$\min_G \max_D V(G,D) \neq \max_D \min_G V(G,D) \tag{3.3}$$

For example, $f(x,y) = \sin(x+y)$, obviously easy to know $\min_y \max_x f(x,y) = 1$ and $\max_x \min_y f(x,y) = -1$.

Also, the generator and the discriminator optimize the same objective function, but in opposite directions, which can easily cause the "rotation" phenomenon. For a simple example $f(x,y) = xy$, the objective function is:

$$\min_x \max_y xy \tag{3.4}$$

Using the gradient descent algorithm with alternating iterations, the following results are obtained, which are found to be unable to converge to the Nash equilibrium point and the objective function keeps oscillating, as shown in Fig. 3.3. When the network structure and objective function are more complex, the situation is obviously not promising.

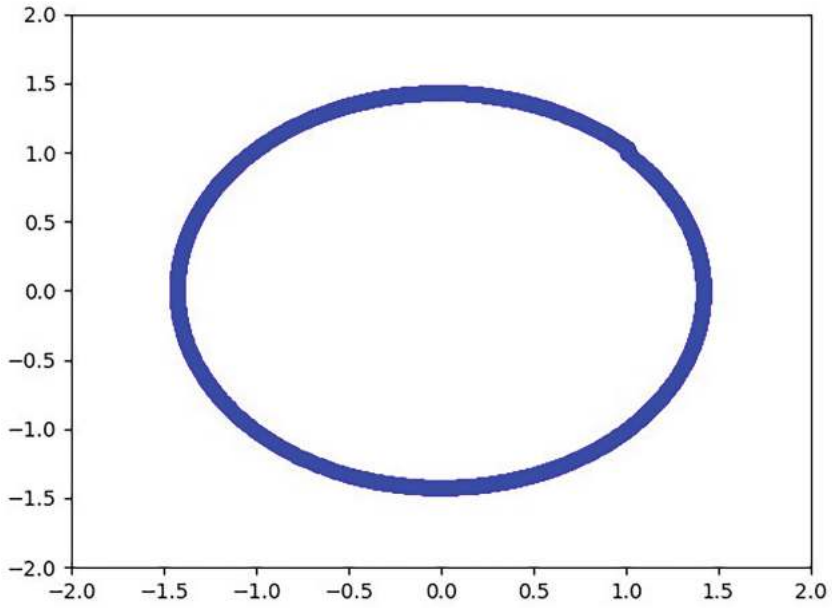


Fig. 3.3 Oscillation non-convergence of object function

3.1.3 Mode Collapse

GAN also has a much criticized mode collapse problem, which is the poor diversity of GAN generation samples. For example, there is an image set containing several kinds of fruits such as apples, oranges, lemons, and grapes. The GAN is trained with the fruit image set, and it is hoped that the generator can generate realistic images of apples, oranges, lemons, etc. When the training process is completed, forward inference reveals that although the pictures generated by the generator are very realistic, they can only generate pictures of apples and oranges, and almost no pictures of lemons, at which point the mode collapse problem occurs. Supposing the probability density function of training dataset p_{data} is as follows, there are four peaks, and each peak is called a mode. If the GAN reaches the optimal, there should be $p_g = p_{\text{data}}$, it is reasonable that generator have four modes, as shown in Fig. 3.4.

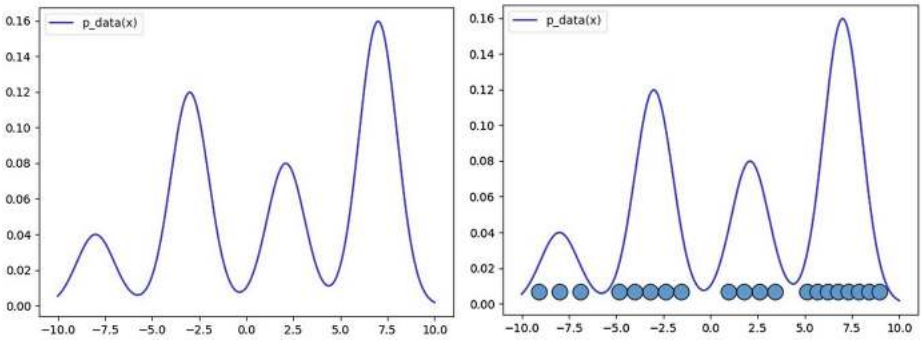


Fig. 3.4 Not occurrence mode collapse of generator

But the actual generator often cannot cover all modes, especially when the mode collapse occurs, the generated samples can only cover a few modes, as shown in Fig. 3.5. Why does the GAN collapse? Because the generator only needs to place the samples under a few modes to deceive the discriminator, and when the discriminator updates and no longer trusts the mode, the generator moves the samples to another mode, as

shown in Fig. 3.5 on the right. This process goes round and round, and the generator never needs to consider covering all modes, and training is just a futile and time-consuming exercise. Also, the mode collapse is related to the perfect discriminator because the perfect discriminator outputs probability 1 for all true samples and probability 0 for false samples, and the generator simply adjusts itself to generate samples close to any true sample with closest distance, without any incentive to cover all modes.

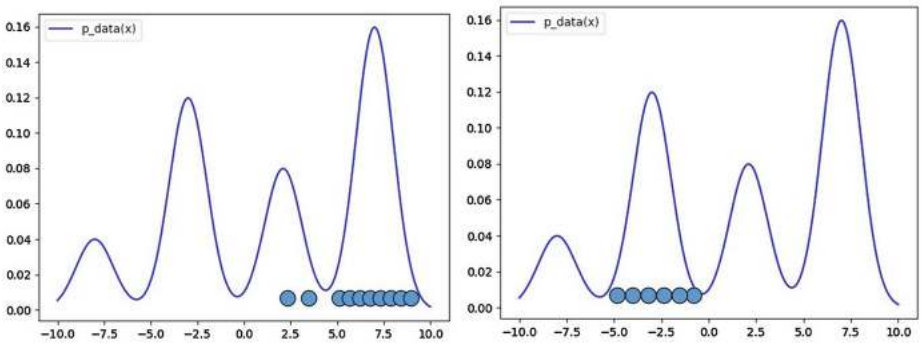


Fig. 3.5 Occurrence of mode collapse

The above points are the differences between the theoretical proofs and the practical ones. The existence of these differences makes GAN in practice not reach the global optimal solution, and it is accompanied by problems such as non-convergence of the object function, oscillation, mode collapse, etc. And there does not exist a technology that can completely solve these problems, but it can alleviate them to a certain extent, and we will introduce these technologies in turn next.

3.2 Annealing Noise

The following problem has been elaborated in the previous section: if the support set of two probability distributions p_g and p_{data} do not intersect or fall on a low-dimensional manifold, the discriminator is perfect when iterative training reaches the optimum thereby causing the gradient to vanish. To solve the problem, it is necessary to make p_g and p_{data} do not fall on a low-dimensional manifold and intersect. One way is to add noise to the input of the discriminator.

The idea is very simple but effective. Generally choose a Gaussian noise ϵ with a mean value 0. Then, Gaussian noise ϵ is added separately to p_g and p_{data} to obtain two new probability distributions $p_{g+\epsilon}$ and $p_{\text{data}+\epsilon}$. The support sets of the two new probability distributions must intersect and are no longer low-dimensional manifolds because the noise is continuous. The support sets of $p_{g+\epsilon}$ and $p_{\text{data}+\epsilon}$ would be dispersed throughout the entire space, and the two support sets are “interwoven” together, with dimensions of the entire space [1]. At this point, we no longer need to overly worry about the training degree of the discriminator. We can boldly train the discriminator to the optimal level because a perfect discriminator requires an output close to 1 for any sample from the training set and close to 0 for any sample from the generator. The optimal discriminator expression is

$$D(x) = \frac{p_{\text{data}+\epsilon}(x)}{p_{g+\epsilon}(x) + p_{\text{data}+\epsilon}(x)} \tag{3.5}$$

Obviously, the optimal discriminator cannot be a perfect discriminator at this point. Take Fig. 3.6 as an example, when no noise is added, the optimal discriminator, i.e., the perfect discriminator, can be very confident that the output of the discriminator is either 0 or 1 for any sample. But after adding noise, for some samples which are very close to the intersection or far from p_g and p_{data}, $D(x)$ it is generally not possible to give 0 or 1.

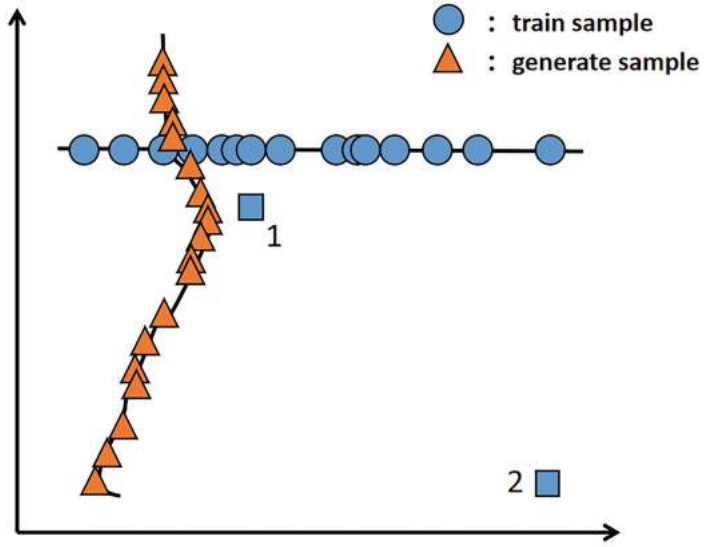


Fig. 3.6 Sample distribution when no noise is added

In GAN, it is necessary to compute the distance between p_g and p_{data} rather than $p_{g+\epsilon}$ and $p_{\text{data}+\epsilon}$, so setting the variance of Gaussian noise to a fixed value is not a wise choice. Referring to the simulated annealing algorithm, the variance of the Gaussian noise can be set to a relatively large value at the early stage of training in order to produce a non-negligible intersection part, and gradually reduce the noise variance until the support sets of p_g and p_{data} overlap, and the noise variance is reduced to 0. Some simple experiments show that at each iteration of the training, if the noise variance is large, the performance is the same as that of a normal GAN training without adding noise; if the variance is small, the parameters will be rotated at the equilibrium point and lack "centripetal force," so the parameters need to be adjusted appropriately to take advantage of the noise term.

Throughout the process, Gaussian noise ϵ "wraps" p_g and p_{data} together, causing p_{data} (the core of $p_{\text{data}+\epsilon}$) and p_g(the core of p_g) to constantly approach each other. When p_g and p_{data} overlap, the "wrapped" noise also disappears meanwhile.

3.3 Spectral Normalization

In WGAN, the discriminator is required to satisfy the 1-Lipschitz restriction that

$$\left| \frac{D(x_1) - D(x_2)}{x_1 - x_2} \right| \leq 1 \tag{3.6}$$

The norm of derivatives of $D(x)$ at any point must less than 1. It needs to be reminded again that this derivative refers to the derivative of $D(x)$ with respect to x, not to the discriminator weight w. The two solutions mentioned in Chap. 2, weight clipping and gradient penalty term, are "soft" methods. The weight clipping proposed in the original WGAN work wants to restrict the weights to a relatively small range, and the gradient penalty term in WGAN-GP directly wants the discriminant to have a gradient $\nabla_x D(x)$ close to 1. Clearly neither of these methods can theoretically guarantee the 1-Lipschitz limit. This section of SNGAN is an elegant, effective, and "hard" means to ensure that the 1-Lipschitz limit is satisfied [2, 3].

3.3.1 Eigenvalue and Singular Value

In order to understand the principle of SNGAN, we try to briefly introduce a part of knowledge about eigenvalues and singular values of matrices. Taking the square matrix as an example, first, how to understand $Ax = y$? From a common computational point of view, it is nothing more than a matrix A multiply a vector x to get a new vector y. But from a perspective of linear algebra, A actually represents a linear transformation that is applied to the vector x, transform x into a new vector y. However, for some special vectors x, we have:

$$Ax = y = \lambda x \tag{3.7}$$

Where λ is a constant. This means that the linear transformation A acting on the special vector x is obtained as λx. The effect of the transformation A at this point is a stretching transformation, which is simply a transformation of x stretched λ times. We call these special vectors as eigenvectors, and the eigenvectors corresponding to the stretching values λ are called eigenvalues. Generally, for n dimensional square matrix A (non-singular square matrix), there will be n eigenvalues and n eigenvectors.

Then, how is the transformation applied to matrix A? There are actually three steps involved in this: first, project vector x onto n eigenvectors, then stretch each projection vector of x by a factor of λ, and finally combine the stretched projection vectors into a new vector y. For example, matrix A is:

$$A = \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix} \tag{3.8}$$

Its corresponding two eigenvalues and eigenvectors are:

$$v_1 = \left[\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right]^T \quad \lambda_1 = 2 \tag{3.9}$$

$$v_2 = \left[\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}\right]^T \quad \lambda_1 = 4 \tag{3.10}$$

Linear transformation A effect acting on the two eigenvectors is as follows (Fig. 3.7):

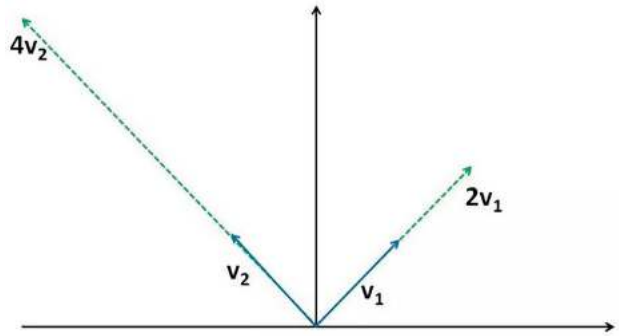


Fig. 3.7 Linear transformation effect

As an example, let $x = [0, 1]^T$, decompose it onto two eigenvectors v_1 and v_2 (Fig. 3.8), extend them by λ_1 and λ_2 times, respectively (Fig. 3.9), and combine the stretched vectors (Fig. 3.10).

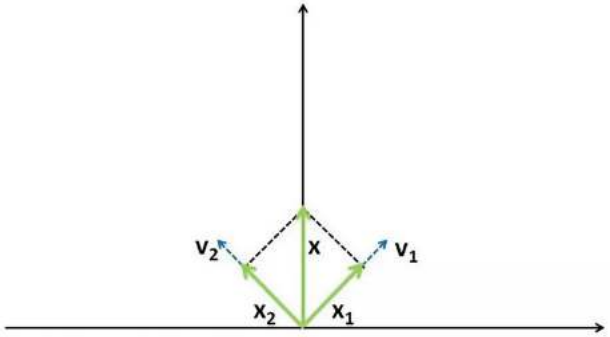


Fig. 3.8 Vector decomposition

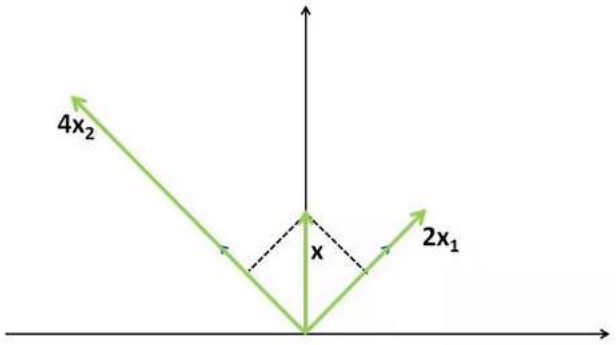


Fig. 3.9 Vector extension

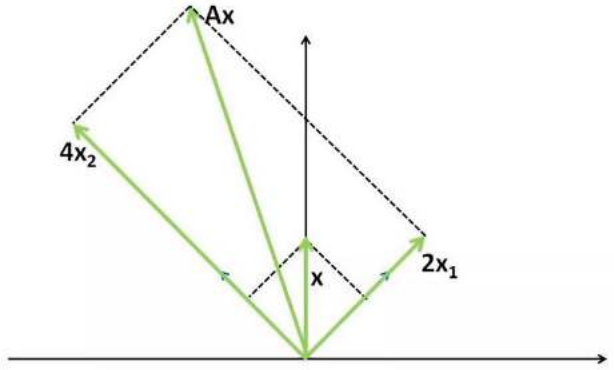


Fig. 3.10 Vector synthesis

Decompose vector x into the directions corresponding to n eigenvectors (essentially solving the representation of x on a set of basis composed of eigenvectors), perform scaling transformations on each basis (performing scaling transformations on the basis composed of eigenvectors), and finally perform vector synthesis (essentially solving the representation of the new vector obtained on the standard basis). This is actually describing the familiar matrix eigenvalue decomposition

$$A = U\Sigma U^{\mathrm{T}} \tag{3.11}$$

The eigenvalues are corresponding to the case of a square matrix, and the singular values are induced by generalizing them to general matrices. The singular value decomposition takes the form of

$$A = U\Sigma V^{T} \tag{3.12}$$

In essence, the eigenvalue decomposition is actually a subsumption of the effects of rotation and scaling in the linear transform, and the singular value decomposition is precisely a decomposition of the three effects of rotation, scaling, and projection of the linear transform (when dimension of V is greater than U exists projection).

3.3.2 Spectral Norm and the 1-Lipschitz Constraint

Then, what is the maximum value of Ax (measured by the 2-norm) for any unit vector x? Obviously, for the above problem, when x is equal to the eigenvector v_2, its value is the highest, because at this time, all x is "projected" onto the eigenvector with the highest scaling coefficient. Choosing other unit vectors will more or less decompose a part in the direction of v_1, with only twice the scaling in the direction of v_1. It is not as good as obtaining a larger value with four times the scaling in the direction of v_2. So, the maximum value of Ax should be the maximum eigenvalue of A. Generally, we define the spectral norm $\sigma(A)$ of matrix A as the maximum singular value of A. The spectral norm actually describes the maximum tensile strength of A and has

(3.13)

$$\frac{||Ax||}{||x||} = ||Ax||_{||x||=1} \leq \sigma(A)$$

It can be further associated with the fact that for a given arbitrary A, divide it by the spectral norm of $\sigma(A)$, and

$$\widehat{A} = \frac{A}{\sigma(A)} \tag{3.14}$$

must satisfy the 1-Lipschitz limit, it can be very easily shown that

$$\frac{\left\|\widehat{A}(x+\Delta x) - \widehat{A}(x)\right\|}{||\Delta x||} \leq \sigma\left(\widehat{A}\right) = \frac{1}{\sigma(A)}\sigma(A) = 1 \tag{3.15}$$

That is to say, any matrix A is highly likely to not satisfy the 1-Lipschitz constraint, but after performing spectral normalization on A (i.e., dividing it by the spectral norm), the maximum tensile strength of its linear transformation is 1, its spectral norm $\sigma\left(\widehat{A}\right)$ is 1, and the ratio of the output change to the input change cannot exceed 1, thus satisfying the 1-Lipschitz constraint.

SNGAN is based on this idea and performs spectral regularization on each layer weight W (i.e., dividing it by the spectral norm) to achieve the 1-Lipschitz constraint of D (x). We know that in each layer of a neural network, a linear operation is usually performed by multiplying the input x by the weight W to obtain the input y of the activation function, that is, $Wx = y$; then input its y into the activation function $f()$ to obtain the output z: $z = f(y)$. Due to the usual use of ReLU as the activation function, the ReLU activation function can be represented by a diagonal square matrix D, where the dimension of D is consistent with the length of y. For specific values of W and x, if the value of the ith dimension of vector y is greater than 0, then the i-th diagonal element of D is 1, indicating activation. Otherwise, it is 0, indicating that it is not activated at this time. At this point, the operation of a layer of neural network can already be represented by the multiplication of diagonal matrix D, weight matrix W, and input vector x, as shown below:

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{3.16}$$

It should be noted that the diagonal elements of D are either 0 or 1 and are related to W and x. Different W and x will result in different D values, but the maximum singular value of D must be 1, which means its spectral norm must be less than or equal to 1 (usually it cannot be 0). So for the discriminator D (x) composed of L-layer neural networks, the operation process can be extended as follows (Fig. 3.17):

$$D(x) = D^L W^L D^{L-1} W^{L-1} \cdots D^i W^i \cdots D^1 W^1 x \tag{3.17}$$

Where D^i denotes the i layer activation function matrix, and W^i denotes the weight matrix of the layer i. The weight matrix of each layer is spectrally normalized

$$\widehat{W}^i = \frac{W^i}{\sigma(W^i)} \tag{3.18}$$

The calculation process of the discriminator becomes:

$$D(x) = D^L \widehat{W}^L D^{L-1} \widehat{W}^{L-1} \cdots D^i \widehat{W}^i \cdots D^1 \widehat{W}^1 x \tag{3.19}$$

At this point, we examine spectral norm of the discriminator after spectral normalization

$$\sigma(A)\sigma(B) \geq \sigma(AB) \tag{3.20}$$

And spectral norm of D^i and $\widehat{W}^i$ are both less than or equal to 1, it is easy to prove

$$\sigma\left(D^L\widehat{W}^L D^{L-1}\widehat{W}^{L-1}\cdots D^i\widehat{W}^i\cdots D^1\widehat{W}^1\right)\leq\sigma\left(D^L\right)\sigma\left(\widehat{W}^L\right)\cdots\sigma\left(D^1\right)\sigma\left(\widehat{W}^1\right)\leq 1 \tag{3.21}$$

It is found that there are

$$\frac{\|D(x+\Delta x)-D(x)\|}{\|\Delta x\|}\leq\sigma\left(D^L\right)\sigma\left(\widehat{W}^L\right)\cdots\sigma\left(D^1\right)\sigma\left(\widehat{W}^1\right)\leq 1 \tag{3.22}$$

That is, the 1-Lipschitz restriction is satisfied, and it is "hard" satisfied with theoretical guarantee! In addition, in practice, it is very resource-intensive to compute the singular values of the matrix, so it is not appropriate to use the singular value decomposition to obtain all the singular values of the matrix, and then take the maximum of them as the Spectral norm, so we use the power method, which can quickly compute the maximum singular values of the matrix, and its calculation process is as follows: for the matrix $A_{m\times n}$ given m dimensional random initial vector μ_0 and n dimensional random initial vector v_0, then multiple iterations are performed to calculate:

$$v_i=A^T u_{i-1}/A^T u_{i-1} \tag{3.23}$$

$$u_i=Av_{i-1}/\|Av_{i-1}\| \tag{3.24}$$

After a sufficient number of iterations, there are:

$$\sigma(A)\approx u_n^T W v_n \tag{3.25}$$

In fact, we only need to do one iteration to get a valid spectral norm calculation result. The reader should be reminded that SNGAN uses spectral normalization weights $\widehat{W}$ to compute the object function, but the weights are updated on the basis of W (Fig. 3.11).

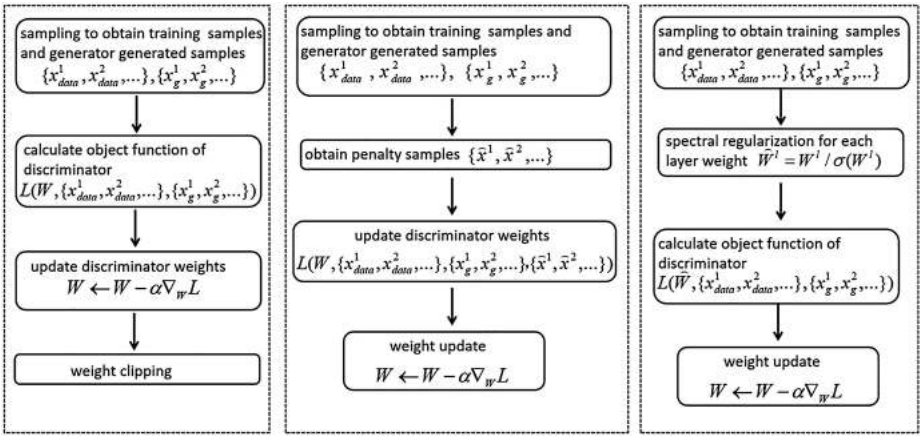


Fig. 3.11 Comparison of three algorithms for handling 1-Lipschitz

SNGAN uses spectral norm to spectrally regularize the weights of each layer of the neural network, thus ensuring that the discriminator satisfies the 1-Lipschitz restriction, and the increased computational effort of Spectral normalization is not large, but its theoretical shortcoming is that the condition requiring each layer of the discriminator to satisfy the 1-Lipschitz restriction is a bit too "hard," and it reduces the search range of the parameter space. Currently, Spectral normalization has been applied to many GAN models, especially for image generation tasks, and is not limited to discriminators, but can also be used in generators.

3.4 Consistent Optimization

3.4.1 Ordinary Differential Equations and Euler's Method

Most of the equations that many people are usually exposed to are algebraic equations, transcendental equations, etc., such as $x^2 = 1$. The solution is one or several values, for example, the solution of the above equation is: $x = 1$ or -1. The differential equation is: a slightly more "abstract" equation, which represents the relationship between unknown function $y(x)$, the derivative of the unknown function $y'(x)$ and the independent variable x, such as

$$\frac{\mathrm{d}y}{\mathrm{d}x} - 2x = 0 \tag{3.26}$$

Its solution (if solvable) should be a function or a family of functions, e.g., the analytic solution of the above equation is $y(x) = x^2 + C$. The unknown function $y(x)$ that is a univariate function is called an ordinary differential equation, and if it is a multivariate function, it is called a partial differential equation. For convenience, the independent variable x is written as time t, the differential equation can be used to represent certain time-dependent laws or dynamical systems:

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} = f(\theta, t) \tag{3.27}$$

It should be noted that for ordinary differential equations, only some special types of equations can be solved analytically, most of them are difficult to find analytical solutions, so in practice, we mainly rely on the numerical method to approximate the numerical solution, taking a simple ordinary differential equation with initial values as an example:

$$\begin{cases} \dot{\theta} = \theta - \frac{2t}{\theta} \\ \theta(t_0) = 1 \end{cases} \tag{3.28}$$

Its analytical solution is $\theta = \sqrt{1 + 2x}$, while the numerical solution can only give partial, discrete pairs of approximate numerical values of the independent and dependent variables, for example

t_n	θ_n
0.1	1.1000
0.2	1.1918
0.3	1.2774
0.4	1.3582
0.5	1.4351
0.6	1.5090
0.7	1.5803
0.8	1.6498
0.9	1.718

The Euler method is a very classical first-order numerical method. Given an initial value and a series of discrete time points with fixed intervals h, it is possible to iteratively calculate:

$$\theta_1 = \theta_0 + hf(\theta_0, t_0)$$

$$\theta_2 = \theta_1 + hf(\theta_1, t_1) \ldots\ldots$$

$$\theta_n = \theta_{n-1} + hf(\theta_{n-1}, t_{n-1}) \tag{3.29}$$

The numerical solution of the differential equation is obtained. According to the recurrence relation:

$$\frac{\theta_{n+1}-\theta_n}{t_{n+1}-t_n}=f(\theta_n,t_n) \tag{3.30}$$

In machine learning or neural networks, we make heavy use of gradient descent, which can actually be seen as a dynamical system. Given a certain object function on the training set:

$$L(\theta)=\sum_{i=1}^{N}\frac{1}{N}L\left(x^{(i)};\theta\right) \tag{3.31}$$

In general, for quite complex object functions, it is not possible to solve the optimal solution of the parameters directly in one step, but only "slowly" by some algorithms, such as the classical gradient descent algorithm, where the parameters are constantly updated, leaving a wonderful trajectory in the parameter space, as shown in Fig. 3.12, whose behavior is very similar to that the behavior is very similar to that of a dynamical system. Consider a dynamical system represented by the ordinary differential equation:

$$\dot{\theta}=-\nabla_\theta L(\theta) \tag{3.32}$$

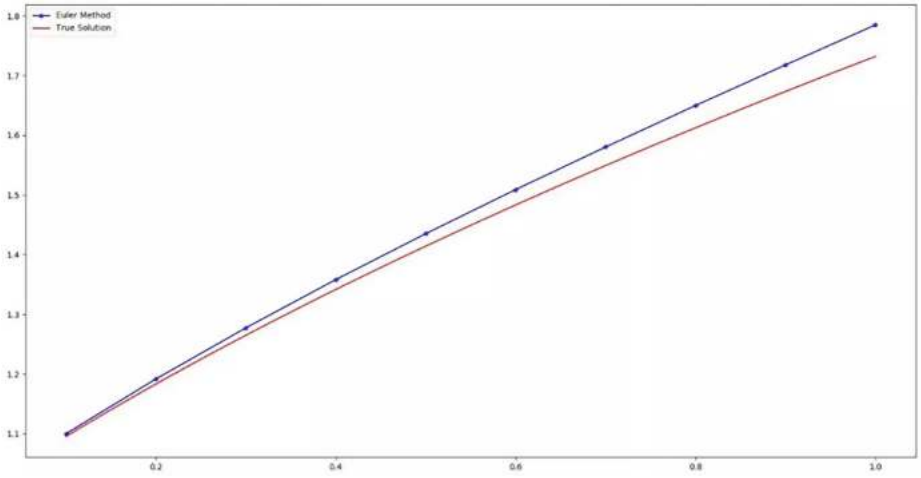


Fig. 3.12 Eulerian solution with true solution

Using Euler's method to solve this dynamical system, the following iterative relations are obtained:

$$\theta_{n+1}=\theta_n-h\nabla_\theta L(\theta) \tag{3.33}$$

If we consider the fixed time interval *h* as learning rate, that is the expression of the very familiar gradient descent algorithm, it should be seen that the so-called gradient descent algorithm, from the dynamics point of view, is to use Euler's method to solve a dynamical system. Of course, we are not only committed to solving the numerical solution of the differential equation or getting the trajectory of the parameters, but more importantly, we want the parameters θ converge to some stable point, so that the dynamical system reaches some stable state, and the object function converges.

3.4.2 GAN Dynamical System

In GAN, we set the optimization objective of the generator is to maximize *f*, and the optimization objective of the discriminator is to maximize *g*. The parameters of the dynamical system are composed of two parts θ (the parameters of the discriminator) and ϕ (the parameters of the generator). Then the differential equation of GAN dynamical system can be written as

$$\begin{pmatrix}\dot{\theta}\\ \dot{\phi}\end{pmatrix}=\begin{pmatrix}\nabla_\theta f(\theta,\phi)\\ \nabla_\phi g(\theta,\phi)\end{pmatrix} \tag{3.34}$$

The entire system is still updated iteratively using the gradient descent method, and it can be understood as using the simultaneous gradient descent algorithm as follows [4, 5]:

$$\theta_{n+1} = \theta_n + h\nabla_\theta f(\theta, \phi) \tag{3.35}$$

$$\phi_{n+1} = \phi_{\mathrm{n}} + h\nabla_\theta g(\theta, \phi) \tag{3.36}$$

That is, the parameters of the generator and discriminator are updated simultaneously at a time step, and their parameter trajectories are shown in Fig. 3.13.

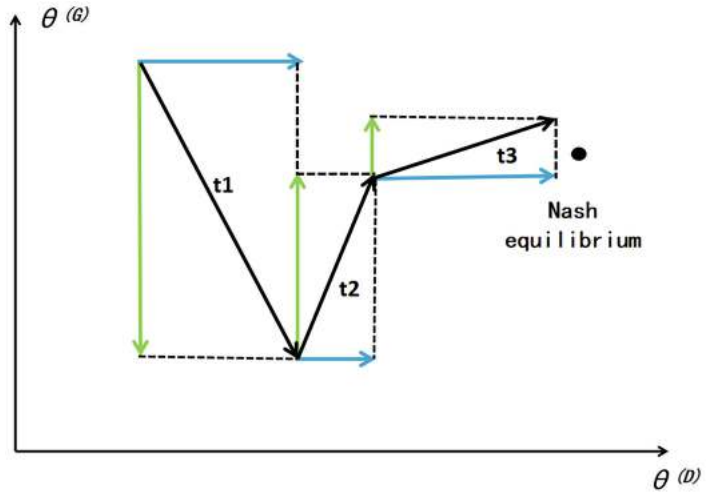


Fig. 3.13 Simultaneous gradient descent method

It should be noted that usually in GAN we use alternating gradient descent, which has some differences (but in many cases does not affect the final conclusion), i.e., alternatingly updating the parameters of the generator and the discriminator in turn, with the parameter trajectory shown in Fig. 3.14:

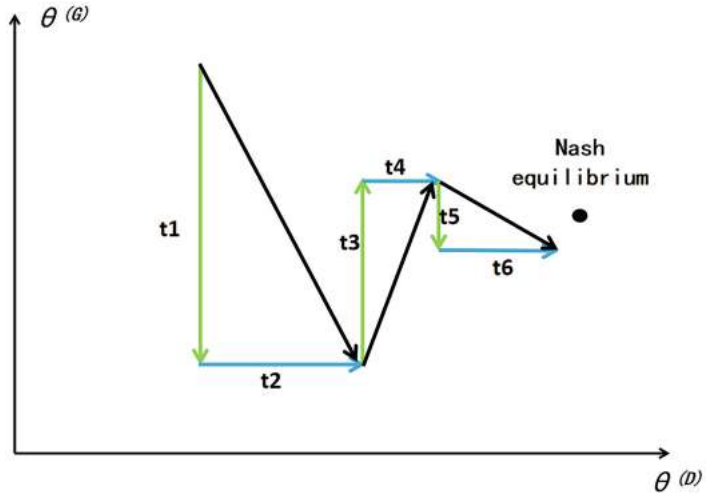


Fig. 3.14 Alternating gradient descent method

GAN is not looking for a global optimal solution, but for a local optimal solution. We want the trajectory of the dynamical system to enter a local convergence point, or Nash equilibrium, with continuous iterations. The Nash equilibrium point is defined as

$$\bar{\theta} = \mathrm{argmin}_\theta\ f(\theta, \phi) \tag{3.37}$$

$$\overline{\phi} = \operatorname{argmin}_{\phi} g(\theta, \phi) \tag{3.38}$$

It is easy to prove that for a zero-sum game ($f = -g$) at the Nash equilibrium point, its Jacobi matrix:

$$\begin{bmatrix} \nabla_{\theta}^{2} f(\theta, \phi) & \nabla_{\theta,\phi} f(\theta, \phi) \\ \nabla_{\phi,\theta} g(\theta, \phi) & \nabla_{\phi}^{2} g(\theta, \phi) \end{bmatrix} \tag{3.39}$$

is negative definite. In turn, one can determine if local convergence is reached by checking the properties of the Jacobi matrix. If at some point, its first-order derivative is 0:

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \nabla_{\theta} \mathrm{f}(\theta, \phi) \\ \nabla_{\phi} \mathrm{g}(\theta, \phi) \end{pmatrix} \tag{3.40}$$

and its Jacobi matrix is negative definite, then the point is a Nash equilibrium point.

Without going into the GAN first, a particularly important proposition related to the convergence of the dynamics is introduced by considering a function of the following form:

$$F(x) = x + hG(x) \tag{3.41}$$

Where h is greater than 0. There is a proposition that: If there is a special point (fixed point $\overline{x}$) such that $F(\overline{x}) = \overline{x}$, and at that fixed point, the absolute values of all the eigenvalues of the Jacobian matrix F'($\overline{x}$) (the eigenvalues of the asymmetric matrix are complex) are less than 1, then starting from any point in a small neighborhood of that fixed point, use a numerical iterative method of the following form:

$$\begin{gathered} x^{(0)} + hG\left(x^{(0)}\right) = F\left(x^{(0)}\right) \to: x^{(1)} \\ x^{(1)} + hG\left(x^{(1)}\right) = F\left(x^{(1)}\right) \to: x^{(2)} \\ \cdots\cdots \\ x^{(n-1)} + hG\left(x^{(n-1)}\right) = F\left(x^{(n-1)}\right) \to: x^{(n)} \end{gathered} \tag{3.42}$$

Then $F(x)$ will eventually converge to $\overline{x}$. For an intuitive description, the numerical iterative process described above is actually using numerical iterations to find the intersection of the two functions $y = x$ and $y = x + hG(x)$, as illustrated in Fig. 3.15. We have a good conclusion about convergence, and the way the value is overlapped is consistent with the actual GAN training method, and we consider docking the GAN to this conclusion. Now, the x corresponds to the parameters of the GAN:

$$x : \begin{pmatrix} \theta \\ \phi \end{pmatrix}$$

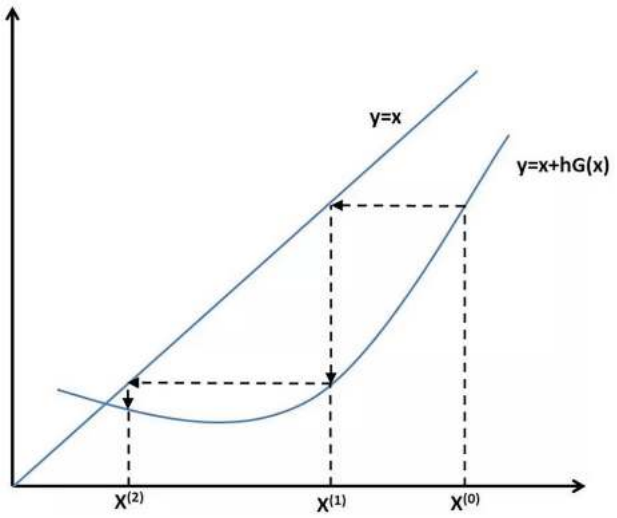


Fig. 3.15 Schematic representation of the numerical iteration process

h can correspond to the learning rate during training, and *G*(*x*) then corresponds to the vector field *v*:

$$G(x) : v(\theta, \phi) = \begin{pmatrix} \nabla_\theta f(\theta, \phi) \\ \nabla_\phi g(\theta, \phi) \end{pmatrix} \tag{3.43}$$

Thus, it appears that the equation expresses the meaning of parameter updating using simultaneous gradient descent (harmlessly due to writing the objective function in max form, which is precisely gradient ascent):

$$\begin{pmatrix} \theta \\ \phi \end{pmatrix} \coloneqq \begin{pmatrix} \theta \\ \phi \end{pmatrix} + h \begin{pmatrix} \nabla_\theta f(\theta, \phi) \\ \nabla_\phi g(\theta, \phi) \end{pmatrix} \tag{3.44}$$

After mapping the general form to the GAN, the previous conclusion about convergence is again considered, i.e., if there exists a point (i.e., fixed point) that satisfies the following form, and at the fixed point, the absolute values of all eigenvalues of the Jacobian matrix of the vector field *v* are less than 1, then the iteration starts from any point in one of the neighborhoods of the fixed point, and eventually convergence will be achieved. In fact, the former condition is nothing but to say that at the fixed point, the *v* = 0. Then we can "check" the training process of GAN, when there is a parameter point with gradient of 0, "check" whether the eigenvalues of the Jacobi matrix of its vector field are all within the unit circle, if they are, the GAN iterations eventually converge to that point.

It seems that it is not difficult to train a GAN to find a parameter point with a gradient of 0, but it may be difficult to achieve the second condition, which is to achieve that the absolute values of all eigenvalues of all eigenvalues of the vector field *v* at the fixed point are less than 1. Consider the expression for the general case:

$$F(x) = x + hG(x) \tag{3.45}$$

The Jacobi matrix of F(*x*) is given by

$$F'(x) = I + hG'(x) \tag{3.46}$$

To perform the eigenvalue decomposition, the unit matrix *I* has an eigenvalue of real number 1, and considering that in general the matrix G′(*x*) is an asymmetric matrix, then its eigenvalues must be complex. Let the eigenvalue decomposed by G′(*x*) be $\lambda = a + bi$, so the eigenvalue decomposed by F′(*x*) is $(ha + 1) + hbi$, as shown in Fig. 3.16 left. The eigenvalues can easily run out of the unit circle. To ensure that its absolute value is less than 1 (i.e., in the unit circle), first ensure that a is less than 0. (This condition cannot be satisfied when *a* is greater than or equal to 0), as shown in Fig. 3.16 right.

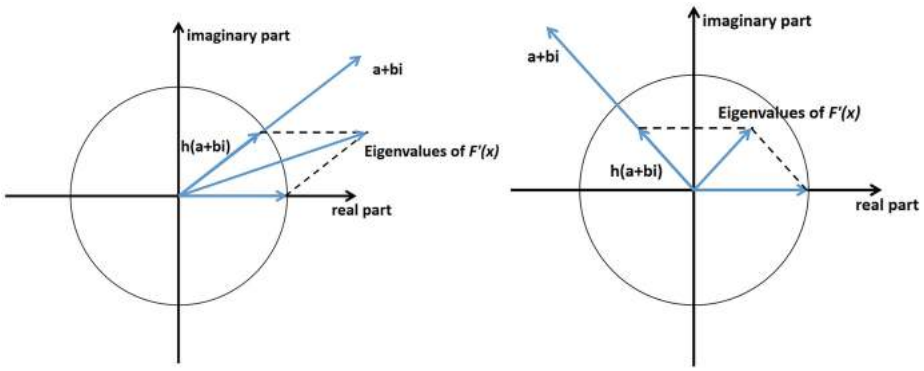


Fig. 3.16 Schematic diagram of Eigenvalues

The real part of the eigenvalues obtained from the decomposition of $G'(x)$ is negative:

$$[(1+ha)+hbi][(1+ha)-hbi]<1 \Leftrightarrow h<\frac{1}{|a|}\frac{2}{1+\left(\frac{b}{a}\right)^2} \tag{3.47}$$

In other words, to enter the convergence state, the real part of the eigenvalues must be negative, and the learning rate h must be small enough. And the upper bound of h depends on the eigenvalues. But there is a paradox here, if set the learning rate too small, training time will become very long. Similarly, it is necessary to ensure that the real part of all eigenvalues of the Jacobian matrix of the vector field v is negative.

$$\begin{bmatrix} \nabla_\theta^2 f(\theta,\phi) & \nabla_{\theta,\phi}\, f(\theta,\phi) \\ \nabla_{\phi,\theta}\, g(\theta,\phi) & \nabla_\phi^2\, g(\theta,\phi) \end{bmatrix} \tag{3.48}$$

However, in practice, this condition is unlikely to be met, especially when there are cases where the real part is almost zero and the value of the imaginary part is relatively large, and the learning rate has to be set small enough. Notice that the Jacobi matrix of vector field v is related to the objective function (f, g) of the generator and discriminator, consider adjusting f and g, so that the real part of the eigenvalues at the fixed points is negative.

3.4.3 Consistent Optimization

Consensus optimization is a theoretically better method that does a little "fiddling" to make the real part of the eigenvalues as negative as possible [4]. Consider first the general form:

$$F(x)=x+hA(x)G(x) \tag{3.49}$$

Define γ is greater than 0, A is the invertible matrix with the expression

$$A(x)=I-\gamma G'(x)^T \tag{3.50}$$

For the sake of rigor, a clarification is needed: If an x is a fixed point of $F(x) = x\ hG(x)$, then that x is also a fixed point of $F(x) = x\ hA(x)G(x)$. The addition of $A(x)$ to the equation does not affect the fixed point, where it may have converged before, and it may still converge at that point after. And there is

$$F'(x)=I+h\big(A'(x)G(x)+A(x)G'(x)\big)=I+hA(x)G'(x)=I+h\Big[I-\gamma G'(x)^T\Big]G'(x)=I+hG'(x)$$

It can be seen that, compared to the above expression, the new addition causes the eigenvalues to be shifted in the negative direction of the real part (the new addition is a negative definite matrix, whose eigenvalues are necessarily negative real), as shown in Fig. 3.17.

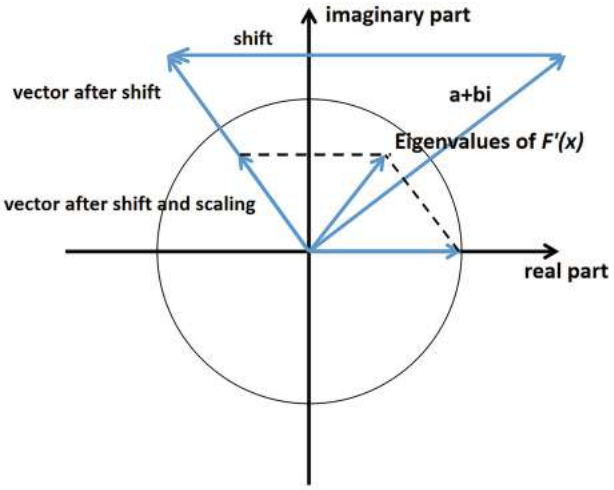


Fig. 3.17 Schematic diagram of consistent optimization

If the hyperparameters γ is reasonable, it is "promising" to ensure that the eigenvalues all fall within the unit circle. Now, we adapt the above approach to the GAN by modifying the objective functions of the generator and discriminator as follows:

$$\max_{\theta} f(\theta, \phi) - \gamma L(\theta, \phi) \tag{3.52}$$

$$\max_{\theta} g(\theta, \phi) - \gamma L(\theta, \phi) \tag{3.53}$$

where $L(\theta, \phi) = \frac{1}{2} \parallel v(\theta, \phi) \parallel^2$, then:

$$\begin{pmatrix} \theta \\ \phi \end{pmatrix} \coloneqq \begin{pmatrix} \theta \\ \phi \end{pmatrix} + h \begin{pmatrix} \nabla_{\theta}[f(\theta, \phi) - \gamma L(\theta, \phi)] \\ \nabla_{\phi}[g(\theta, \phi) - \gamma L(\theta, \phi)] \end{pmatrix} \tag{3.54}$$

can be reduced to:

$$\begin{pmatrix} \theta \\ \phi \end{pmatrix} \coloneqq \begin{pmatrix} \theta \\ \phi \end{pmatrix} + hv(\theta, \phi) - \gamma v'(\theta, \phi)^{\mathrm{T}} v(\theta, \phi) \tag{3.55}$$

According to the previous conclusion, if γ is reasonable and the learning rate h is small enough, all the eigenvalues will fall into the unit circle and the parameters will enter the fixed point (i.e., the Nash equilibrium state) as the iterations are continuously updated. The added regularization term does not solve the problem of requiring a small enough learning rate, but it "guarantees" that the feature values fall into the unit circle as much as possible. As a final note, in a general GAN, the objective functions of the generator and discriminator are of opposite signs, but we add regularization terms of the same sign to them at the same time, and their optimization objectives are the same in the regular term part, so it is called consistent optimization.

3.5 GAN Training Techniques

3.5.1 Feature Matching

In GAN, the discriminator D outputs a scalar between 0 and 1 indicating the probability that the accepted samples are from the real dataset, and the training goal of the generator G is to try to maximize the value of this scalar. From the perspective of feature matching [6], the discriminator D consists of two parts; firstly, the first half of the discriminator $f(x)$ extracts abstract features, and the neural network in the second half determines the classification based on the abstract features, as shown in Fig. 3.18.

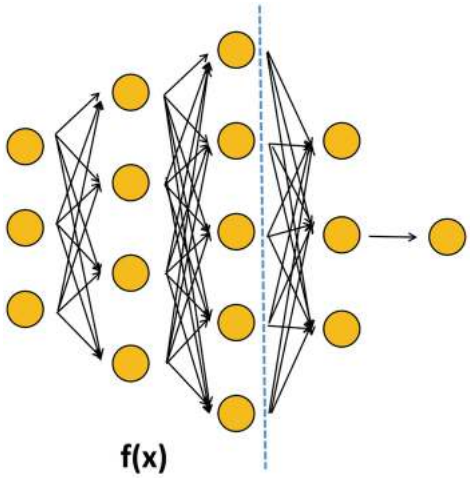


Fig. 3.18 Schematic diagram of feature matching

$f(x)$ denotes the output of the discriminator up to the activation function of a neuron in the middle layer. When training the discriminator, we try to find a way $f(x)$ to extract features that can distinguish between two types of samples and when training the generator, we can stop focusing on the probabilistic output of $D(x)$, we can focus on: whether the abstract features extracted with $f(x)$ from the sample generated by the generator match the abstract features extracted with $f(x)$ in the real sample. In addition, to match the distribution of these two abstract features, consider their first-order statistical feature: the mean, which can be rewritten as the objective function of the generator

$$\| \mathbb{E}_{x \sim p_{data}} f(x) - \mathbb{E}_{z \sim p_z} f(G(z)) \|_2^2 \tag{3.56}$$

With such an approach, we can keep the generator from overtraining, make the training process relatively stable, and mitigate the mode collapse problem to some extent.

3.5.2 Historical Averages

Historical averaging [6] is a very simple way to add a term to the object function of a generator or discriminator:

$$\| \theta - \frac{1}{t} \sum_{i=1}^{t} \theta[i] \|^2 \tag{3.57}$$

where θ is the parameter of the generator or discriminator network, and θ[i] is the parameter of the ith iteration. This makes the parameters of the discriminator or generator do not suddenly produce large fluctuations, and intuitively, when the Nash equilibrium is about to be reached, the parameters would keep adjusting around the Nash equilibrium and not easily run out. This technique does help to enter the Nash equilibrium and thus converge the object function when dealing with low-dimensional problems, but the help may be limited when facing high-dimensional problems in GAN.

3.5.3 Single-Side Label Smoothing

The label smoothing (label smoothing) method was first proposed in 1980 [7], and it has a very wide range of applications in classification problems, mainly to solve the overfitting problem. In general, the last layer of our classifier uses a softmax layer to output the classification probability (Sigmoid is just a special case of softmax), and we illustrate the effect of label smoothing with a two-classification softmax function. For a given sample x, if we do not use label smoothing, but only use "hard" labels, the cross-entropy object function is

$$-\log \frac{e^{z_1}}{e^{z_1} + e^{z_2}} \tag{3.58}$$

This time the classifier is trained by minimizing the cross-entropy object function, essentially making:

$$p(y=1)=\frac{\mathrm{e}^{z_1}}{\mathrm{e}^{z_1}+\mathrm{e}^{z_2}}\to 1 \tag{3.59}$$

$$p(y=0)=\frac{\mathrm{e}^{z_2}}{\mathrm{e}^{z_1}+\mathrm{e}^{z_2}}\to 0 \tag{3.60}$$

is actually what makes z_1 tends to infinity, and z_2 tends to 0. For a given sample *x*, making the value of z_1 infinitely large (of course, this is not possible in practice) and making z_2 tend to 0, endlessly fitting the label 1, produces an overfitting and reduces the generalization ability of the classifier. If label smoothing is used, for a given sample *x* with a class of 1, the smoothing label is [1 − *ε*, *ε*], then the cross object function is

$$-(1-\epsilon)\log\frac{\mathrm{e}^{z_1}}{\mathrm{e}^{z_1}+\mathrm{e}^{z_2}}-\epsilon\log\frac{\mathrm{e}^{z_2}}{\mathrm{e}^{z_1}+\mathrm{e}^{z_2}} \tag{3.61}$$

When the object function reaches its minimum value, there is:

$$z_1-z_2=\log\frac{\epsilon}{1-\epsilon} \tag{3.62}$$

By selecting appropriate parameters, the theoretical optimal solutions z_1 and z_2 exist with a fixed constant difference (determined by *ε*), thus preventing the situation where z_1 approaches infinity and vastly exceeds z_2. If this trick is used in the discriminator of GAN, i.e., the sample output probability value 0 generated by the generator becomes *β*, then the single-sample cross-entropy object function generated by the generator is

$$-\beta\log\left[D(x)\right]-(1-\beta)\log\left[1-D(x)\right] \tag{3.63}$$

While labeling the samples in the dataset from 1 down to *α*, then the one-sample cross-entropy object function in the dataset is

$$-\alpha\log\left[D(x)\right]-(1-\alpha)\log\left[1-D(x)\right] \tag{3.64}$$

The total crossover object function is the sum of two terms:

$$\mathbb{E}_{x\sim\mathrm{p_g}}\{-\beta\log\left[D(x)\right]-(1-\beta)\log\left[1-D(x)\right]\}+\mathbb{E}_{x\sim\mathrm{p_{data}}}\{-\alpha\log\left[D(x)\right]-(1-\alpha)\log\left[1-D(x)\right]\} \tag{3.65}$$

Its optimal solution *D*(*x*) is:

$$D(x)=\frac{\alpha p_{\mathrm{data}}(x)+\beta p_{\mathrm{g}}(x)}{p_{\mathrm{data}}(x)+p_{\mathrm{g}}(x)} \tag{3.66}$$

In actual training, there are a large number of such instances *x*: its probability distribution in the training dataset is 0, while the probability distribution generated by the generator is not 0, but after passing through the discriminator, the output is *β*. In order to promptly "identify" the sample, it is preferable to reduce *β* to 0, which is referred to as one-sided label smoothing.

3.5.4 Virtual Batch Regularization

Virtual batch normalization (VBN) [6] is an improved version of the batch normalization (BN) technique. The advent of BN greatly improves the training speed and convergence speed and also reduces the network initialization requirements. However, BN can make the sample output highly dependent on the other samples in the same batch. To avoid this problem, VBN defines a reference batch, which is selected and fixed from the beginning, and the statistical values of the reference batch are used for the regularization of each batch during training. Since we use the same reference batch throughout the training, there is a risk of overfitting, and to mitigate this, the reference batch can be combined with the current batch to compute the

regularization parameters. Since VBN's need to compute two batches when performing forward propagation is relatively computationally expensive, it is only recommended to use the VBN technique in the generator.

3.5.5 TTUR

In the TTUR (two time-scale update rule) [8] method, different learning rates are used for the discriminator *D* and the generator *G*. It is generally believed that the learning ability of the generator is weaker than that of the discriminator, and the discriminator has to learn the new mode before the generator can guide the generator, but the learning ability does not only depend on the size of the learning rate, but also on the objective function, the current value of the objective function, the optimization algorithm, the network architecture, etc. Therefore, the learning rate of the generator can be larger than that of the discriminator, and more fundamentally, the learning rates of both should be chosen independently of each other should be considered independently of each other. In addition, according to some experimental results, the effect of different learning rates of generators and discriminators is usually better than the same learning rate of both.

3.5.6 Zero Center Gradient

A penalty term is used in WGAN-GP to make the discriminator output on the penalty sample with a gradient close to 1 with respect to the input, which is due to 1-Lipschitz, but another similar 0-centered gradient penalty term can achieve equally good results [9], i.e.,

$$\lambda \mathbb{E}_x \parallel \nabla_x D(x) \parallel^2 \tag{3.67}$$

For this penalty term, we can interpret the use of the W-divergence measure the distance between p_{data} and p_{g}:

$$\max_{\theta} \mathbb{E}_{x \sim p_{\text{data}}}[D(x)] - \mathbb{E}_{x \sim p_g}[D(x)] - \lambda \mathbb{E}_x \parallel \nabla_x D(x) \parallel^p \tag{3.68}$$

When $p = 2$ then the 0-center gradient penalty objective function is obtained. For the sampling method of the penalty term samples, WGAN-GP uses the approach of interpolating between the training samples and the generation samples, or it can choose (1) random interpolation within the training samples and random interpolation within the generation samples, (2) mixing the two types of samples and then randomly selecting two samples for interpolation, (3) mixing and sampling the two types of samples, (4) sampling from the training samples, (5) sampling from the generation samples, etc. Empirically, the various sample collection methods for penalty terms do not differ too significantly and need to be chosen specifically for the specific task.

3.5.7 Other Recommendations

There are some other suggestions for GAN training, the principles of which are not fully explained, but have some effect in practice, [10] as follows:

(1) Scaling the image pixel values between -1 and 1.

(2) Using tan h as the output layer of the generator.

(3) Using a normal distribution for noise *z*.

(4) BN can usually be trained stably.

(5) Use LeakyReLU as the activation function as much as possible.

(6) Upsampling using PixelShuffle and transposed convolution.

(7) Avoid maximizing the pool for downsampling and use convolution with step size and average pooling.

(8) In the GAN, use the Adam optimizer if possible.

(9) Early tracking of training failure signals and stopping training, e.g., discriminant object function rapidly converging to zero.

(10) During the training and testing phases, a certain amount of noise is introduced using DropOut at certain layers in the generator.

3.6 Mode Collapse

3.6.1 Two Solutions for Mode Collapses

In theory, if the GAN can converge to the optimal Nash equilibrium point, the mode collapse problem will be solved naturally. For example, in Fig. 3.19, the red line represents the probability density function of the generated data, while the blue line represents the probability density function of the training dataset. Originally, the red line has only one mode, which means that the generator will almost only produce one kind of sample (one peak represents a sample mode). While in the theoretical optimal solution, the red line and the blue line overlap, so the sampling in the generator can naturally get almost three kinds of samples, which are consistent with the performance of the data in the training set.

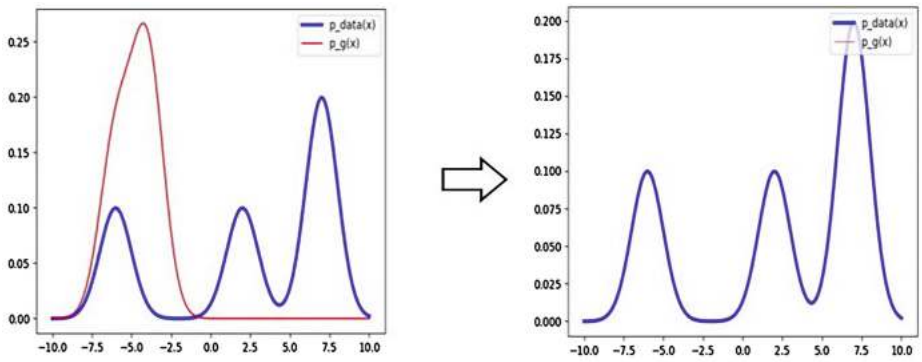


Fig. 3.19 Global optimal solution to avoid mode collapse

Of course, the global optimal solution is almost never reached in practice, and our seemingly convergent GAN actually only enters a local optimal solution. Therefore, in general, we have two ideas to solve the mode collapse problem:

1. Improve the learning ability of GAN to enter a better local optimal solution, as shown in Fig. 3.20 below, by training the red line to slowly approach the shape and size of the blue line, a better local optimal will naturally have more modes, which intuitively can alleviate the mode collapse problem to some extent. For example, unrolledGAN, which increases the "prophetic" capability of the generator.
2. Abandoning the search for better solutions and only explicitly requiring the GAN to capture more modes on top of the GAN (as shown in Fig. 3.21), although the similarity between the red and blue lines is not high, but "forcing" adds diversity to the generated samples, and most of these methods directly modify the structure of the GAN, such as MADGAN and VVEGAN.

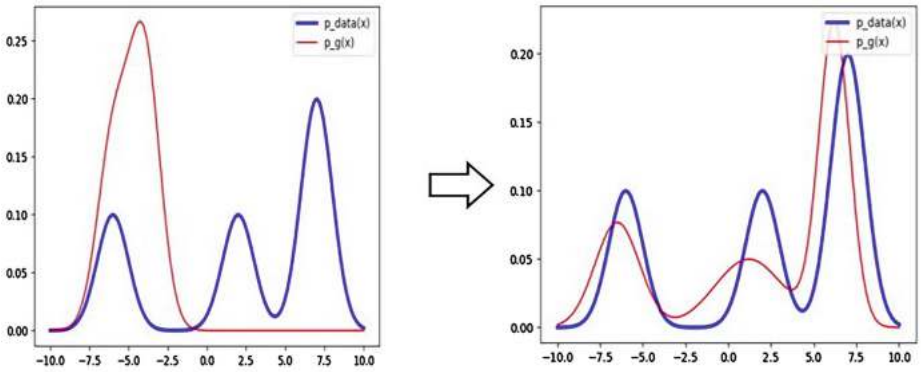


Fig. 3.20 Enhancing learning capacity

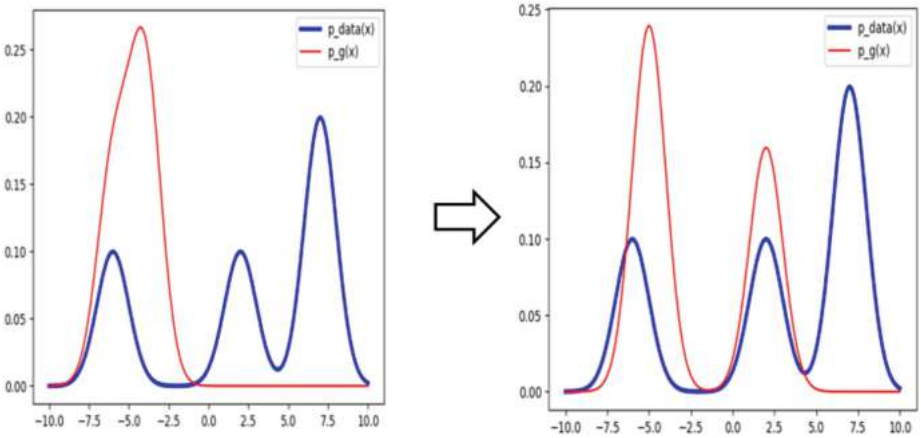


Fig. 3.21 Forced addition of sample diversity

3.6.2 unrolledGAN

First of all, it should be noted that the fact that the generator simply aggregates the samples under a few high probability peaks at a certain moment is not the fundamental reason why we hate mode collapse, if the generator can automatically adjust the weights and spread the generated samples over the entire stream of training data, it can automatically jump out of the current mode collapse state, and the generator does theoretically "have," as GoodFellow proved that GAN would theoretically achieve an optimal solution.

But the reality is that the constant training of the generator does not make it learn to improve the diversity of the generated samples; the generator just keeps shifting and aggregating the samples from one peak to another. This process is "endless" and cannot break out of the mode collapse cycle. Whenever you terminate training, you face mode collapse, but at different moments, the generated samples are clustered under different peaks. There is a certain inevitability to this occurrence, and we first describe this process schematically using the primitive form GAN with the objective function of

$$\min_{\phi}\max_{\theta} \mathrm{V} = \mathbb{E}_{x\sim p_{data}} \log\left(D(x)\right) + \mathbb{E}_{z} \log\left(1 - D(G(z))\right) \tag{3.69}$$

The probability distribution of the real dataset is shown in Fig. 3.22, and the generator generates the following distribution of green samples:

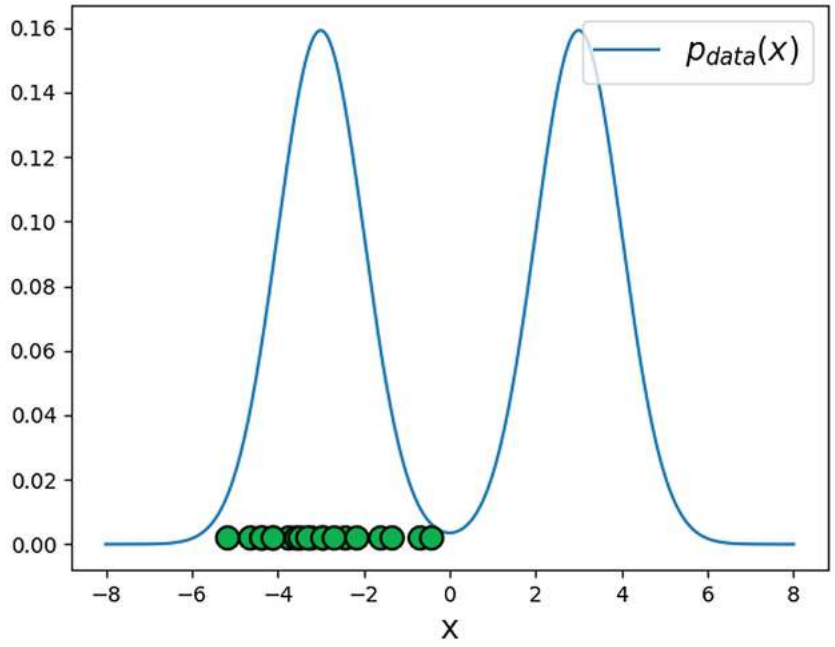


Fig. 3.22 Distribution of the samples generated by the generator

We start by updating the discriminator:

$$\theta = \underset{\theta}{\operatorname{argmax}}\ \mathrm{V}(\phi, \theta) \tag{3.70}$$

Assuming that the discriminator reaches the optimal state, the expression should be

$$D(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)} \tag{3.71}$$

Correspondingly, the image of function $D(x)$ is (Fig. 3.23)

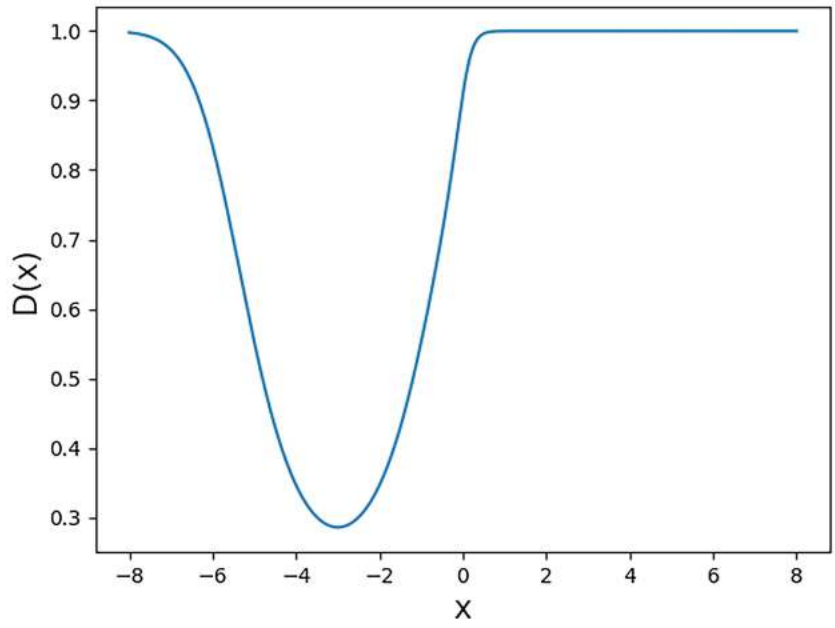


Fig. 3.23 Function image of $D(x)$

It can be seen that at this point the discriminator immediately "suspects" authenticity of sample nearby $x = -3$, and next updates the generator:

$$\phi = \underset{\phi}{\operatorname{argmin}}\ \mathrm{V}(\phi, \theta) \tag{3.72}$$

In this case, the generator will be very "helpless," and the best way to minimize the objective function is to aggregate the samples to $x = 3$ nearby, i.e. (Fig. 3.24),

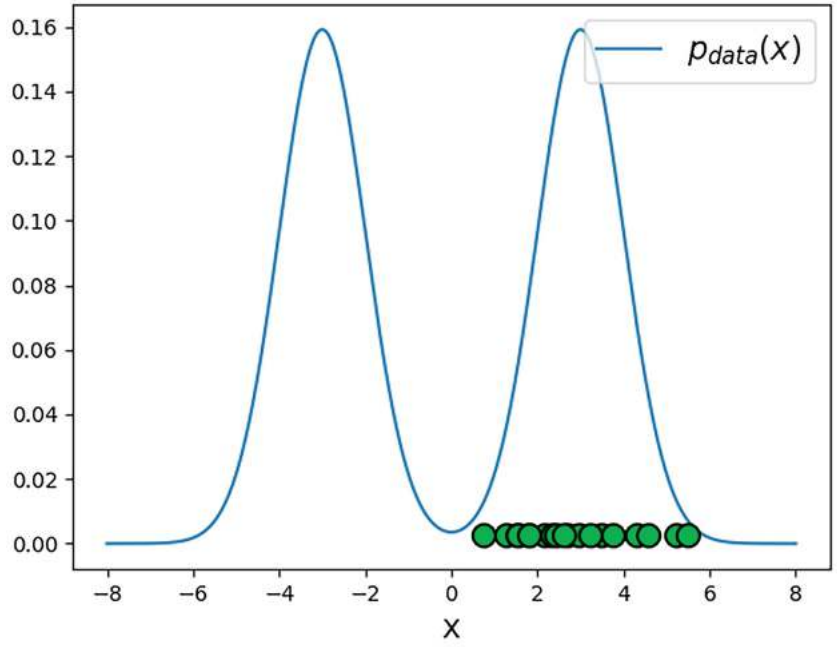


Fig. 3.24 Sample aggregation from around $x = 3$

Update the discriminator again, same process as above, the discriminator will immediately "suspect" the authenticity of sample points nearby $x = 3$. The cycle of such bad results will continue. In this regard, unrolledGAN argues that it is the lack of foresight of the generator that leads to the dilemma of not being able to escape from the collapse of the model, and that each time the generator updates its parameters, it only considers the optimal solution that can be obtained in the current state of the generator and the discriminator is not the optimal solution in the long run [11] .

We improve the "foresight" of the generator by making some improvements. Specifically, the objective function of the discriminator remains unchanged, and the parameters are updated in a gradient descent manner successively K times as follows:

$$\theta^0 = \theta$$
$$\cdots\cdots$$
$$\theta^K = \theta^{K-1} + \eta \frac{\partial V\left(\phi, \theta^{K-1}\right)}{\partial \theta^{K-1}} \tag{3.73}$$

And the optimization objective of the generator is modified as follows:

$$\phi = \underset{\phi}{\operatorname{argmin}}\ \mathrm{V}\left(\phi, \theta^K(\phi, \theta)\right) \tag{3.74}$$

That is, when the generator is updated, it will not only consider the state of the current generator, but also additionally consider the state of the discriminator after *K* times update, and combine the two information to make the optimal solution. The change of its gradient is as follows:

$$\frac{\mathrm{dV}(\phi,\theta)}{\mathrm{d}\phi}=\frac{\partial \mathrm{V}\left(\phi,\theta^{K}(\phi,\theta)\right)}{\partial \phi}+\frac{\partial \mathrm{V}\left(\phi,\theta^{K}(\phi,\theta)\right)}{\partial \theta^{K}(\phi,\theta)}\frac{\partial \theta^{K}(\phi,\theta)}{\partial \phi} \tag{3.75}$$

The first is the calculated gradient in the familiar form of a standard GAN, while the second is an additional term that takes into account the state of the discriminator after *K* updates. Let's now look at the earlier problem where the unrolledGAN jumps out of the mode collapse loop. With the same initial state, the generator faces the following two possibilities when performing the next update, with the previously mentioned mode collapse state on the left and the ideal sample generation state on the right (Fig. 3.25):

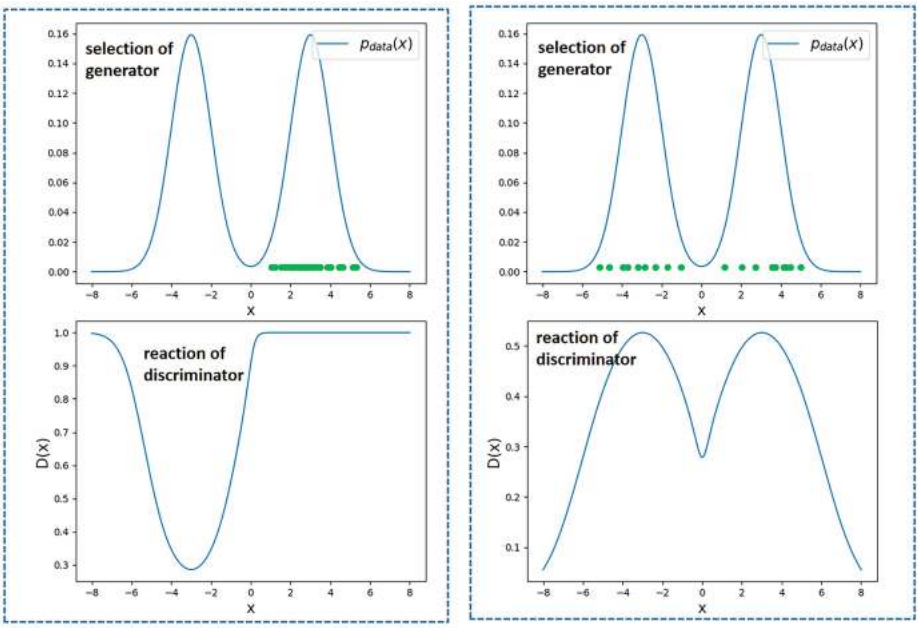


Fig. 3.25 Effect of using unrolledGAN

It is calculated that choosing the right side will produce a smaller objective function value than choosing the left side, so in practice, the generator will perform the gradient update to the right side of the state and thus jump out of the mode collapse. It can be seen that the core reason for the generator to jump out of the mode collapse is that the update parameters not only consider the current state, but also additionally consider the reaction of discriminator, but it should be noted that this is a significant increase in computational effort.

3.6.3 DRAGAN

The parameter optimization problem of GAN is not a convex optimization problem and there are many local Nash equilibria. Even if the GAN enters a certain Nash equilibrium state and the object function exhibits convergence, it can still produce a mode collapse, and we consider that the parameters enter a bad local equilibrium point at this time. Through practice, it is found that when the GAN has a mode collapse problem, it is usually accompanied by the following performance: when the discriminator updates the parameters near the training sample, its gradient value is very large, so the solution of DRAGAN [12] is: for the discriminator, the gradient penalty term is imposed near the training sample: the

$$\mathbb{E}_{x\sim p_{\mathrm{data}},\delta\sim \mathrm{N_d}(0,\mathrm{c}I)}[\| \nabla_x D(x+\delta) \| -1]^2 \tag{3.76}$$

This approach attempts to construct linear functions in the vicinity of the training samples since linear functions are convex functions with globally optimal solutions. It should be additionally noted that the form

of DRAGAN is quite similar to WGAN-GP, except that WGAN-GP imposes a gradient penalty in the full sample space, while DRAGAN only imposes a gradient penalty near the training samples.

3.6.4 MADGAN and MADGAN-Sim

The MAD-GAN and its variants introduced in this section represent one of the second class of methods [13]. Its core idea is: even though a single generator can generate mode collapse problems, if multiple generators are constructed simultaneously and each generator is allowed to generate different modes, the combination of such multiple generators can guarantee the diversity of the generated samples, as in the following figure with three generators (Fig. 3.26):

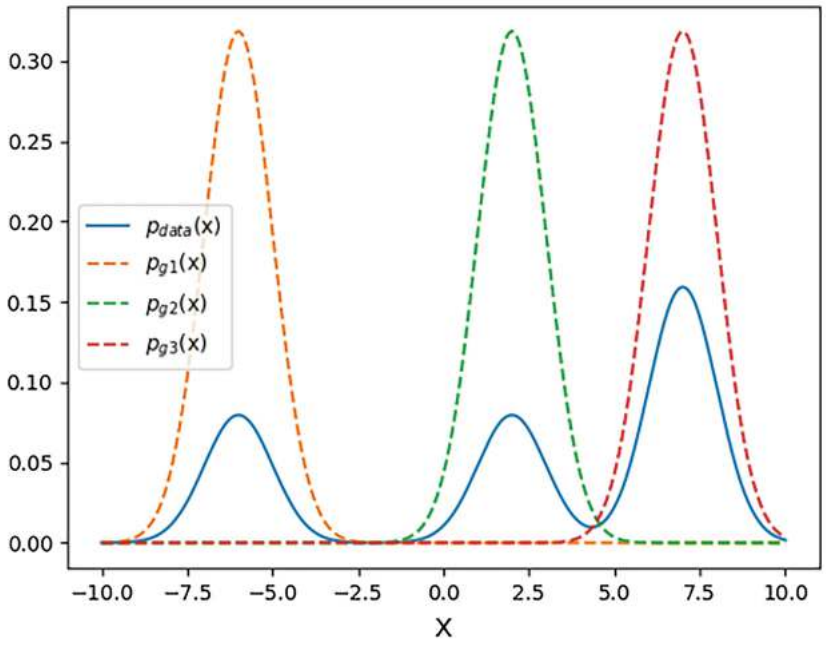


Fig. 3.26 Distribution of the three generators

It is important to note that it does not make much sense to simply add several generators that are isolated from each other; they may merge into the same state and do not add diversity, such as the three generators in Fig. 3.27:

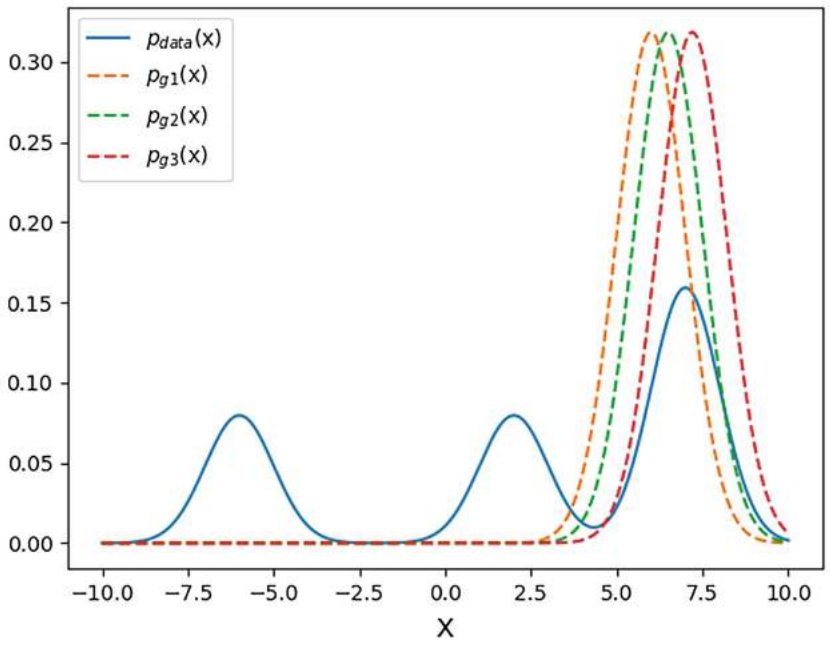


Fig. 3.27 Nonsensical multiple generators

Ideally, multiple generators would be "linked" to each other, and different generators would try to produce dissimilar samples, and all of them would be able to cheat the discriminator. In the MAD (Multi-agent diverse) GAN, there are *k* standard GAN generators with different weight value, and the purpose of each generator is still to generate false samples to try to deceive the discriminator. For the discriminator, it needs not only to distinguish whether the samples come from the training dataset or from one of the generators (which is still the same as the discriminator of the standard GAN), but also to drive the generators to produce dissimilar samples as much as possible.

Some modifications to the discriminator are needed: change the last layer of the discriminator to *k* + 1 dimensional softmax function. For any input sample *x*, *D*(*x*) is a *k* + 1 dimensional vector, where the first *k* dimension represents the probability that sample x comes from the *k* generators, and the *k* + 1 dimension represents the probability that sample *x* comes from the training dataset. At the same time, the delta function of the *k* + 1 dimension is constructed as a label, so if *x* is from the *i*-th generator, then the *i*-th dimension of the delta function is 1 and the rest is 0, and if *x* is from the training dataset, the *k* + 1 dimension of the delta function is 1 and the rest is 0. Clearly, the objective function of *D* should be minimizing the cross-entropy between D(*x*) and delta function:

$$\max_{\theta} \mathbb{E}_x - \mathrm{H}(\delta, D(x)) \tag{3.77}$$

Intuitively, such an object function would force each *x* to be generated from only one generator as much as possible, and not from the others, expanding it as

$$\max_{\theta} \mathbb{E}_{x\sim \mathrm{p}_{\mathrm{data}}} \log\left[D_{k+1}(x)\right] + \sum_{\mathrm{i}=1}^{\mathrm{k}} \mathbb{E}_{x_{\mathrm{i}}\sim p_{gi}} \log\left[D_i(x_i)\right] \tag{3.78}$$

The generator objective function is:

$$\min_{\phi_{\mathrm{i}}} \mathbb{E}_{z\sim p_z} \log\left[1 - D_{k+1}G_i(z)\right] \tag{3.79}$$

For a fixed generator, the optimal discriminator is

(3.80)

$$D_{k+1}(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + \sum_{i=1}^{k} p_{gi}(x)}$$

$$D_i(x) = \frac{p_{gi}(x)}{p_{\text{data}}(x) + \sum_{i=1}^{k} p_{gi}(x)} \tag{3.81}$$

It can be seen that the form is almost identical to the standard form of GAN, except that different generators "reject" each other to produce different samples. In addition, it can be shown that when

$$p_{\text{data}}(x) = \frac{1}{k} \sum_{i=1}^{k} p_{gi} \tag{3.82}$$

Once again, it can be seen that the probability density function of the generated samples of each generator does not need to approximate the probability density function of the training set in MAD-GAN, each generator is responsible for generating different samples separately, and it is only necessary to ensure that the average probability density function of the generators is equal to the probability density function of the training set.

MAD-GAN-Sim is a "more powerful" version, which not only considers that each generator is responsible for generating different samples, but also considers the similarity of the samples in more detail. The starting point is that samples from different modes should look different, so different generators should generate samples that do not look similar.

This idea is described in mathematical notation as

$$D(G_i(z)) \geq D(G_j(z)) + \Delta(F(g_i(z)), F(g_j(z)))\forall j \in K \backslash i \tag{3.83}$$

where $F(x)$ denotes some kind of mapping from the space of generated samples to the feature space (we can choose the intermediate layer of the generator, the idea is similar to feature value matching). $\Delta(x, y)$ denotes the measure of similarity, and the cosine similarity function is mostly chosen to calculate the similarity of the features corresponding to the two samples. For a given noise input z, consider the sample generation of the ith generator and other generators, if the sample similarity is relatively large, then $D(G_i(z))$ should be much larger than $D(G_j(z))$, and since the value of $D(G_j(z))$ is relatively small, $\mathrm{G_j}(z)$ will adjust to no longer generate the previous similar sample, but to generate other samples. Using this "exclusion" mechanism, we achieve so that different generators should generate samples that do not look similar.

By introducing the above constraints into the generator, we can train the generator in such a way that for any generator, for a given z, if the above conditions are satisfied, the gradient is computed normally like MAD-GAN with

$$\nabla_{\phi_{\mathrm{i}}} \log\left[1 - D(G_i(z))\right] \tag{3.84}$$

If the condition is: not satisfied, the above condition is: added to the objective function as a regularization term, and the gradient is

$$\nabla_{\phi_{\mathrm{i}}} \log\left[1 - D(G_i(z))\right] - \lambda \left[D(G_i(z)) - \frac{1}{k-1} \sum_{j \in K \backslash i} D(G_j(z)) + \Delta(F(g_i(z)), F(g_j(z))) \right] \tag{3.85}$$

The idea of MAD-GAN-Sim is very straightforward and clear, but at the cost of adding a very large amount of computation.

3.6.5 VVEGAN

VVEGAN [14] solves the mode collapse problem by adding an encoder E. We know that the generator $G(z)$ accepts noise (assume Gaussian noise) as input and outputs sample x, while the encoder $E(x)$ accepts the

sample x as input and outputs encoded representation z. If the encoder is well trained, it can be considered as the inverse process of the generator

$$E(x) = G^{-1}(x) \tag{3.86}$$

For example, if a certain noise z^* is passed through the generator to obtain sample x^*, sample x^* can be fed into the encoder to obtain z^* again (Fig. 3.28).

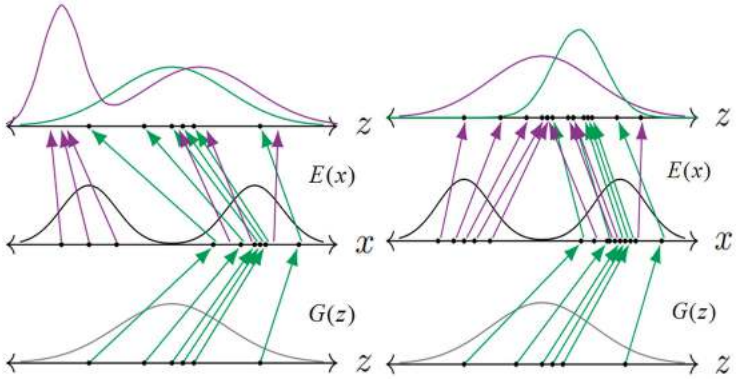


Fig. 3.28 Schematic of encoder detection mode crash

We illustrate the method with a simple example where noise z satisfies the standard normal distribution $N(0, I)$, as shown in the distribution at the bottom of Fig. 3.29, assuming that the distribution of the training data $p_{data}(x)$ is a mixed distribution of two normal distributions, as shown in the middle distribution of Fig. 3.29. When the generator undergoes a mode collapse, $G(z)$ maps all the noise from the normal distribution to the mode on the right side of pdata, then the encoder $E(x)$ is used to map the generated sample x to the noise space because the encoder is the inverse process of the generator; it is easy to know $E(G(z))$ is still a standard normal distribution. But if the encoder $E(x)$ is used to map the training sample x to the noise space $p_\gamma(z)$, the resulting z is not necessarily a standard normal distribution, as shown in the top two distributions in Fig. 3.29.

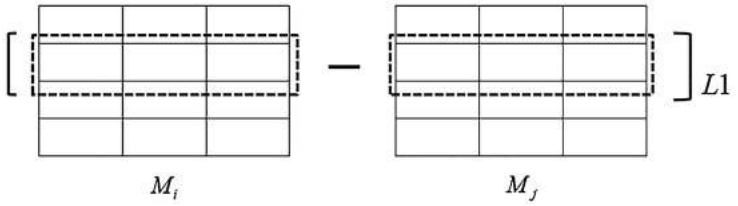


Fig. 3.29 Calculation of difference in mini-batch discriminator

Conversely, if the encoder $E(x)$ can map the training sample x to the standard normal distribution $p_\gamma(z) = N(0, I)$, then the mapping result $E(G(z))$ of the encoder $E(x)$ for the generated sample cannot be a standard normal distribution. Therefore, we can use the mismatch between the two as an indication of mode collapse, which is measured by the cross-entropy $H(z, z_\gamma)$ between $z \sim N(0, I)$ and $p_\gamma(z)$. Additionally, considering that the encoder must be trained sufficiently well (i.e., the reconstruction error is small), the overall objective functions of t generator $G(z)$ (parameterized by φ) and the encoder $E(x)$ (parameterized by γ) are:

$$\min_{\phi,\gamma} \mathbb{E}_z\left[\| z - E(G(z)) \|_2^2\right] + H(z, z_\gamma) \tag{3.87}$$

According to the Bayesian formula, $p_\gamma(z)$ is expanded as:

(3.88)

$$p_\gamma(z) = \int p_\gamma(z|x)p_{data}(x)dx$$

Where $p_\gamma(z|x)$ denotes the conditional distribution of z given x. The first reconstruction error term can also be expanded as

$$\int p(z) \int q_\gamma(x|z) \parallel z - E(x) \parallel^2 dxdz \tag{3.89}$$

The second cross-entropy term can be expanded as

$$H(z, z_\gamma) = -\int p(z) \log [p_\gamma(z)]dz = -\int p(z) \log \left[\int p_\gamma(z|x)p_{data}(x)dx\right]dz \tag{3.90}$$

This equation is not solvable because $p_{data}(x)$ is unknown. By introducing $p_\phi(x|z)$ and using the variational method, an upper bound on the cross-entropy term can be obtained as

$$\mathrm{D_{KL}}(p_\phi(x|z)p(z) \parallel p_\gamma(z|x)p_{data}(x)) - \mathbb{E}[\log p(z)] \tag{3.91}$$

We cannot minimize the original objective function, but can optimize the upper bound of the far objective function. Define $\mathrm{p}_\phi(x|z)$ as the generator $G(z)$ in GAN, and define $\mathrm{p}_\gamma(z|x)$ as the encoder $E(x)$ mentioned above. At the same time, introduce the discriminator $D_\theta(x, z)$ to estimate the logarithm of the ratio of $p_\phi(x|z)p(z)$ to $p_\gamma(z|x)p_{data}(x)$, so that address the calculation issue of the KL divergence in the variational upper bound. Specifically, $D_\theta(x, z)$ should be trained to reach the optimal $D^*_\theta(x, z)$, at which point the following should hold.

$$D^*_\theta(x, z) = \log \frac{p_\phi(x|z)p(z)}{p_\gamma(z|x)p_{data}(x)} \tag{3.92}$$

With the help of the discriminator, the KL divergence scatter can be estimated by sampling, and the $\mathbb{E}[\log p(z)]$ is a constant term need not be ignored. The objective function of the discriminator is inferred back from its optimal solution form, with the help of logistic regression, the objective function of discriminator D_θ is finally rewritten as

$$\min_\theta -\mathbb{E}_\phi[\log (\sigma(D(x, z)))] - \mathbb{E}_\gamma[\log (1 - \sigma(D(x, z)))] \tag{3.93}$$

Where $\mathbb{E}_\phi$ represents the expected value computed over the joint distribution of $p_\phi(x|z)p(z)$, while $\mathbb{E}_\gamma$ denotes the expected value computed over the joint distribution of $p_\gamma(z|x)p_{data}(x)$.

Now the basic training process of VEEGAN is as follows, in each iteration, sample N instances $\{z^{(1)}, ..., z^{(\mathrm{N})}\}$ from the noise $p(z)$, allowing the noise to pass through the generator $G(z)$ to obtain N generated samples $\left\{x_{\mathrm{G}}^{(1)}, \ldots, x_{\mathrm{G}}^{(\mathrm{N})}\right\}$. Then, sample N instances $\{x^{(1)}, ..., x^{(\mathrm{N})}\}$ from the training dataset, enabling the samples to pass through the encoder $E(x)$ to generate N encoded noises $\left\{z_E^{(1)}, \ldots, z_E^{(N)}\right\}$. The gradient is calculated from the empirical form of the discriminator's objective function:

$$g_\theta \leftarrow -\nabla_\theta \frac{1}{\mathrm{N}}\left[\sum_{i=1}^{N} \log \left(\sigma\left(D_\theta\left(z^{(i)}, x_G^{(i)}\right)\right)\right) + \log \left(1 - \sigma\left(D_\theta\left(z_E^{(i)}, x^{(\mathrm{i})}\right)\right)\right)\right] \tag{3.94}$$

Based on the objective function of encoder to calculate its gradient:

$$g_\gamma \leftarrow \nabla_\gamma \frac{1}{N}\left[\sum_{i=1}^{N} \parallel z^{(i)} - z_E^{(i)} \parallel_2^2\right] \tag{3.95}$$

Based on the objective function of generator to calculate its gradient:

$$g_\phi \leftarrow \nabla_\phi \frac{1}{N}\left[\sum_{i=1}^{N} D_\theta\left(z^{(i)}, x_G^{(i)}\right)\right] + \nabla_\phi \frac{1}{N}\left[\sum_{i=1}^{N} \parallel z^{(i)} - z_E^{(i)} \parallel_2^2\right] \tag{3.96}$$

Finally, using gradient descent is sufficient. Overall, VVEGAN is a relatively concise and intuitive improvement method that also alleviates the mode collapse problem to some extent.

3.6.6 Mini-Batch Discriminator

Mini-batch discriminator [6] believes that the cause of the mode collapse is still in the discriminator because the discriminator can only process one sample at a time independently, the gradient information obtained by the generator on each sample lacks "uniform coordination" and all point in the same direction, and there is no mechanism to require the output of the generator to be different from each other. There is no mechanism that requires the outputs of the generators to differ significantly from each other.

The solution given by the mini-batch discriminator is to have the discriminator no longer consider one sample independently, but a mini-batch of samples at the same time. The specific method is as follows: For each sample $\{x^{(1)}, x^{(2)}, ..., x^{(N)}\}$ in a small batch, the result of a certain intermediate layer L of the discriminator is extracted as an n-dimensional vector $f(x_i)$. This vector is then multiplied by a learnable tensor $T_{n \times p \times q}$, resulting in a $p \times q$ dimensional feature matrix M_i for sample x_i, which can be viewed as obtaining p features, each of dimension q.

Next, for each sample x_i, the sum of the differences between its r-th feature and those of other samples in the small batch is as follows:

$$o(x_i)_r = \sum_j \exp\left(- \parallel M_{i,r} - M_{j,r} \parallel_{L1}\right) \tag{3.97}$$

Among them, $M_{i,r}$ represents the r-th row of M_i, and the L1 norm is used to express the difference between the two vectors.

Then each sample will be calculated to obtain a corresponding vector:

$$o(x_i) = \left[o(x_i)_1, o(x_i)_2, \ldots, o(x_i)_p\right]^T \tag{3.98}$$

Finally, $o(x_i)$ is to be used as the next layer $L + 1$ of the intermediary layer drawn from the additional access, which means that a mini-batch layer has been added on the basis of the original discriminator. Its input is $f(x_i)$, its output is $o(x_i)$, and it also includes a learnable parameter T in between. While the original discriminator requires the probability that a sample originates from the training dataset, the task of the mini-batch discriminator is still to output the probability that a sample originates from the training dataset, but it is more capable because it can use other samples from the batch as additional information.

For the small-batch discriminator, when the generator undergoes mode collapse and needs to be updated, $G(z)$ first generates a batch of samples $\{G(z)^{(1)}, G(z)^{(2)}, ..., G(z)^{(N)}\}$. Since these samples are all within a single mode, the mini-batch output results $\{o(G(z)^{(1)}), o(G(z)^{(2)}), ..., o(G(z)^{(N)})\}$ will inherently differ significantly from the mini-batch output results computed from the training dataset. The captured difference information will not result in a substantially low value for the small-batch discriminator $D(G(z_i))$, and the small-batch discriminator will not simply output the same gradient direction for all samples.

In Progressive GAN [15], a simplified version of a mini-batch discriminator is given with the same idea as above, except that the computation is simpler, and for the input sample of the discriminator $\{x^{(1)}, x^{(2)}, ..., x^{(N)}\}$, a certain intermediate layer is extracted as the feature $\{f(x^{(1)}), f(x^{(2)}), ..., f(x^{(N)})\}$, and calculate the standard deviation of each dimension and find the mean value,

$$o = \frac{1}{N}\sum_{i=1}^{N}(\sigma_i) \tag{3.99}$$

Among them

$$\sigma_i = \sqrt{\frac{1}{N-1}\sum_{j=1}^{N}\left(f(x_j)_i - \widehat{f}_i\right)^2} \tag{3.100}$$

Finally, *o* is concatenated with the output of the intermediate layer as the feature map. Progressive GAN does not contain parameters that need to be learned, but directly calculates the statistical features of the batch samples, which is more concise.

References

1. Arjovsky, Martin, and Léon Bottou. "Towards principled methods for training generative adversarial networks." arXiv preprint arXiv:1701.04862 (2017).
2. Miyato, Takeru, et al. "Spectral Normalization for Generative Adversarial Networks." international Conference on Learning Representations. 2018.
3. Yoshida, Yuichi, and Takeru Miyato. "Spectral norm regularization for improving the generalizability of deep learning." arXiv preprint arXiv:1705.10941 (2017).
4. Mescheder, Lars, Sebastian Nowozin, and Andreas Geiger. "The numerics of GANs. "Proceedings of the 31st International Conference on Neural Information Processing Systems. 2017.
5. Nagarajan, Vaishnavh, and J. Zico Kolter. "Gradient descent GAN optimization is locally stable. "Proceedings of the 31st International Conference on Neural Information Processing Systems. 2017.
6. Salimans, Tim, et al. "Improved techniques for training gans." Advances in neural information processing systems 29 (2016): 2234-2242.
7. Müller, Rafael, Simon Kornblith, and Geoffrey E. Hinton. "When does label smoothing help?." Advances in Neural Information Processing Systems 32 (2019): 4694-4703.
8. Heusel, Martin, et al. "Gans trained by a two time-scale update rule converge to a local nash equilibrium." Advances in neural information processing systems 30 (2017).
9. Mescheder, Lars, Andreas Geiger, and Sebastian Nowozin. "Which training methods for GANs do actually converge?." International conference on machine learning. PMLR, 2018.
10. How to Train a GAN? Tips and tricks to make GANs work. https://github.com/soumith/ganhacks
11. Metz, Luke, et al. "Unrolled generative adversarial networks." arXiv preprint arXiv:1611.02163 (2016).
12. Kodali, Naveen, et al. "On convergence and stability of gans." arXiv preprint arXiv:1705.07215 (2017).
13. Ghosh, Arnab, et al. "Multi-agent diverse generative adversarial networks." Proceedings of the IEEE conference on computer vision and mode recognition. 2018.
14. Srivastava, Akash, et al. "Veegan: Reducing mode collapse in gans using implicit variational learning." Proceedings of the 31st International Conference on Neural Information Processing Systems. 2017.
15. Karras, Tero, et al. "Progressive Growing of GANs for Improved Quality, Stability, and Variation." International Conference on Learning Representations. 2018.

P. Long, X. Guo, *Generative Adversarial Network*
https://doi.org/10.1007/978-981-96-9404-4_4

4. Evaluation and Visualization of GAN

Peng Long[1] ✉ and Xiaozhou Guo[2]
(1) Beijing YouSan Educational Technology, Beijing, China
(2) • China Electronics Technology Group Corporation No. 54 Research Institute, Shijiazhuang, China

Abstract
This chapter presents an introduction to the evaluation indicators and visualization aspects of GANs. The evaluation indicators covered mainly include methods such as Inception Score (IS), Mode Score (MS), modified Inception Score (m-IS), Frechet Inception Distance (FID), Maximum Mean Discrepancy (MMD), Wasserstein distance, the nearest neighbor classifier, GANtrain and GANtest, Nearest Real Data Similarity (NRDS), Structural Similarity Index Measure (SSIM), Peak Signal-to-Noise Ratio (PSNR), and Sharpness Difference. Additionally, an open-source tool named GAN Lab is introduced. This tool provides a visual representation of the training process, data flow, and working principles of GANs.

Keywords IS – FID – GAN evaluation indicators – GAN lab

This chapter would introduce the evaluation indicators and visualization contents of GAN. The evaluation indicators mainly include IS series, FID, MMD, 1-Nearest Neighbor Classifier, and many other indicators; in order to enable relevant readers to grasp and learn the principles of GAN and conduct corresponding simple experiments, this chapter introduces GAN Lab, an open source tool.

- Section 4.1 Evaluation indicators
- Section 4.2 GAN Visualization

4.1 Evaluation Indicators

In discriminative models, the trained model is tested on a test set and then a quantifiable indicators is used to indicate how well the model is trained, e.g., most simply, the performance of the classification model is evaluated using classification accuracy and the performance of the regression model is evaluated using mean squared error. Similarly on generative models an evaluation indicators is needed to quantify the effectiveness of GAN generation.

4.1.1 Requirements of Evaluation Indicators

The indicator used to evaluate the merit of the generative model GAN cannot be arbitrary, and it should consider some requirements as much as possible. A few more important requirements are listed here: (1) models that generate more realistic samples should get better scores, i.e., the quality of sample generation can be evaluated (2) models that generate more diverse samples should get better scores, i.e., the GAN can be evaluated for overfitting, missing patterns, pattern collapse, simple memory (i.e., the GAN simply memorizes the training dataset),. and diversity. (3) For the hidden variables of GANz, if there is a clear "meaning" and the hidden space is continuous, then it is possible to control$z$. The GAN should be better evaluated if the desired sample is obtained. (4) Boundedness, i.e., the values of evaluation indicators should have clear upper and lower bounds. (5) GAN is usually used for image data generation, and some transformations of images do not change the semantic information (e.g., rotation), so the evaluation indicators should not be significantly different for images before and after some transformations. (6) The results given by the evaluation indicators should be consistent with human perception. (7) The computation of evaluation indicators should not require too many samples and should not have a large computational complexity. Considering the actual situation, these requirements often cannot be satisfied at the same time, and each different index has its own advantages and disadvantages.

4.1.2 IS Series

The Inception Score indicator is applicable to evaluate the GAN of generated images, and the evaluation indicator should first evaluate the quality of the GAN generated images, but the image quality is a very subjective concept. However, it is not easy for the computer to recognize this problem, and it is better to design a computable quantitative.

IS (Inception Score) [1] employs such an approach, where the generated images x are fed into an already trained Inception model, such as Inception Net-V3, which is a classifier that outputs a 1000-dimensional label vector y for

each input image, and each dimension of the vector represents the probability that the input sample belongs to a certain category. Assuming that our Inception Net-V3 is trained well enough, then for high-quality generated image x, Inception Net-V3 can classify them into a certain class with a high probability, i.e., the values of label vector $p(y|x)$ are more concentrated, shaped as [0.9,0..., 0.02,0]. We can use entropy to quantify this indicator:

$$H(y|x) = -\sum_i p(y_i|x) \log [p(y_i|x)] \tag{4.1}$$

Where $p(y_i|x)$ denotes the probability of x belonging to y, i.e., y_i. And in order to avoid ambiguity, the calculation is shown in Fig. 4.1:

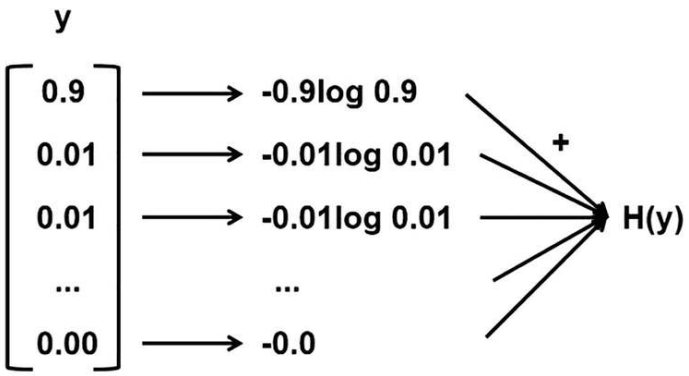


Fig. 4.1 Schematic diagram of entropy calculation

Entropy is a measure of the degree of confusion. For lower quality input images, the classifier cannot give a clear category, and its entropy should be relatively large, while for higher quality images, its entropy should be smaller. When $p(y|x)$ is one-hot distribution, its entropy reaches minimum 0. Another indicator considered by IS is the diversity of the samples. If the GAN generates a batch of samples with good diversity, then the category distribution of the label vector $\{y_1, y_2, ..., y_N\}$ should also be relatively uniform, that is, the probabilities of different categories are basically equal (of course, it is assumed that the categories of the training samples are balanced here), and the mean value should tend to be uniformly distributed as shown in Fig. 4.2.

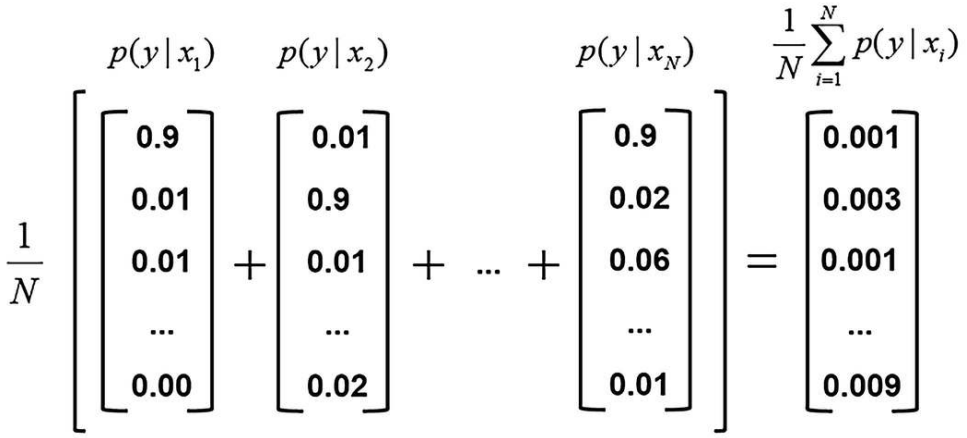


Fig. 4.2 Label vector distribution

Also because

$$\frac{1}{N}\sum_{i=1}^{N} p(y_i|x) \approx \mathbb{E}_x[p(y|x)] = p(y) \tag{4.2}$$

Therefore, the label vector *y* can be used to quantitatively describe the entropy concerning the categories; if the diversity of the generated samples is good (covering many categories), then the entropy where $p(y)$ relative to the categories would be greater; if the diversity of the generated samples is poor, then the entropy would be smaller. We define the entropy where $p(y)$ relative to the categories as:

$$H(y) = -\sum_{i=1}^{N} p(y_i)\ \log\ [p(y_i)] \tag{4.3}$$

Among them, $p(y_i)$ represents the probability of the *i*-th class, i.e., y_i. By considering both image quality and diversity, the mutual information $I(x; y)$ between the samples and labels can be designed as an evaluation indicator for generative models. Mutual information describes the degree of uncertainty reduction of one random variable given another, also known as information gain, defined as $I(x; y) = H(y) - H(y|x)$. Before knowing *x*, the marginal distribution $p(y)$ has relatively high entropy with respect to the class, indicating a greater degree of uncertainty for the label *y* (which may be close to a uniform distribution). When *x* is given, the conditional distribution $p(y \mid x)$ has reduced entropy relative to the class, which decreases the uncertainty of the label *y* (possibly approaching a one-hot distribution).

The greater the difference, the better the quality of the sample. According to the

$$\mathbb{E}_x[D_{KL}(p(y|x) \parallel p(y))] = H(y) - H(y|x) \tag{4.4}$$

The KL divergence represents the difference between two distributions, and when the KL divergence value is larger, it means the difference between two distributions is larger; the smaller the KL divergence value is, the smaller the difference between distributions is. The KL divergence of all samples is calculated to find the average, but in essence, it is still evaluated by information gain. To facilitate the calculation, an exponential term is added, and the final IS is defined in the following form:

$$\exp\left(\mathbb{E}_x D_{KL}(p(y|x) \parallel p(y))\right) \tag{4.5}$$

For the actual calculation of IS, the equation used is

$$\exp\left(\frac{1}{N}\sum_{i=1}^{N} D_{KL}\left(p\left(y\middle|x^{(i)}\right) \parallel \widehat{p}(y)\right)\right) \tag{4.6}$$

For $\widehat{p}(y)$ (the empirical distribution of $p(y)$), use the generative model to generate N samples, and feed the N samples into the classifier to obtain N label vectors, calculate the mean and make

$$\widehat{p}(y) \approx \frac{1}{N}\sum_{i=1}^{N} p\left(y^{(i)}\right) \tag{4.7}$$

For KL divergence, the calculation is as follows:

$$D_{KL}\left(p\left(y\middle|x^{(i)}\right) \parallel \widehat{p}(y)\right) = \sum_{j} p\left(y_j\middle|x^{(i)}\right) \log \frac{p\left(y_j\middle|x^{(i)}\right)}{\widehat{p}(y_j)} \tag{4.8}$$

IS, as an evaluation indicator for GAN, has gained a relatively wide acceptance since it was proposed in 2016, but there are some problems and drawbacks that cannot be ignored. (1) When GAN overfitting occurs, the generator only “remembers” the samples of the training set and the generalization performance is poor, but IS cannot detect this problem, and IS would still be high due to the good sample quality and diversity. (2) Since Inception Net-V3 is trained on ImageNet, IS would favor the object category in ImageNet

instead of focusing on realism, and the GAN generated image, no matter how realistic it is, would have a low IS as long as its category does not exist in ImageNet. (3) If the diversity of GAN-generated categories is sufficient, but the pattern collapse problem occurs within the class, IS cannot detect it. (4) IS only considers the distribution of generators p_g and ignores the distribution of the dataset p_{data}. (5) IS is a pseudo-metric. (6) The high or low IS is affected by the image pixels. These problems limit the generalization of IS. Next, we list several improved versions of IS.

MS(Mode Score) [2] is an improved version of IS that takes into account the labeling information of the training dataset, which is defined as

$$\exp\left(\mathbb{E}_x D_{KL}(p(y|x) \parallel p^*(y)) - D_{KL}(p(y) \parallel p^*(y))\right) \tag{4.9}$$

Where $p^*(y)$ denotes the class probability of the label vector obtained from the samples that have gone through the training dataset, and $p(y)$ denotes the class probability of the label vector obtained after the GAN generated samples, MS also considers the quality and diversity of the generated samples though it can be shown to be equivalent to IS.

The m-IS (Modified Inception Score) focuses on the problem of intra-class pattern collapse. For example, a well-trained GAN using ImageNet can generate 1000 classes of images evenly, but in each class, only one image can be generated, that is, the generated apple image would always Zhang a look, but the generation quality and class diversity of the GAN is m-IS computes the cross-entropy for the labels of samples in the same class:

$$-p(y|x_i)\,\log\,p(y|x_i) \tag{4.10}$$

Where x_i, x_j are samples with same class, whose class is determined by the output of Inception Net-V3. Considering the intra-class cross-entropy into IS yields m-IS, which is

$$\exp\left(\mathbb{E}_{x_i}\left[\mathbb{E}_{x_j}[D_{KL}(p(y|x_i) \parallel p(y|x_j))]\right]\right) \tag{4.11}$$

It can be seen that m-IS evaluates the quality of GAN generation and intra-class diversity. When the m-IS score is larger, the GAN generation performance is better.

The consideration of AMS (AM Score) is that IS assumes that the category labels have uniformity and the probability of generating 1000 classes by the generative model GAN is approximately equal, so we can use y to quantify this item. But when the data are not uniform in the category distribution, the IS

evaluation index is not reasonable, and a more reasonable choice is to calculate the KL divergence of the category label distribution of the training dataset and the category label distribution of the generated dataset, i.e.,

$$D_{KL}(p^*(y) \parallel p(y)) \tag{4.12}$$

Additionally, one term about the sample quality is kept constant, the expression of AMS is

$$D_{KL}(p^*(y) \parallel p(y)) + \mathbb{E}_x[H(y|x)] \tag{4.13}$$

Obviously, the GAN generation performance is better when the AMS score is smaller.

4.1.3 FID

FID (Fréchet Inception Distance) is a indicator for evaluating GANs [3], proposed in 2017, which is based on the idea that the samples generated by a generator and the samples generated by a discriminator are sent to a classifier (e.g., Inception Net-V3 or other CNNs), respectively, and the abstract features of the intermediate layer of the classifier are extracted. Assuming that this abstract feature fits a multivariate Gaussian distribution, estimate the mean μ_g and variance Σ_{g} of these generated samples and the mean μ_{data} and variance Σ_{data} of training samples, then calculate the Frechet distance of the two distributions, and this distance value is the FID:

$$\parallel \mu_{data} - \mu_g \parallel + \mathrm{tr}\left(\Sigma_{\mathrm{data}} + \Sigma_{\mathrm{g}} - 2((\Sigma_{\mathrm{data}}\Sigma_{\mathrm{g}})^{\frac{1}{2}}\right) \tag{4.14}$$

Finally, the FID is used as an evaluation metric. As shown in Fig. 4.3, the dashed part indicates the intermediate layer.

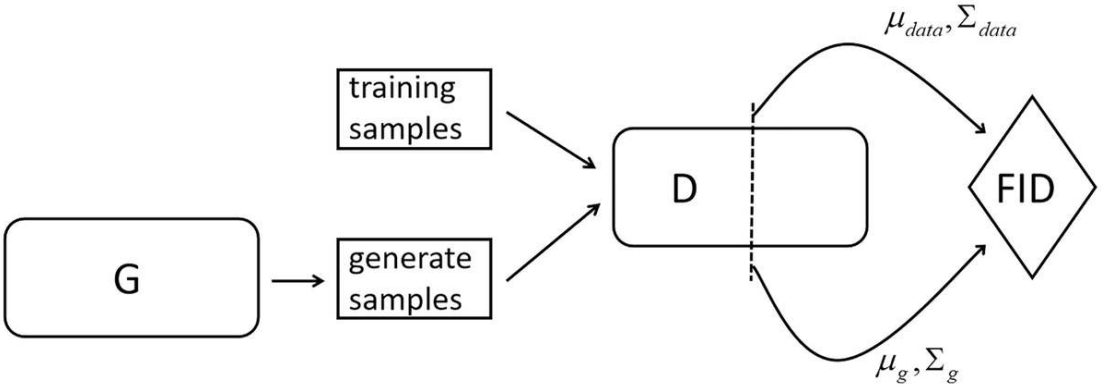


Fig. 4.3 Calculation of FID

The smaller the value of FID indicates that the closer the two Gaussian distributions are, the better the performance of GAN. In practice, it is found that FID has better robustness to noise and can have a better evaluation of the quality of the generated images, and the score it gives is more consistent with human visual judgment, and the computational complexity of FID is not high, although the first-order moments and second-order moments of the samples considered by FID only, but overall, FID is still more effective, and its theoretical shortcomings are: the simplification of Gaussian distribution assumptions do not hold in practice.

4.1.4 MMD

MMD (Maximum Mean Discrepancy) [4] has a very wide application in transfer learning, and it is a measure of the difference between two distributions in Hilbert space, so MMD can be considered to measure the distance between the training dataset distribution p_{data} and the generated dataset p_g, and then use this distance as the evaluation indicator of GAN. If the MMD distance is smaller, it means that p_{data} and p_g are closer, and the performance of the GAN is better.

To calculate the MMD, first choose a kernel function $k(x,y)$ which maps samples to a real number, such as a polynomial kernel function:

$$k(x,y)=\left(\gamma x^{\mathrm{T}} y+C\right)^{d} \tag{4.15}$$

Gaussian kernel function:

$$k(x,y)=\exp\left(-\parallel x-y \parallel^{2}\right) \tag{4.16}$$

The MMD distances are:

$$\mathbb{E}_{x,x' \sim p_{data}}\left[k\left(x,x'\right)\right]-2\mathbb{E}_{x\sim p_{data},y'\sim p_g}[k(x,y)]+\mathbb{E}_{y,y' \sim p_g}\left[k\left(y,y'\right)\right] \tag{4.17}$$

However, for the actual calculation, we cannot find the expectation, but need to use the samples to estimate the MMD value. For samples from the training set and generated, the estimated value of MMD is

$$\frac{1}{C_N^2}\sum_{i\neq i'}k\left(x^{(i)},x^{(i')}\right)-\frac{2}{C_N^2}\sum_{i\neq j}k\left(x^{(i)},x^{(i')}\right)+\frac{1}{C_N^2}\sum_{j\neq j'}k\left(y^{(j)},y^{(j')}\right) \tag{4.18}$$

Since MMD is estimated using samples, even though p_{data} and p_g are exactly the same, the estimated MMD may not be equal to zero.

4.1.5 Wasserstein Distance

Wasserstein distance, also known as earth-mover distance and bulldozer distance, is similar to MMD, which is also a measure of the difference between two distributions, so it can also be used as an evaluation indicator for GAN. If the Wasserstein distance is smaller, it means p_{data} and p_g are closer, the better the performance of the GAN. In the WGAN with superior performance, the Wasserstein distance of the two distributions is first learned by the discriminator (critic), and then the generator is trained with the objective function of minimizing the Wasserstein distance.

When using the Wasserstein distance as an evaluation indicator, it is necessary to first have a discriminator $D(x)$that has been trained well. For the samples from the training set and generated, the estimated value of Wasserstein distance is

$$\frac{1}{N}\sum_{i=1}^{N} D(x_i) - \frac{1}{N}\sum_{j=1}^{N} D(y_j) \tag{4.19}$$

This evaluation indicator can detect the simple memory and mode collapse of the generated samples and is relatively fast and easy to compute. However, it should be noted that since the use of Wasserstein distance as an evaluation indicator depends on the discriminator and the training dataset, it can only evaluate the performance of the GAN trained with a specific training set, e.g., it cannot evaluate the performance of the orange image generator for the discriminator (critic) trained with the apple image training set, so it also has some limitations.

4.1.6 1-Nearest Neighbor Classifier

The basic idea of the 1-Nearest Neighbor Classifier [5] is that it is desired to compute the decision whether p_{data} and p_g are equal. If they are equal, then the GAN is proved to be excellent, and if the difference is relatively large, the GAN is poor. The approach is as follows, for samples $x^{(1)}, x^{(2)}, \ldots, x^{(N)}$ drawn from the training sample set and samples $y^{(1)}, y^{(2)}, \ldots, y^{(N)}$ sampled from the generator probability distribution p_g, calculate the leave-one-out (LOO) accuracy using the 1-NN classifier, and use the accuracy as the evaluation indicator.

Specifically, combine $x^{(1)}, x^{(2)}, \ldots, x^{(N)}$ and $y^{(1)}, y^{(2)}, \ldots, y^{(N)}$ and their corresponding labels into a new sample set D, which contains a total of $2N$ samples. The samples in D are divided into two parts D_1 and D_2 using the leave-one-out cross-validation method, where D_1 has $2N - 1$ samples and D_2 has only one sample. The 1-NN binary classifier is trained using D_1 and validated in D_2 to calculate the correct rate (0% or 100%). Each time D_2 chooses a different sample, so the above process is looped $2N$ times. Calculate the overall correct classification rate, and use it as the evaluation index of GAN.

If distributions p_{data} and p_g are equal (i.e., $p_{data=}p_g$) and the sample size is relatively large, the 1-NN classifier cannot separate them well and the results are close to random guesses with an overall correct rate close to 50%, as shown in Fig. 4.4:

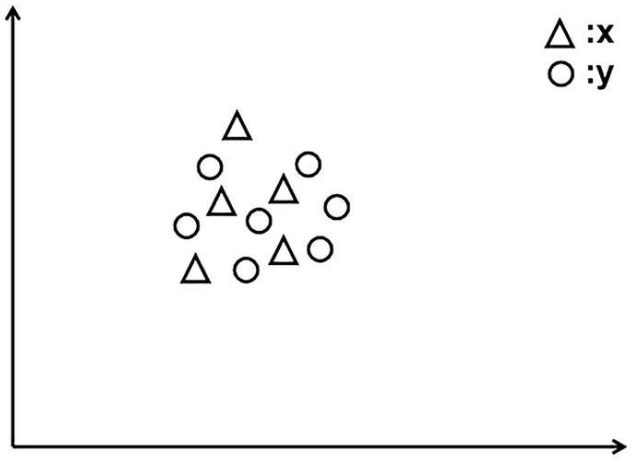


Fig. 4.4 Sample distribution when $p_{data=}p_g$

When the problem of simple memory occurs in the GAN, i.e., the generator generates exactly the same samples as the training samples, the overall correctness of any test sample on 1-NN is 0% because there exists a sample with distance 0 from the test sample but with opposite category labels, so the overall correctness is 0%, as shown in Fig. 4.5:

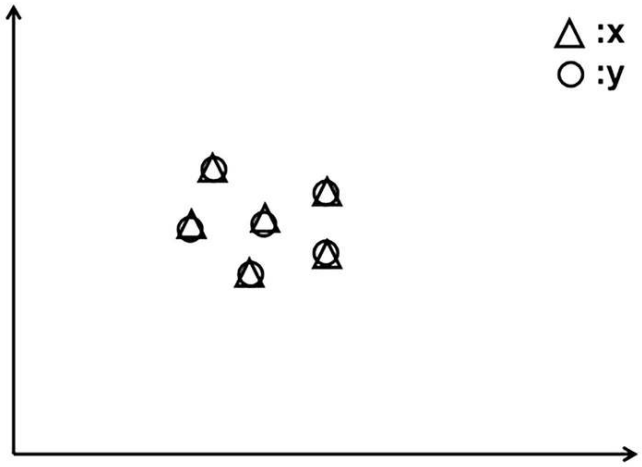


Fig. 4.5 Sample distribution at GAN simple memory

In the extreme case, when the generator-generated samples are very different from the training set samples, i.e., when the GAN generation is very poor, the overall accuracy is also 100% for any test sample on 1-NN since 1-NN is fully capable of accurate classification, as shown in Fig. 4.6:

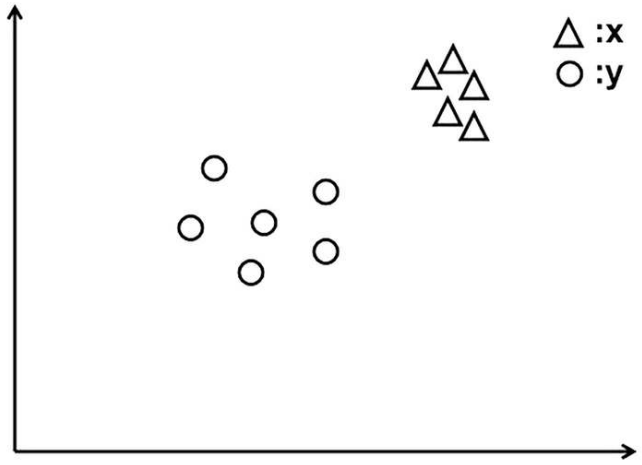


Fig. 4.6 Distribution when the samples are very different

When the total correct rate of 1-Nearest Neighbor Classifier is close to 50%, the better the performance of the generator is indicated. Also note that the reason for choosing 1-NN as a binary classifier here is that 1-NN is simple in structure, easy to compute and does not contain any hyperparameters.

4.1.7 GANtrain and GANtest

In GANtrain and GANtest [6], it is not designed to give quantifiable evaluation indicators, but to calculate several indicators and perform comparative analysis so as to evaluate the performance of the GAN, which is evaluated here

as a GAN that can generate multi-class samples. Define the training sample set S_t, the validation set S_v, and the sample set S_g generated by the GAN, and then train the classifier on the training set S_t, calculate the accuracy on the validation set S_v, and record the accuracy as GANbase; the classifier is trained on the generation set Sg and the accuracy is calculated on the validation set Sv, and the accuracy is denoted as GANtrain. The classifier is trained on the training set St, the accuracy is calculated on the generated set Sg, and the accuracy is denoted as GANtest.

Comparing GANbase and GANtrain, when there is a problem with GAN, GANtrain should be smaller than GANbase. This may be due to the loss of modes in the generated set S_g compared to the training set S_t, the generated samples not being realistic enough for the classifier to learn the relevant features, or the GAN not having clearly separated the categories, resulting in class mixing, etc. When GANtrain is close to GANbase, it indicates that the images generated by GAN are of high quality, exhibiting diversity similar to that of the training set.

Comparing GANbase and GANtest, ideally, the values of both should be close. If GANtest is very high, it suggests that the GAN is overfitting and the problem of simple memory has occurred. If GANtest is very low, it indicates that the GAN does not have a good data distribution and that the image quality is not high. The accuracy of GANtest measures the proximity between the generated images and the data manifold.

4.1.8 NRDS

NRDS (Normalized Relative Discriminative Score) [7] can be used for the comparison of multiple GAN models. The basic idea is that in practice, for the training dataset and the sample set generated by the GAN generator, it is always possible to train a classifier *C* that can completely separate the two classes of samples as long as a sufficient number of epochs are used. However, if the probability distributions of the two classes of samples are closer (i.e., the GAN generation effect is better), more epochs are needed to completely separate the two classes of samples; conversely, for the worse GAN generation effect, it is not necessary to train the classifier *C* as many times epochs to completely separate the two classes of samples.

As shown in Fig. 4.7, in each epoch, for *N* GANs, *N* batches of generated samples (fake samples) are obtained by sampling from them, and these are sent to the classifier *C* along with the training set samples (real samples) and the corresponding labels. The classifier is then tested separately on the *N* batches of fake samples, recording the output results of the *N* classifiers (the results should be the average of the batches). After training for a sufficient

number of epochs, the classifier should output almost 1 for real samples and almost 0 for fake samples. At this point, an N epoch-output curve is created for the N GANs, denoted as C_i, in order to estimate the area enclosed by the curve, as shown in Fig. 4.8:

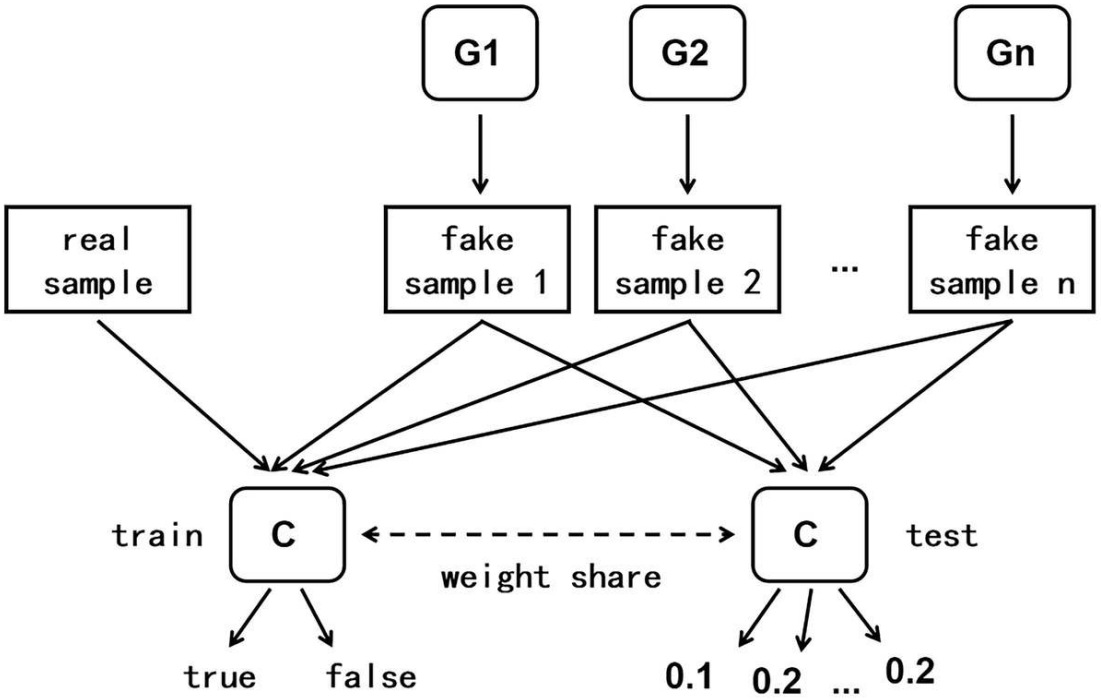


Fig. 4.7 $A(C_i)$ Calculation schematic of NRDS

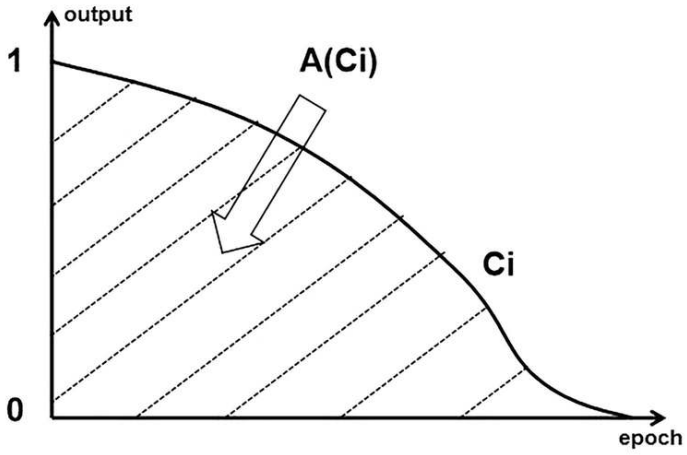


Fig. 4.8 Calculation schematic of $A(C_i)$

Denote the areas as $A(C_i)$, and finally calculate NRDS as follows:

(4.20)

$$\mathrm{NRD\,S}_i = \frac{A(C_i)}{\sum_{j=1}^{N} A(C_j)}$$

The larger the value of NRDS, the greater the “loss” in separating the two distributions completely, which indicates that p_g is closer to p_{data}.

4.1.9 Image Quality Measures

In this class of evaluation indicators, we directly quantify the quality of the image itself, unlike IS with the help of Inception V3 or training other neural networks, etc. Typical representatives here are SSIM, PSNR, and Sharpness Difference.

SSIM (Structural SIMilarity) measures the three aspects of luminance $l(x, y)$, contrast $c(x,y)$, and structure $s(x,y)$ between two image samples x and y. Three aspects are measured for comparison, which can be understood as an evaluation index describing the similarity of two images, where the brightness is

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \tag{4.21}$$

Contrast ratio is

$$c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \tag{4.22}$$

The structure is:

$$s(x,y) = \frac{2\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3} \tag{4.23}$$

Among them, μ_x, μ_y, σ_x, σ_y, σ_{xy} represent the local mean, variance, and covariance of x and y, respectively. C_1, C_2, C_3 are constants introduced to avoid division by zero, generally chosen as $C_1 = (k_1L)^2$, $C_2 = (k_2L)^2$, $C_3 = 0.5 \times C_2$, where k_1 is typically set to 0.01, k_2 to 0.03, and L denotes the range of pixel values. During the calculation, an $M \times N$ sized image block centered on either x or y can be taken sequentially from the image to compute the three parameters and solve.

$$\mathrm{SSIM}(x,y) = l(x,y)c(x,y)s(x,y) \tag{4.24}$$

The SSIM of the whole image is calculated by averaging the SSIM of each image block. SSIM has symmetry and the SSIM value reaches the maximum value of 1 when two images are identical.

PSNR (Peak Signal-to-Noise Ratio), which is the peak signal-to-noise ratio, is also used to evaluate the image quality. For example, in conditional GAN, the images in the training set in a certain category can be evaluated in comparison with the conditionally generated images, so as to evaluate the effect of conditional GAN generation. For example, for two images *I* and *K*, calculate their mean square error:

$$\mathrm{MSE}_{I,K} = \frac{1}{MN} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} [I(i,j) - K(i,j)]^2 \tag{4.25}$$

The peak signal-to-noise ratio is then calculated as

$$\mathrm{PSNR}_{I,K} = 10 \log_{10} \left(\frac{\mathrm{MAX}^2}{\mathrm{MSE}} \right) \tag{4.26}$$

Where MAX is the maximum possible pixel value of the image, e.g., 255 in grayscale images. in case of color images, the PSNR of the three channels of RGB can be calculated and then taken as the average value; or the three-channel MSE can be calculated and divided by 3, and then the PSNR can be calculated. Obviously, the larger the PSNR value is, the smaller the difference between the two images is, and the better the quality of the generated image is.

SD (Sharpness Difference) is calculated in a similar way to PSNR, but it is more concerned with the difference in sharpness information. For example, for two images *I* and *K*, the sharpness error is calculated as follows:

$$\mathrm{GRADS}_{I,K} = \frac{1}{N} \sum_{i} \sum_{j} |(\nabla_i I + \nabla_j I) - (\nabla_i K + \nabla_j K)| \tag{4.27}$$

Among them

$$\nabla_i I = \mid I_{i,j} - I_{i-1,j} \mid \tag{4.28}$$

$$\nabla_j I = \mid I_{i,j} - I_{i,j-1} \mid \tag{4.29}$$

SD was then calculated as:

$$\mathrm{SD}_{I,K} = 0 \log_{10} \left(\frac{\mathrm{MA\ X}^2}{\mathrm{GRADS}_{I,K}} \right) \tag{4.30}$$

Where MAX is the maximum possible pixel value of the image is the same as above. Obviously, the larger the SD value is, the smaller the difference in sharpness between the two images, the better the quality of the generated image.

4.1.10 Average Log-Likelihood

The previously mentioned methods, we all consider the generator as a black box that generates samples and does not directly deal with the probability density function p_g, which is also due to the design mechanism of GAN. However, it would be nice to have an p_g expression for the training set, the most straightforward evaluation indicator should be: calculate the logarithmic likelihood function of the samples in the training set under p_g (which can also be considered as calculating the KL divergence), and the larger the log-likelihood function is, the better the generator is, as follows:

$$\frac{1}{N} \sum_{i=1}^{N} \log p_g\left(x^{(i)}\right) \tag{4.31}$$

The problem here is how to get the expression or approximate the expression of p_g? One way is to use nonparametric estimation. For example, using the KDE (Kernel Density Estimation) method, for the samples $x^{(1)}$, $x^{(2)}$, ..., $x^{(N)}$, the probability density function p^* is:

$$p^*(x) = \frac{1}{Z} \sum_{i} K(x - x_i) \tag{4.32}$$

Z is the normalization constant, where the kernel function $K(\cdot)$ can be defined as a Gaussian kernel function, uniform kernel function, trigonometric kernel function, etc. freely chosen. Once the approximate probability density function is obtained, the log-likelihood can be calculated and used as an evaluation index. However, according to the actual situation, its evaluation effect is not satisfactory mainly with the following problems: facing the high-dimensional distribution, it is difficult to get a more accurate estimation of the probability density function for non-parameters, and in addition, there is no obvious

correlation between the log-likelihood function and the quality of the sample, GAN can give a high log-likelihood value but the quality of the sample is still very poor.

We have shown a wide variety of GAN evaluation indicators, and there are actually more, we have only shown some of them. According to the comparison results of the experiments, there does not exist any one evaluation indicator that can beat all other evaluation indicators in all aspects, nor does there exist any one indicator that can be well satisfied in all seven requirements, but there does exist some indicators whose quality completely surpasses another one. Therefore, when selecting the evaluation indicators of GAN, the indicators should be selected according to the actual scenario requirements, or several indicators should be selected to examine the effect of GAN generation from different perspectives.

4.2 GAN Visualization

GAN Lab [8] is an open source GAN visualization tool developed by Google. Using GAN Lab requires no installation process, no deep learning libraries such as PyTorch or TensorFlow, and no specialized hardware GPU. It can be opened through a web browser (Chrome is recommended) at https://poloclub.github.io/ganlab/

If you are interested in the source code, you can visit github to learn it yourself: https://github.com/poloclub/ganlab/github.com

Users can use GAN Lab to interactively train the generative model and visualize the intermediate results of the dynamic training process, using animation to understand every detail of the GAN training process, the screen is simple and beautiful; I think this is the best overall effect of the GAN visualization tool, the main interface is shown in Fig. 4.9:

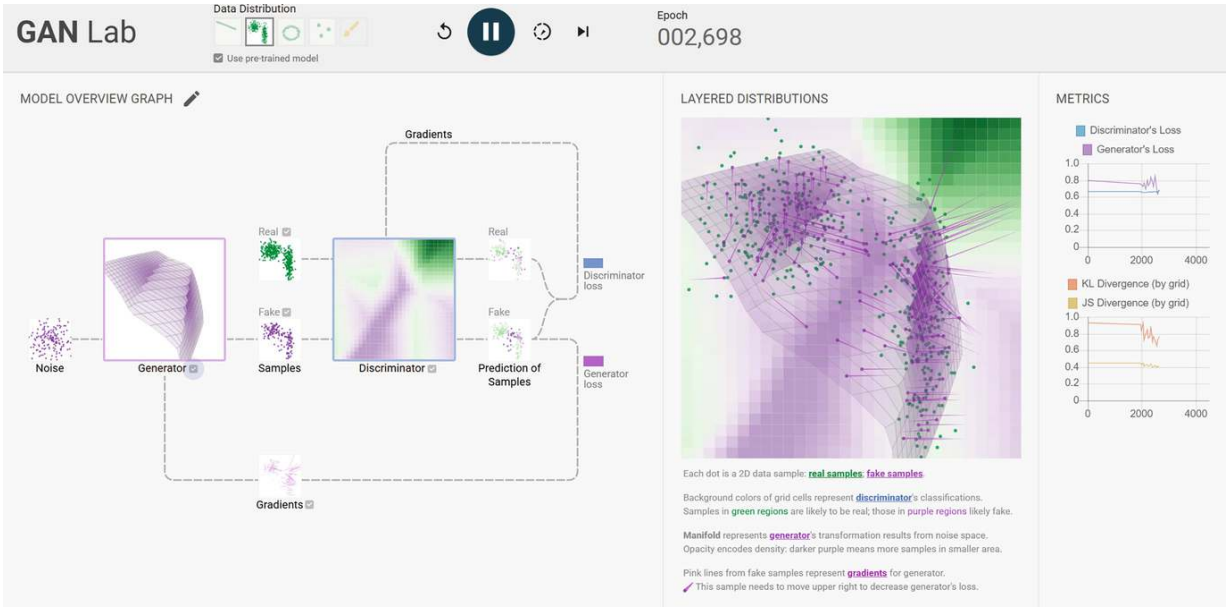


Fig. 4.9 Main body interface

The main body of GAN Lab includes three parts: MODEL OVERVIEW GRAPH, LAYERED DISTRIBUTIONS, and METRICS, where MODEL OVERVIEW GRAPH visualizes the GAN model as a picture, showing the basic structure of GAN, data flow, and visualizing the input and output data; LAYERED DISTRIBUTIONS visualizes the real samples, generator-generated samples, and generator gradient; LAYERED DISTRIBUTIONS visualizes the real samples, generator-generated samples, and generator gradients; METRICS records the metric of the distribution distance during iterative training.

4.2.1 Setting Up the Model

First, at the top of the interface, we can choose different data distributions. It should be emphasized that the visualization can only be shown in dimensions not exceeding 3D, and the data with dimensions exceeding 3D cannot be fully displayed, so the noise, training samples, real samples, and generation samples fake samples in the whole GAN Lab are all 2D data. Some experimental results about GAN may be related to data dimensionality, which cannot be reflected in GAN Lab. For the sake of screen simplicity, some adjustment buttons of model parameters are hidden, if you want to show them completely, please make sure to light up the edit button next to MODEL OVERVIEW GRAPH to yellow.

For the distribution of noise, 1D Gaussian distribution, 1D uniform distribution, 2D Gaussian distribution, 2D uniform distribution can be chosen, where 1D means that the sample only exhibits Gaussian/uniform distribution in one dimension and the other dimension is kept as fixed value. When the

mouse is placed over the Generator, GAN Lab shows the dynamic process of converting from noise space to generating data manifold, as shown in Fig. 4.10:

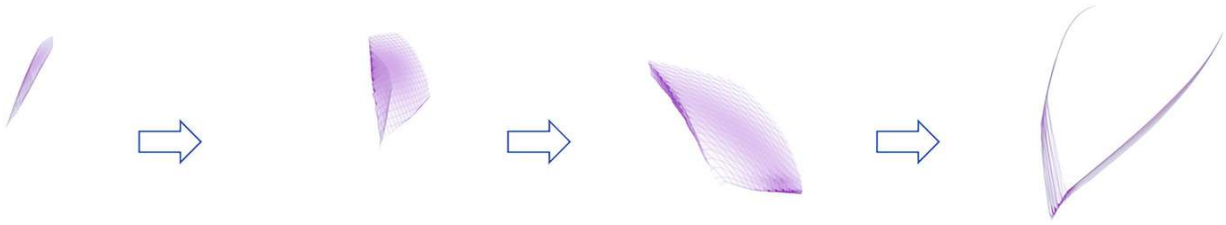


Fig. 4.10 Dynamic conversion process

For the distribution of training data, the four types built into GAN Lab can be selected, as shown in Fig. 4.11:

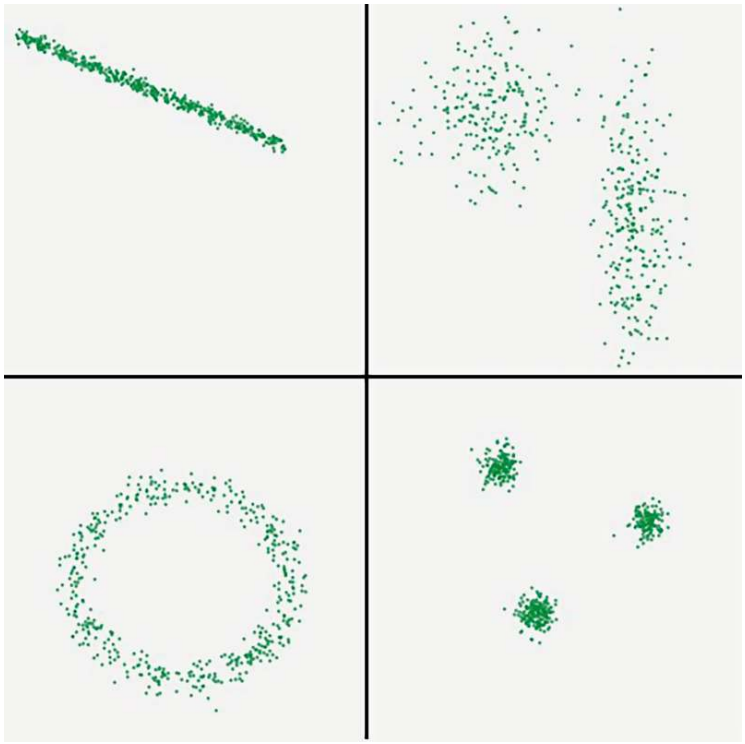


Fig. 4.11 Types of training data distribution

You can also use the drawing function to “draw” the desired training data distribution by selecting the fifth type of Draw one by yourself, sketching the data distribution in the whiteboard, and then clicking apply, as shown in Fig. 4.12:

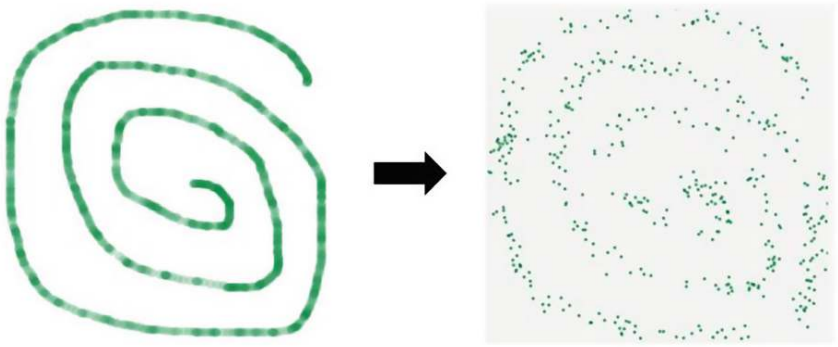


Fig. 4.12 Draw one by yourself distribution

The GAN Lab provides the simplest GAN, which only supports a single generator single discriminator structure, and both are fully connected layers, for which the generator Generator can set the number of hidden layers and the number of neurons within each hidden layer. For simplicity, you can use an already trained model by simply selecting use pre-trained model at the top of the page. For the loss function, GAN Lab provides Log loss and LeastSq loss, where the former is the objective function in the original version of GAN and the latter is the objective function in the least squares GAN. For the number of iterations of the generator and discriminator, you can set the number of times the generator/discriminator needs to be trained in each round of iterations, and manually adjust the updates per epoch. For the optimization algorithm, GAN Lab provides two algorithms, SGD and Adam, for both generator and discriminator, which can be selected in Optimizer.

4.2.2 Training Model

After setting the model structure, parameters and other information, you can control the training process of the model from the console at the top of the interface, as shown in Fig. 4.13:

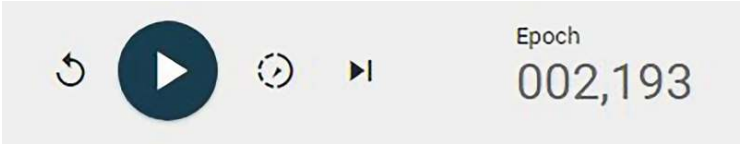


Fig. 4.13 Console

The first button Reset the model means that the model is completely reset and the parameters can be reset.

The second button Run/Pause training indicates the start/pause of the training process, and the training process visualization content is updated at different times while the data stream is displayed.

The third button Slow-motion mode indicates entering slow-motion mode, after lighting it in yellow, the operation flow of GAN can be shown in steps, in MODEL OVERVIEW GRAPH page, only the nodes and data flow involved in the current step would be shown explicitly, other parts are shown in vain, which helps to understand the GAN forward calculation and backward propagation The operational flow, as shown in Figs. 4.14 and 4.15 for the discriminator and generator in slow-motion mode, respectively:

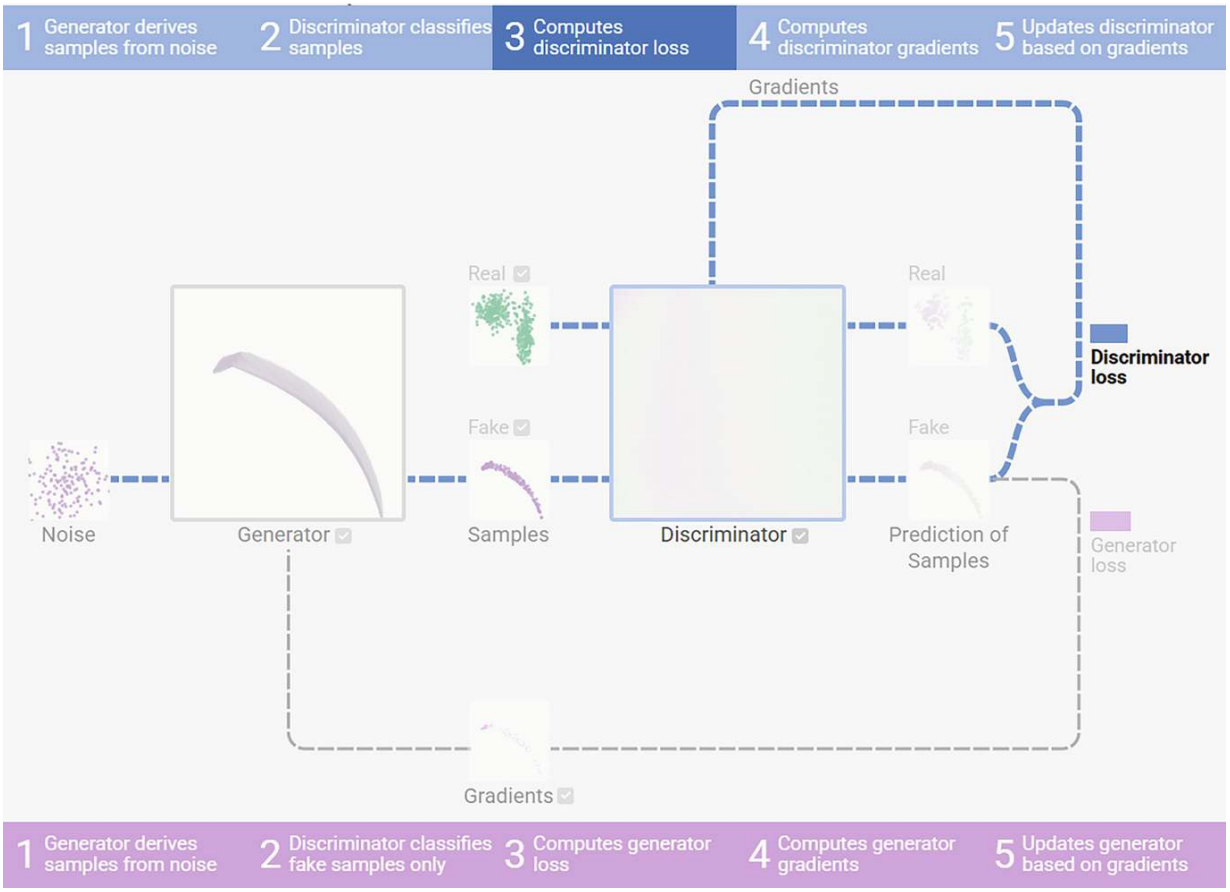


Fig. 4.14 Discriminator slow motion

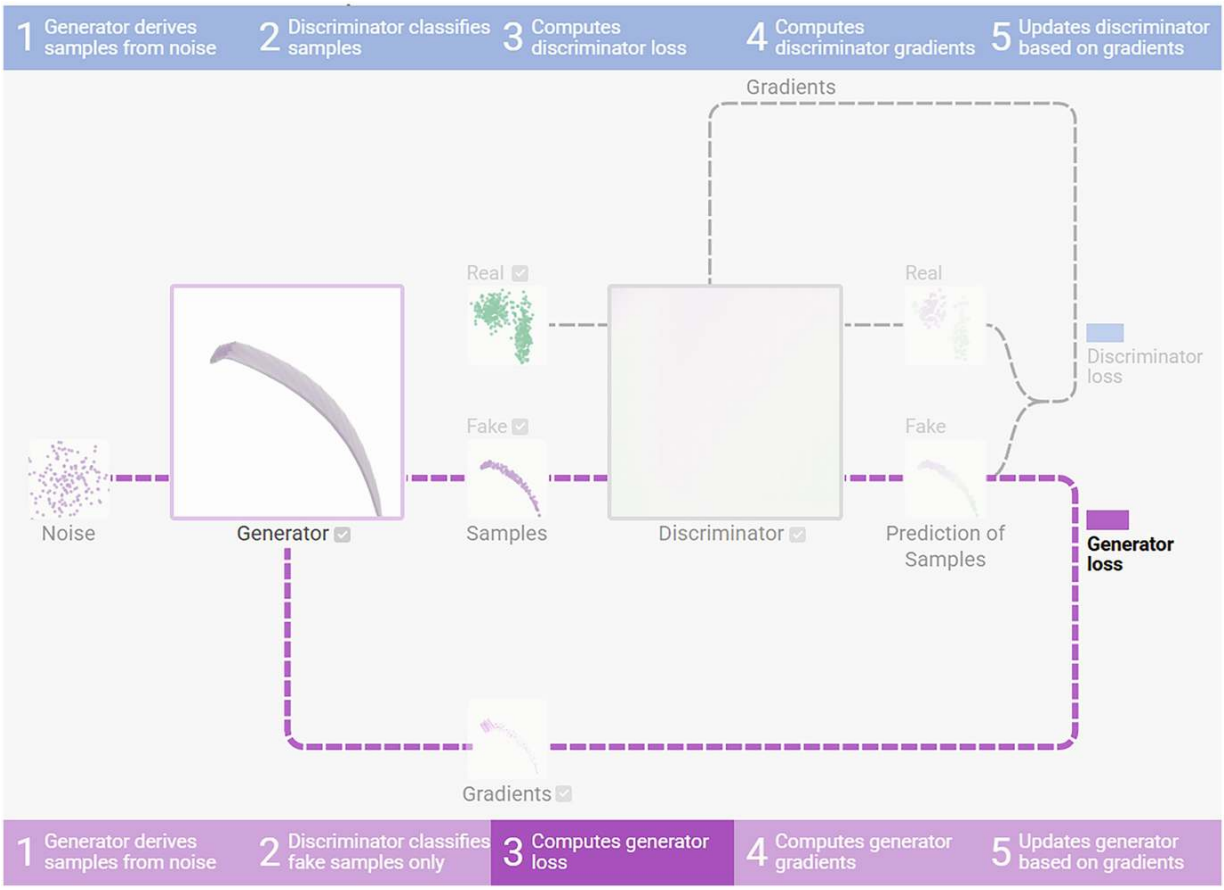


Fig. 4.15 Generator slow motion

The fourth button Train for one epoch can be used to control the rhythm of training, i.e., training only once. After lighting it in yellow, you can choose to train Generator only once, or Discriminator only once, or train both separately once Both, and each click would train one epoch.

The final epoch records the number of iterations of the model trained so far.

In the METRICS section, the losses of the generator and discriminator are recorded, along with the KL divergence and JS scatter of the two distributions (gridded calculations), as shown in Fig. 4.16. Note that the METRICS section is updated only once every 2000 epochs.

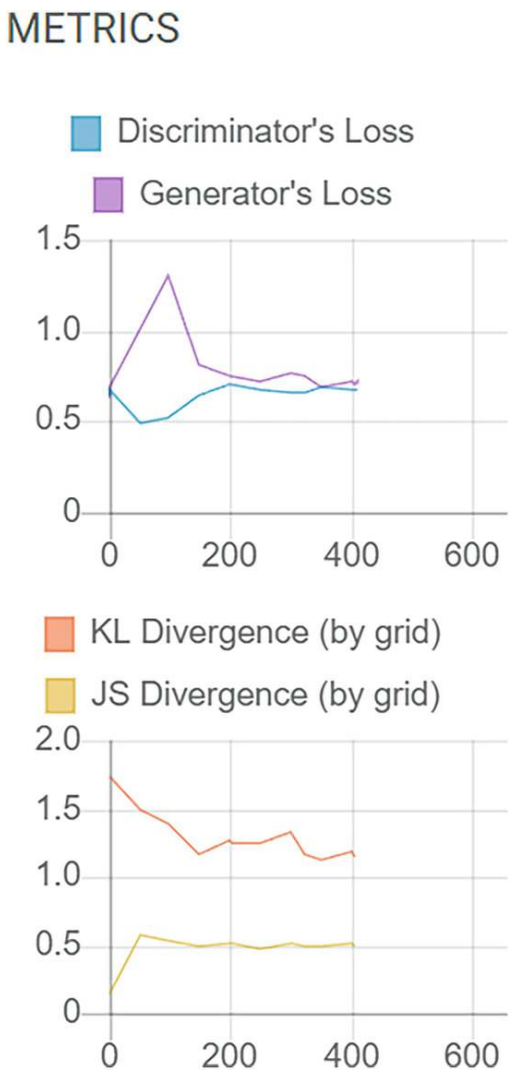


Fig. 4.16 METRICS interface

4.2.3 Visualization Data

In the MODEL OVERVIEW GRAPH module, each node is visualized. The distribution of the samples of the Noise node (represented in green), the distribution of the samples of the Real Samples node of the training dataset (represented in purple), and the distribution of the samples of the Fake Samples node of the generated data (represented in purple) are clearly displayed in the two-dimensional plane, as shown in Fig. 4.17:

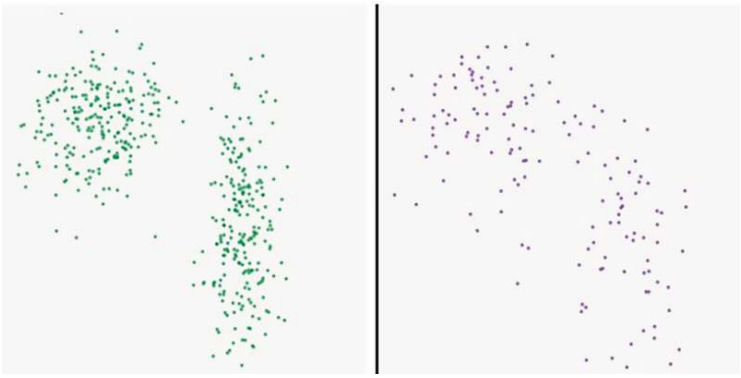


Fig. 4.17 Sample display

In the Generator node, the stream shape of the generated data is displayed visually. The purple part indicates the range of the data stream shape, and the shade of purple degree indicates the high probability of the data distribution, with dark purple indicating high probability and light purple indicating low probability, as shown in Fig. 4.18:

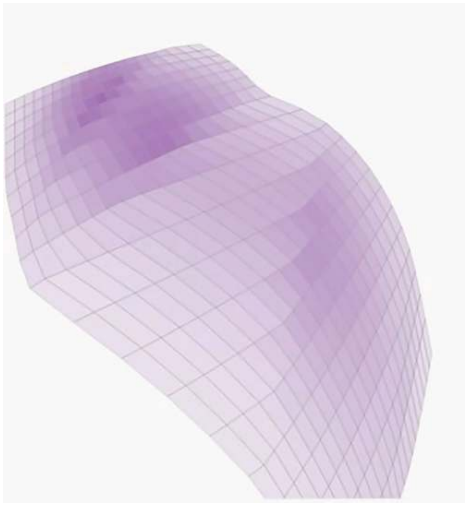


Fig. 4.18 Generating a data streamer

In the Discriminator node, the overall prediction result of the discriminator is visualized in the form of a heat map, where the green part indicates that the discriminator considers it as a true sample, and the darker the green color indicates that the discriminator output is closer to 1; the purple part indicates that the discriminator considers it as a false sample, and the darker the purple color indicates that the discriminator output is closer to 0; the white part indicates that the discriminator output is close to the white part indicates that the discriminator output is close to 0.5, which can also be interpreted as the classification surface of the classifier, as shown in Fig. 4.19:

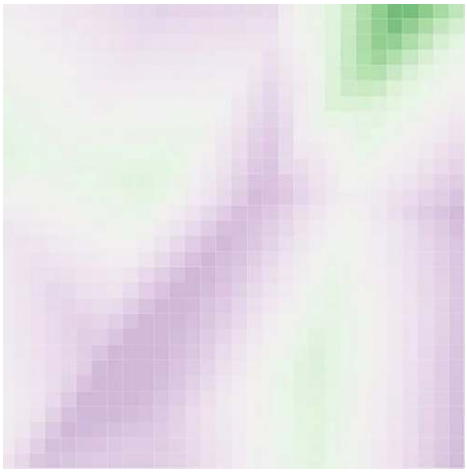


Fig. 4.19 Heat map of the discriminator

In the prediction of samples node of the discriminator, the output results of each real sample Real Samples and false sample Fake Samples after the discriminator are shown with the same color meaning as above.

In the Gradients node of the generator, the visualization results show the false samples, i.e., the generated samples, and the gradients together, indicating the direction of the gradient computed for each false sample with a straight line segment, and the length of the line segment indicates the magnitude of the gradient, as shown in Fig. 4.20:

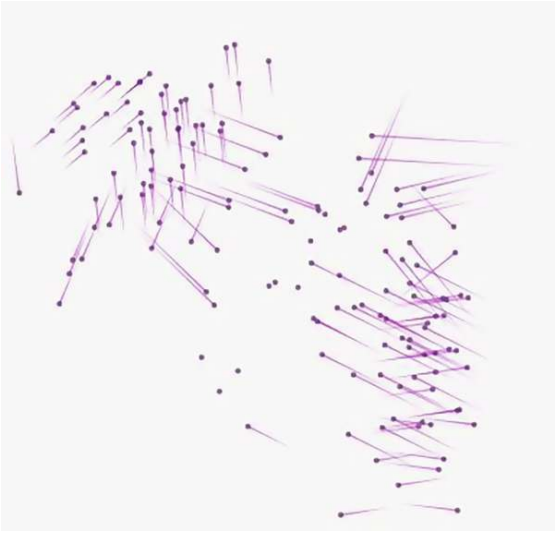


Fig. 4.20 Schematic representation of the sample gradient

LAYERED DISTRIBUTIONS presents the five nodes of real sample, false sample, stream shape of false sample, result graph of discriminator, and gradient of false sample together in the same graph, as shown in Fig. 4.21:

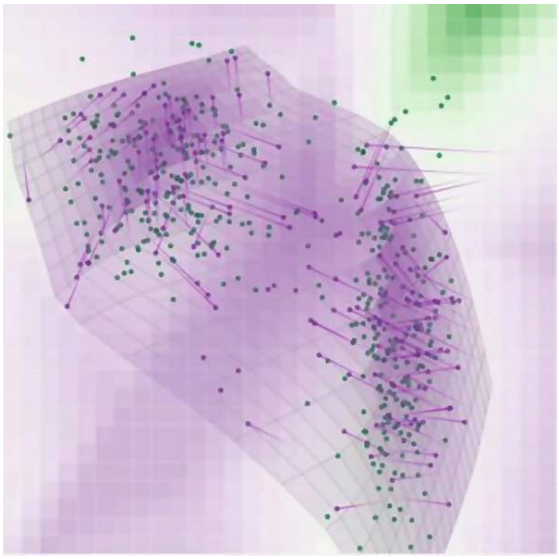


Fig. 4.21 Layered distributions interface

You can adjust the contents displayed in the LAYERED DISTRIBUTIONS module to selectively display, just click on the words real samples, fake samples, discriminator, generator, and gradients in the introduction below, and the solid lines under the words indicate that the nodes' The solid line under the word means that the visualization content of the node would be displayed in LAYERED DISTRIBUTIONS, while the dotted line would not be displayed, as shown in Fig. 4.22:

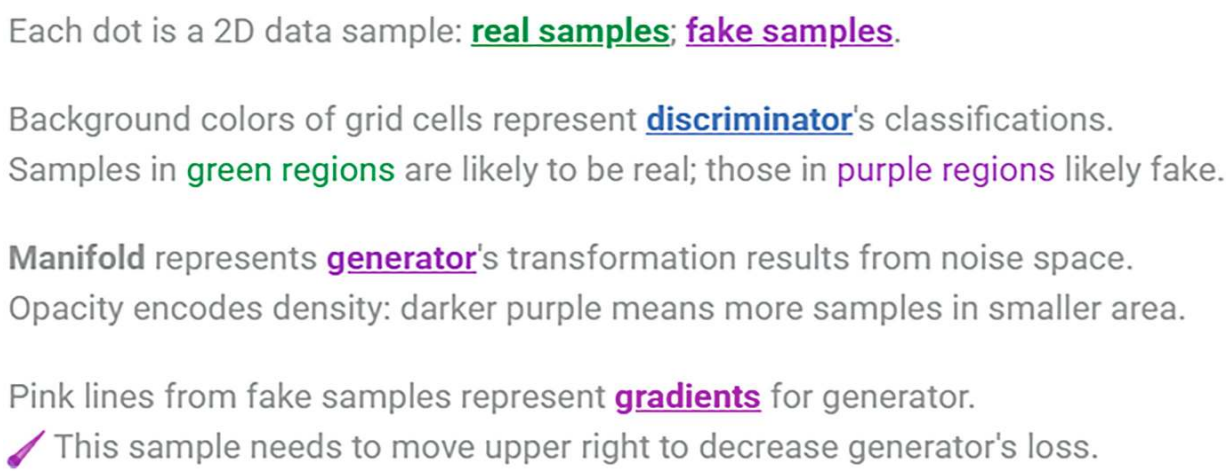


Fig. 4.22 Module display selection

4.2.4 Two Demos

Let's take an example to see how to understand the workflow of GAN through GAN Lab. First, the training generator brings the false sample (purple) closer to the real sample (green), and the gradient of the false sample also indicates

that training brings the two distributions closer together, as shown in Fig. 4.23:

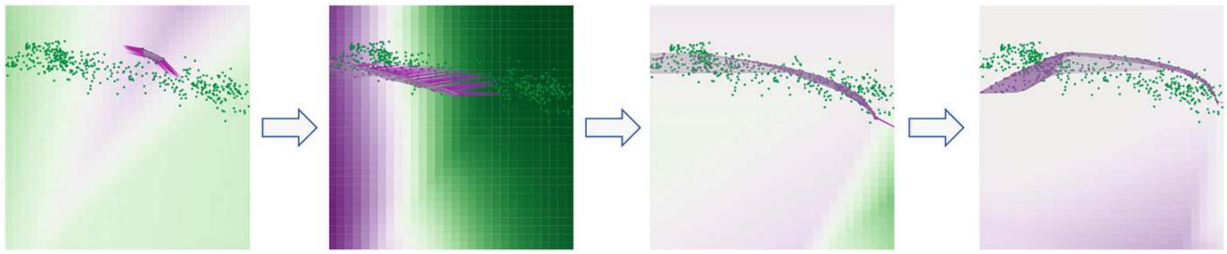


Fig. 4.23 Generator training process

Next, the discriminator is trained, and the discriminator does not have an effect on the distribution of the samples, but does have an effect on the output heat map, as shown in Fig. 4.24:

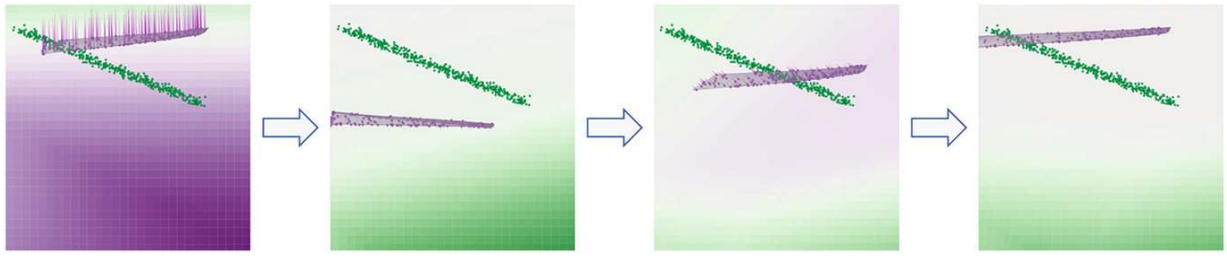


Fig. 4.24 Discriminator training process

The updates are continuously iterated, and finally the true and false samples almost overlap and the discriminator outputs 0.5 (white) at these sample points, as shown in Fig. 4.25:

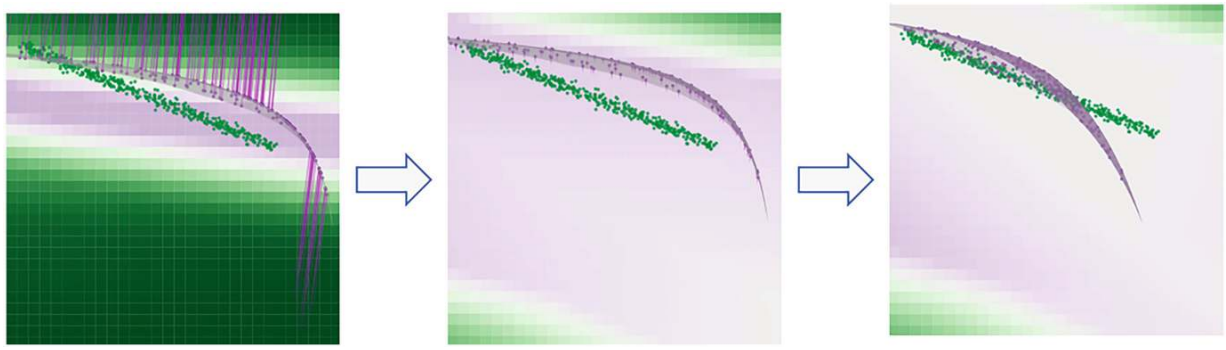


Fig. 4.25 Iterative process

Alternatively, we can analyze the data samples to understand the pattern collapse problem, as shown in Figure 4.26, where all generated spurious

samples are clustered to a single point and the generator does not fit the distribution of the real samples at all.

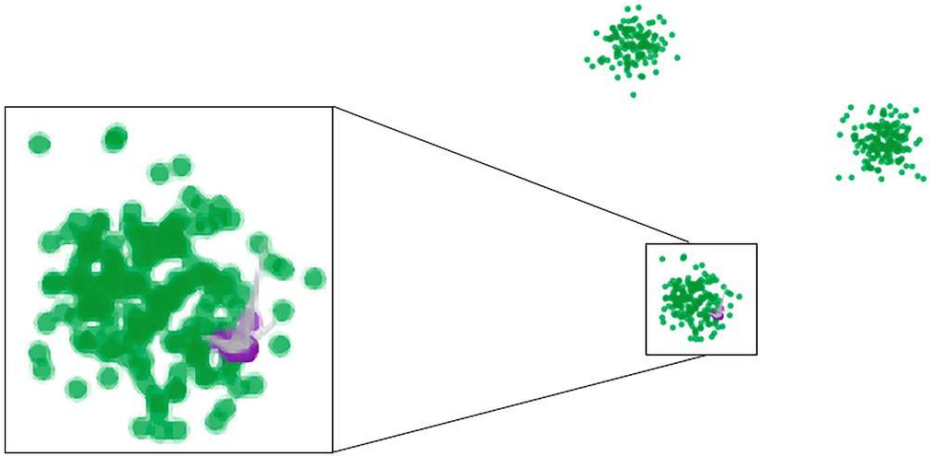


Fig. 4.26 Pattern Crash Check

GAN Lab is a very good GAN visualization software known so far, simple and vivid, suitable for introduction, but for more difficult problems, it is not yet able to do complete visualization due to various limitations.

References

1. Salimans, Tim, et al. "Improved techniques for training gans." Advances in neural information processing systems 29 (2016): 2234.2242.

2. Che, Tong, et al. "Mode regularized generative adversarial networks." 5th International Conference on Learning Representations, ICLR 2017. 2019.

3. Heusel, Martin, et al. "Gans trained by a two time-scale update rule converge to a local nash equilibrium." Advances in neural information processing systems 30 (2017).

4. Gretton, Arthur, et al. "A Kernel Method for the Two-Sample Problem." Journal of Machine Learning Research 1 (2008): 1-10.

5. Lopez-Paz, David, and Maxime Oquab. "Revisiting classifier two-sample tests. "International Conference on Learning Representations. 2017.

6. Shmelkov, Konstantin, Cordelia Schmid, and Karteek Alahari. "How good is my GAN?." Proceedings of the European Conference on Computer Vision (ECCV). 2018.

7. Zhang Z, Song Y, Qi H. Decoupled learning for conditional adversarial networks [C]//2018 IEEE Winter Conference on Applications of Computer Vision (WACV). IEEE, 2018: 700-708.

8. Kahng M, Thorat N, Chau D H, et al. Gan lab: understanding complex deep generative models using interactive visual experimentation [J]. IEEE transactions on visualization and computer graphics, 2018, 25(1): 310-320.
[Crossref]

P. Long, X. Guo, *Generative Adversarial Network*
https://doi.org/10.1007/978-981-96-9404-4_5

5. Image Generation

Peng Long[1] ✉ and Xiaozhou Guo[2]
(1) Beijing YouSan Educational Technology, Beijing, China
(2) • China Electronics Technology Group Corporation No. 54 Research Institute, Shijiazhuang, China

Abstract
This chapter comprehensively explores the advancements and methodologies in image generation using Generative Adversarial Network. It begins by outlining key applications of GANs, including training data expansion, synthetic data refinement, and creative content generation, highlighting their role in reducing manual data collection efforts and enhancing model generalization. The chapter then delves into foundational frameworks such as Deep Convolutional GAN (DCGAN), detailing its architecture, transposed convolution mechanisms, and limitations in generating low-resolution images.

Subsequent sections focus on conditional GAN variants (CGAN, InfoGAN, ACGAN), which integrate explicit or implicit control over generated outputs through labeled or latent variables. Multi-scale GAN architectures, including LAPGAN and Progressive GAN, are introduced to address high-resolution image synthesis challenges via hierarchical residual learning. StyleGAN, a landmark model for attribute disentanglement and high-fidelity face generation, is analyzed in depth, emphasizing its mapping network, adaptive instance normalization (AdaIN), and style-mixing techniques.

The chapter also covers advanced topics such as multi-discriminator and multi-generator frameworks to mitigate mode collapse, as well as GAN applications in data augmentation (BAGAN) and simulation refinement (SimGAN). Finally, practical implementations of DCGAN and StyleGAN are demonstrated, including code interpretation and training details.

Keywords Generative adversarial networks (GAN) – Image generation – DCGAN – Conditional GAN – Multi-scale GAN – StyleGAN – Data augmentation

---

Image generation is the first widely studied task in the application area of generative adversarial networks. With the continuous development of image generation frameworks in recent years, current images generated by GAN can already achieve the effect of faking, even making some countries have to legislate to restrain the

dissemination of fake images, and in this chapter we summarize the core techniques of image generation GAN.

5.1 Image Generation Applications

In this section, we will first introduce applications related to image generation. Using generative models, we can greatly reduce the cost of manually collecting high-quality data and even create images with artistic value.

5.1.1 Training Data Expansion

Data is a crucial aspect in machine learning-related tasks, and imbalance in datasets is a widespread phenomenon. Data imbalance can cause models to learn results that are biased toward classes with more samples, thus reducing the generalization ability of the model.

Although there are very many data enhancement methods that can generate pattern-rich images, most of them are local modifications of existing images, while GAN, as an excellent generation framework, can generate high-quality image data from scratch. Through data generation, we can expand the dataset and thus train models with better generalization ability.

The current research related to image generation in the field of face is the most mature, and the mainstream framework represented by StyleGAN [1] can generate high-definition face images with 1024 resolution, and Fig. 5.1 shows some of the generated high-quality portrait images.

Fig. 5.1 Face image generation

Training GAN for image generation itself requires a relatively large dataset, which is a "chicken and egg" versus "egg and chicken" problem. In some fields where the number of training samples is very small, such as industrial defect detection field, medical field, researchers are continuing to study the few-shot sample generation framework for solving the problem of sample generation with small training data, which is a relatively cutting-edge and immature area in the current image generation field.

5.1.2 Data Quality Improvement

We often use simulated data to train models, partly because simulated data can expand the capacity of the dataset, and partly because many data collection processes are difficult or costly. However, there are often large differences between the simulated data and the real data, resulting in the generalization ability of the trained model being affected, for example, there are large differences between the environment simulated by the autonomous driving simulator and the real road conditions.

Therefore, some researchers have started to use GAN to improve the quality of simulation data. Figure 5.2 shows the effect of SimGAN [2], proposed by researchers at Apple, to improve the simulated eye picture (Synthetic) to get a more realistic picture (Refined).

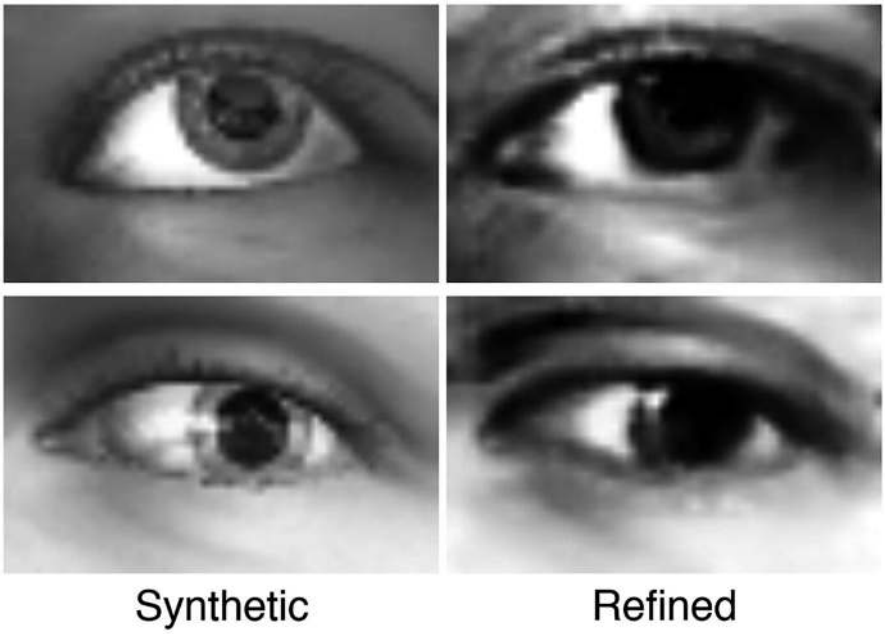


Fig. 5.2 SimGAN improves the realism of simulated eye pictures

5.1.3 Content Creation

GAN models can be used both to assist in generating training datasets and can be deployed directly as industrial-grade products to reduce labor costs in related fields,

typically in design [3, 4]. The design industry has always been a challenging and demanding field, requiring creators with solid design skills and artistic inspiration to obtain potentially popular solutions from millions of design jobs, such as FaceBook researchers used GAN for clothing design.

Figure 5.3 shows some of GAN's design work of which the work on the far right, titled Edmond De Belamy, fetched $430,000 at auction.

Fig. 5.3 Design work

For the field of art and design, however, the evaluation of models faces some challenges. For example, how to assess the creativity of the results, i.e., the high artistic value, and how to control the color and texture details.

5.2 Deep Convolutional GAN

Generative adversarial network GAN in the usual sense refers to a network with input noise vectors and output real images, which contains discriminators and generators. The generator is used for image generation, and the discriminator is used to discriminate the real image from the generated image. Figure 5.4 shows a schematic structure of an image generation GAN.

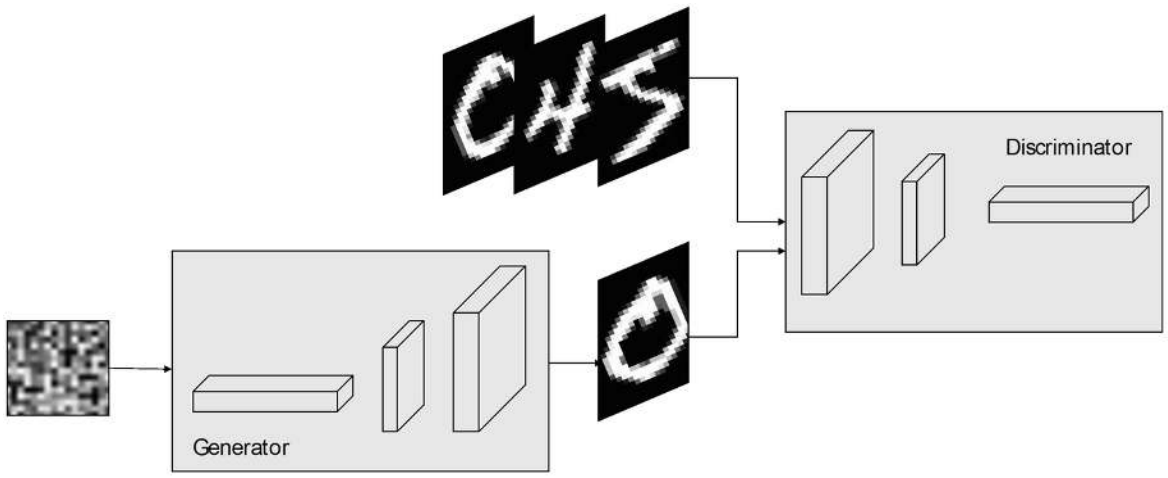


Fig. 5.4 The basic structure of image generation GAN

Generator inputs the noise and outputs the resulting image. The noise is usually a one-dimensional vector that is deformed (reshape) into a two-dimensional feature map, and then several deconvolution layers are used to learn the upsampling. Whereas the discriminator does not differ from the basic image classifier in terms of model structure, we next describe the specific implementation of a full convolution GAN.

5.2.1 Deep Convolutional GAN

DCGAN [5] is an early deep fully convolutional image generation GAN, which can be seen as a generic name for a series of image generators.

The input to the DCGAN generator is a 1 × 100 vector, which is then learned by a fully connected layer, deformed into a 4 × 4 × 1024 tensor, and then upsampled by four deconvolutional networks with a multiplicity of 2 to generate a 64 × 64 image, and the generator structure is shown in Fig. 5.5:

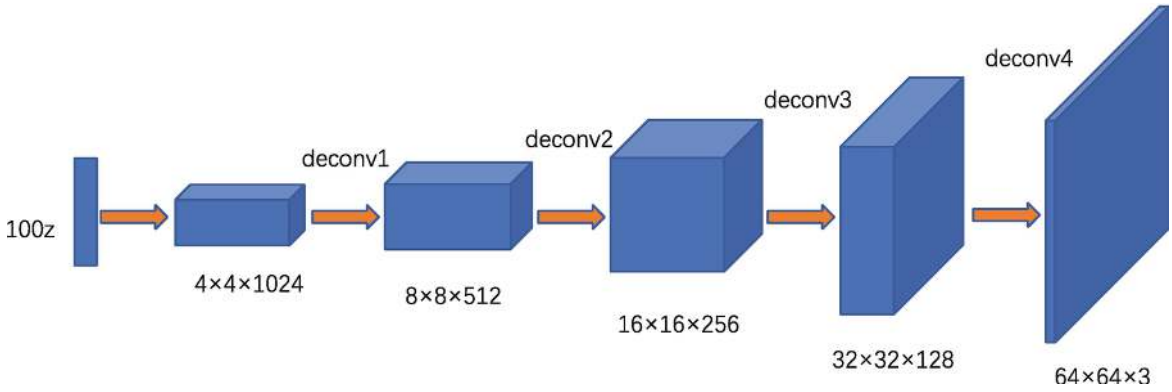


Fig. 5.5 Schematic diagram of the generator structure of DCGAN

The specific configuration of the generator is shown in Table 5.1.

Table 5.1 Configuration of each convolutional layer of the generator

Generator convolution layer	Input/output feature resolution	Number of input/output feature channels	Convolution kernel size
deconv1	4 × 4/8 × 8	1024/512	5 × 5
deconv2	8 × 8/16 × 16	512/256	5 × 5
deconv3	16 × 16/32 × 32	256/128	5 × 5
deconv4	32 × 32/64 × 64	128/3	5 × 5

In the original implementation of DCGAN, upsampling was performed using fractional stride convolution, which is now generally referred to as transposed convolution, and is schematically illustrated in Fig. 5.6:

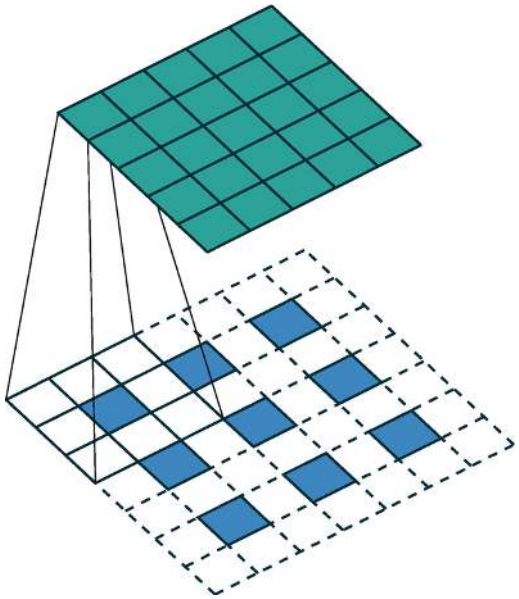


Fig. 5.6 Schematic diagram of fractional stride convolution

In Fig. 5.6, the input matrix block is shown at the bottom and the output matrix block is shown at the top. When the convolution is performed, a null is inserted in the middle of the adjacent elements in the input matrix block, which produces the effect of a stride size of 1/2, and is therefore called fractional stride convolution.

The input of the discriminator is a 64 × 64 size graph, which can be used with a resolution and channel transformation strategy symmetric to the generator, and the resolution is finally reduced to a 4 × 4 size after four convolutions with a stride size of 2. The schematic diagram is shown in Fig. 5.7.

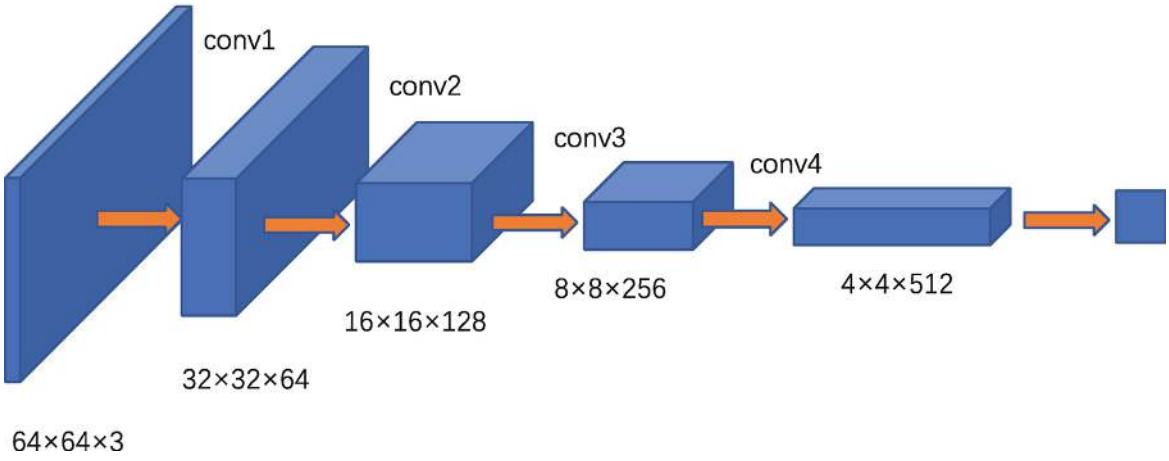


Fig. 5.7 Schematic diagram of the discriminator structure of DCGAN

The convolutional layer configuration of the discriminator in Fig. 5.7 is shown in Table 5.2.

Table 5.2 Configuration of each convolutional layer of the discriminator

Discriminator convolution layer	Input/output feature resolution	Number of input and output channels	Convolution kernel size
conv1	64 × 64/32 × 32	3/64	5 × 5
conv2	32 × 32/16 × 16	64/128	5 × 5
conv3	16 × 16/8 × 8	128/256	5 × 5
conv4	8 × 8/4 × 4	256/512	5 × 5

DCGAN model is relatively small and simple in principle and can generate some pictures with simple texture types, such as Fig. 5.8 shows the generated handwritten numbers, and the generated numbers have good authenticity, but the picture resolution is low, only 32 × 32 size.

Fig. 5.8 Handwritten digits generated by DCGAN

5.2.2 Thinking About DCGAN

DCGAN is not only a basic image generation framework, but the authors explore more experiments beyond the basic image generation tasks in their paper on DCGAN.

The authors used the features learned by the discriminator for image classification and found that they could obtain comparable classification results with mainstream SVM and CNN models, verifying the semantic discriminative ability of the discriminator.

The authors analyze in some detail the effect of the noise vector z on the semantic properties of the generated image results, assuming that *z* is a vector from the Latent Space. By controlling the vector z, it is possible to change the specific semantic content of the generated image, as well as to edit the semantic content, such as changing the expression and pose of the face.

Figure 5.9 shows the results from the semantic attribute editing in the paper.

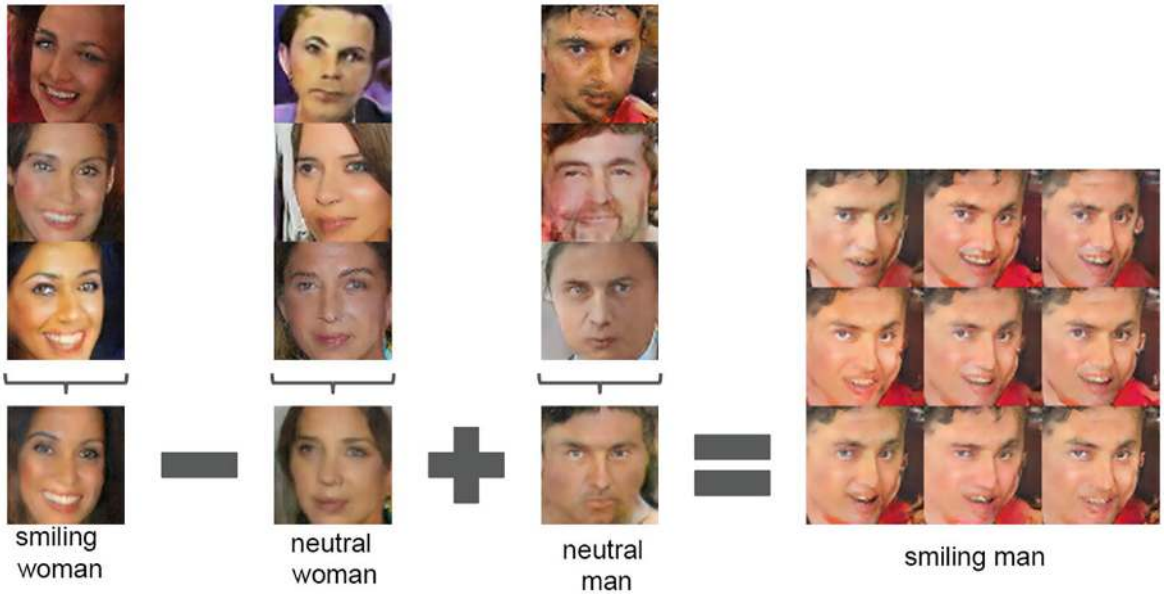


Fig. 5.9 DCGAN face attribute editing results [5]

where the images smiling woman, neutral woman, and neutral man are all generated sets of images with related properties, and the corresponding average noise vectors are represented by Vector("smiling woman"), Vector("neutral woman"), and Vector("neutral man").

By Vector("smiling woman")-Vector("neutral woman")+Vector("neutral man ") to get the new Vector, which is then fed into the generator to generate the face with the smiling man attribute. This shows that to some extent the gender and smiling expression attributes can be edited linearly by the noise vector Z.

Although Fig. 5.9 shows that attribute editing can be performed by the noise vector Z, the quality of the generated images is very low, and it has not been verified on more attributes that complex image attributes can be edited very effectively by the noise vector Z alone because the correspondence between the original noise vector Z and the generated result G(Z) is very complex and confusing.

Figure 5.10 shows a schematic representation of the distribution of the simple two-dimensional vector z with the generated results.

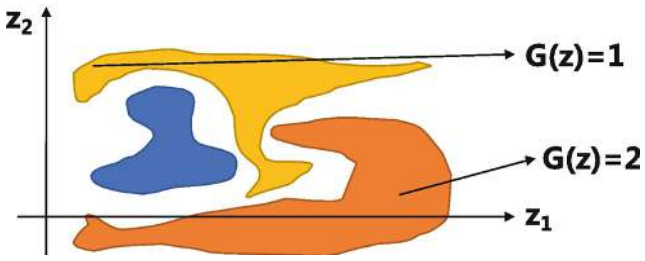


Fig. 5.10 Complex relationship between z-vector space and high-level semantics

Different colors in Fig. 5.10 indicate spatial distributions in Z corresponding to different high-level semantic attributes, such as different classes of handwritten

numbers and different face attributes. Although it must exist objectively, it is difficult for us to obtain an explicit representation of this distribution.

If Z corresponds to a simpler distribution of the generated results G(Z), then we can better control the properties of the generated images, as Fig. 5.11 shows a more linear Z-space of 1.

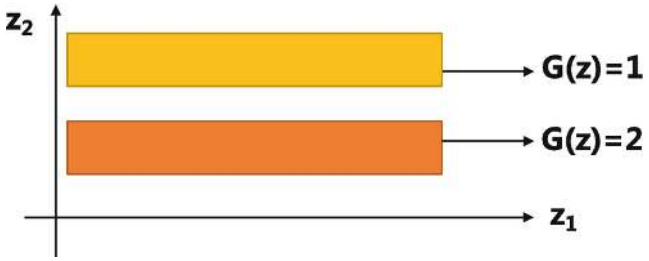


Fig. 5.11 Simple relationship between z-vector space and high-level semantic

How to obtain a simpler z-vector space is an important research of image generation framework we will introduce next, including conditional GAN, attribute editing GAN.

5.3 Conditional GAN

While primitive GANs can generate images that satisfy the data distribution in the training dataset, they have no way to directly control the properties of the generated results, such as generating a specific class of numbers, or a certain class of stroke styles. Instead, we often need controllable generation results which require networks where conditions can be controlled, and the relevant frameworks are mainly supervised conditional GAN and unsupervised conditional GAN.

5.3.1 Supervised Conditional GAN

Supervised Conditional GAN (CGAN) [6] achieves the control of the generated image data by using the conditional control vector directly as input, and Fig. 5.12 shows the schematic diagram of CGAN.

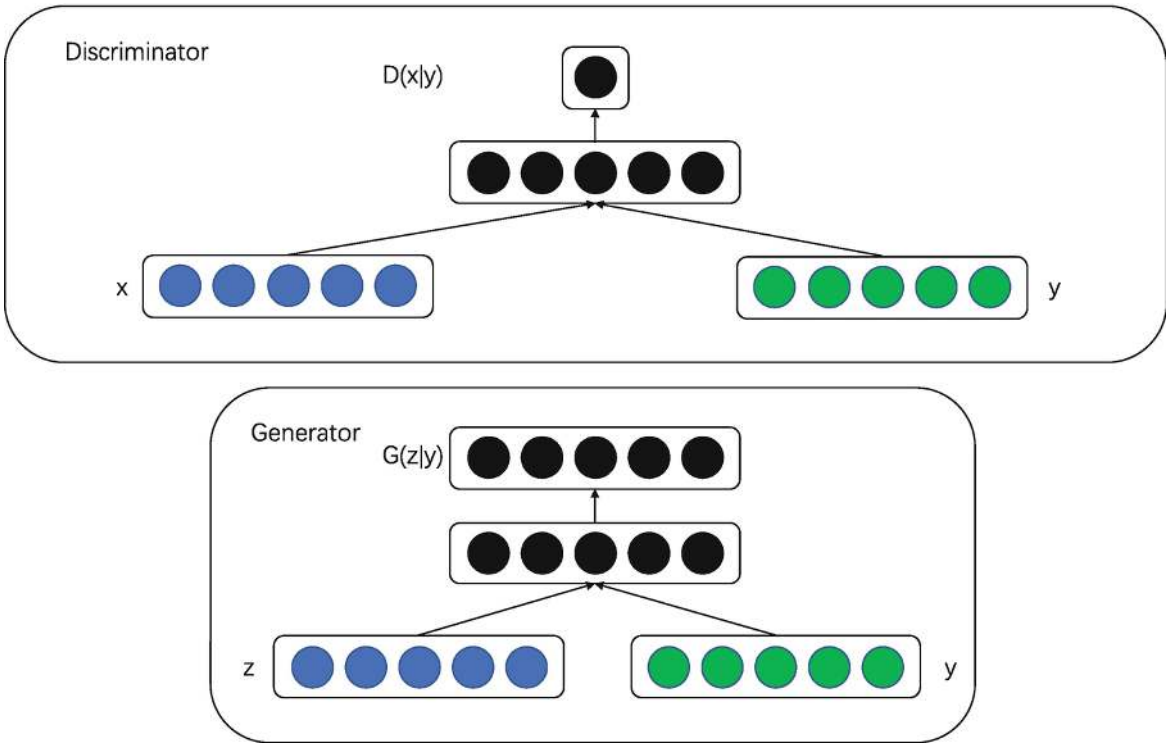


Fig. 5.12 Schematic diagram of CGAN structure

The *y* in Fig. 5.12 is the conditional vector, such as the label for image classification, which will be used as part of the input to the generator and discriminator, respectively.

When input to the generator, *y* needs to be directly spliced with the noise vector *z*.

When input to the discriminator, *y* needs to be filled with spatial dimensions and then stitched with image *x* by channel.

The optimization objective of the conditional GAN is of the same form as that of the original GAN, except that conditional constraints are added to the generator and discriminator inputs, as in Eq. (5.1).

$$\text{minmax}\, V(D,G) = E_{x\sim p_{data}(x)}[\log \mathrm{D}(\mathrm{x}|\mathrm{y})] + E_{z\sim p_z(z)}[\log\,(1-\mathrm{D}(\mathrm{G}(\mathrm{z}|\mathrm{y})))] \quad (5.1)$$

Figure 5.13 shows the results of number generation based on the control of conditional variables, and different rows in the figure correspond to different conditional vectors, i.e., number categories. The generation of specific numbers can be controlled by the conditional vectors, which is not possible with DCGAN.

Fig. 5.13 Graph of CGAN generation results

5.3.2 Unsupervised Conditional GAN

InfoGAN [7] is an unsupervised conditional GAN that, unlike CGAN, does not have its conditional control variables explicitly defined, i.e., it is not fully interpretable, but relies on the design of the optimization objective to capture the relationship between the implicit conditional variable c and the generated data.

The network structure of InfoGAN is shown in Fig. 5.14.

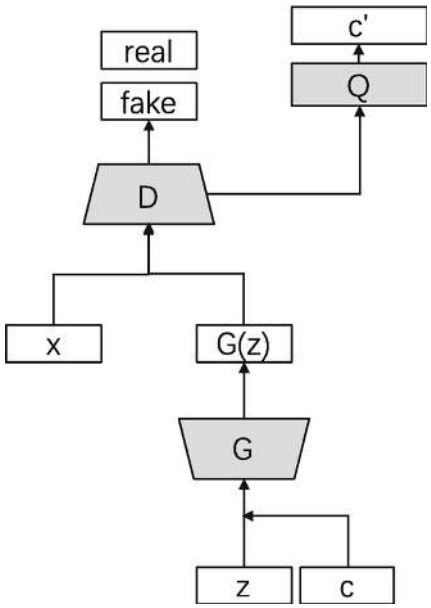


Fig. 5.14 Schematic diagram of InfoGAN structure

In CGAN, the conditional variable c is used as the labeled input, while InfoGAN implicitly splits the input noise into two parts, the incompressible noise z and the interpretable hidden variable c, and adds a fully connected layer $Q(c|x)$ to predict the hidden variable to obtain c', and computes the resulting c' with c mutual information is added to the loss.

The advantage of InfoGAN over CGAN is that the variable c can be not only a category but also other attributes that are not easily interpretable. In contrast to the standard GAN, the learning of this implicit attribute can be captured by maximizing the mutual information between the observed and implicit variables.

$$\min_{G,Q}\max_{D} V_{\text{InfoGAN}}(D,G,Q) = V(D,G) - \lambda L_I(G,Q) \tag{5.2}$$

Taking the handwritten digit generation as an example, let the dimension of c be 12, the first 10 dimensions represent the category attributes, and the next two dimensions represent the hidden attributes, we expect the model to learn two important attributes, stroke, and direction, and the generated results are shown in Fig. 5.15.

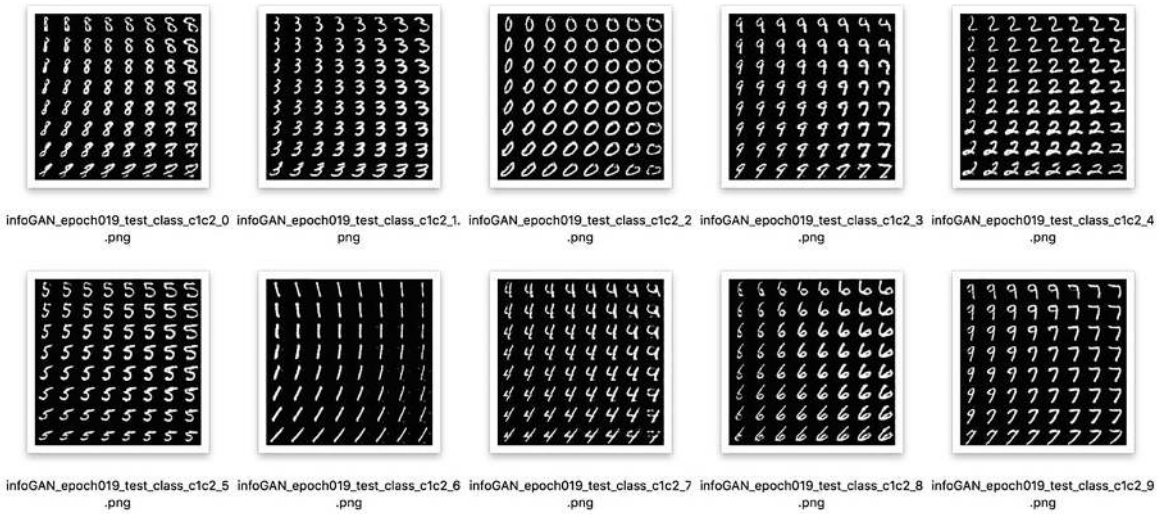


Fig. 5.15 InfoGAN generation results

Each figure in Fig. 5.15 generates a certain category of figures, which is the control of the conditional category property. And the figures in the plots have different stroke and rotation properties, which is the result of the control of the implicit conditional variables in the last two dimensions. It can be seen that InfoGAN does capture some attributes implicitly, and many subsequent GANs are inspired from it.

5.3.3 Semi-supervised Conditional GAN

If the category of the image is discriminated while generating the image, we can use GAN for image classification tasks, of which the representative frameworks are SGAN and ACGAN.

SGAN [8] compared with standard GAN, in fact, is that the output of the discriminator has been changed to include not only true-false discriminations but also multiple category discriminations, and its prediction output dimension is $N + 1$,

where *N* dimension is used to predict the category of true samples and one dimension is used to predict true-false.

ACGAN [9], on the other hand, adds an additional branch of classification output, which separates true-false discrimination from category discrimination as a separate auxiliary task, instead of accomplishing both category discrimination and true-false distinction through different dimensions of discriminator output directly like SGAN, which further improves the image generation quality.

The discriminator objective of ACGAN is the same as that of cGAN, with an additional classification objective, as defined in (5.3). Both real and fake samples need to be classified, and both discriminator and generator need to maximize Eq. (5.3).

$$L_c = E[\log \ P(C = c|X_{\text{real}})] + E[\log P(C = c|X_{\text{fake}})] \tag{5.3}$$

A comparison of the structures of CGAN, SGAN, InfoGAN, and ACGAN is shown in Fig. 5.16.

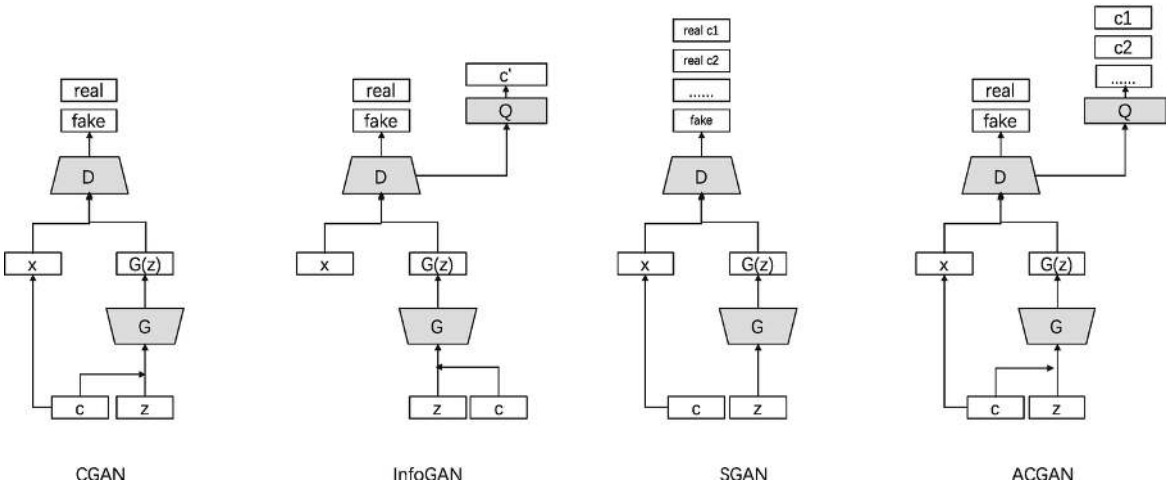


Fig. 5.16 Comparison of CGAN, SGAN, InfoGAN, and ACGAN structures

5.3.4 Complex Forms of Conditional Input

When we want to perform more complex control, we can use multiple conditions, even multimodal ones, as input, as well as add conditional control at multiple locations in the network.

5.3.4.1 Add Conditional Control in Multiple Locations

As Fig. 5.17 shows the case of image generation controlling different domain styles [10], where Domain is the domain vector. For the discriminator, the input image has the most obvious domain discriminable features, so the Domain vector is used as input along with the input map, while the domain invariant features need to be extracted in the next convolution layer, so the Domain vector is no longer input.

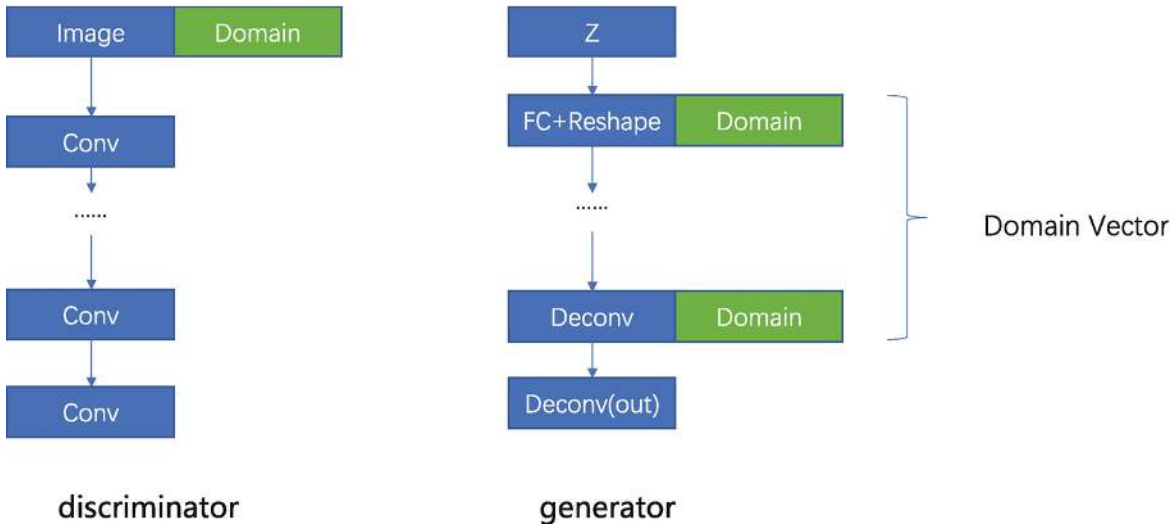


Fig. 5.17 Schematic diagram of multi-level conditional CGAN

The goal of the generator is to achieve domain-specific image generation, which requires that the features at each abstraction level contain domain information, so the Domain vector is fed into each deconvolution layer.

5.3.4.2 Multimodal Condition Control

The basic conditional GAN uses a one-dimensional label stitched together with an image as input; however, the conditional vector itself can also be an image or even a text string.

Take StackGAN [11] as an example, it generates text-to-image by inputting the description text as a condition to the generator and discriminator, respectively. The content of the image generated by StackGAN matches the description of the text, for example, the generated result is "a bird is singing," and "a red flower and the stamen is yellow".

The emergence of conditional GAN has led to a significant development of GAN-based image editing, image stylization and other tasks, and has become a very popular class of GAN model structures.

5.4 Multi-scale GAN

The early GAN network represented by DCGAN has two characteristics, one is that the resolution of the generated images is too low and the quality is not good enough, none of them exceeds 100 × 100 resolution. This is because the generator is difficult to learn to generate high-resolution samples at one time, and the discriminator is strong for high-resolution pictures, making the whole training process unstable.

Based on this, structures such as Pyramid GAN (i.e., LAPGAN) [12] and Progressive GAN (i.e., Progressive GAN) [13] are proposed and widely used, which take a step-by-step approach to generate images from coarse to fine with reference to

the pyramid structure inside the image domain, and each generation process learns the residuals instead of the complete image.

5.4.1 LAPGAN

Figure 5.18 shows the complete structure of LAPGAN, which starts from the input noiseZ_3 and boosting it step by step to finally generateI_0, which is a cascaded structure.

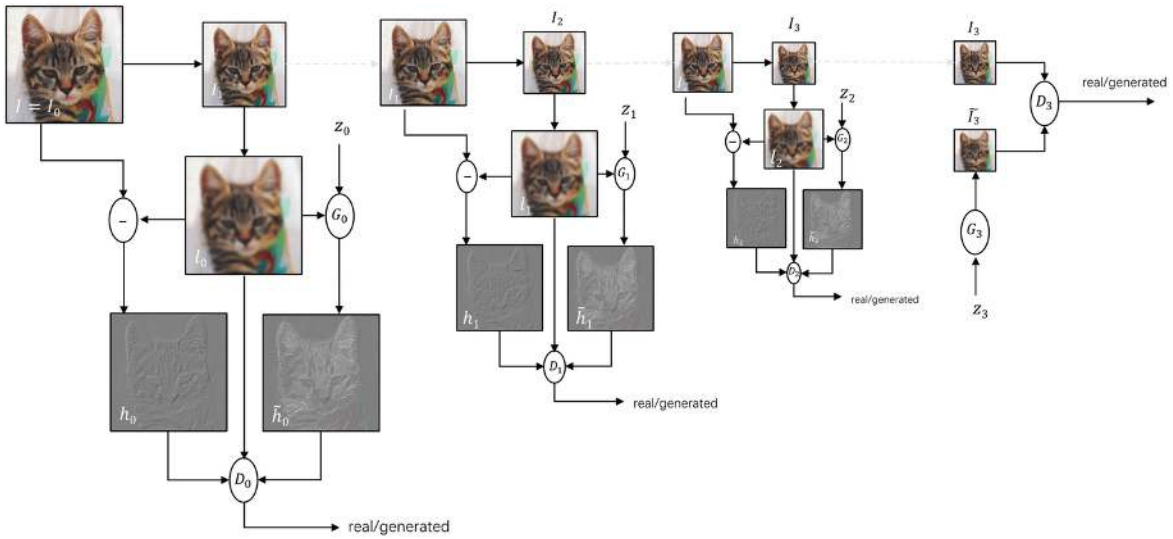


Fig. 5.18 LAPGAN training process

G_3 is the lowest level generator part, which has only random noise Z_3 as input, and the output is $\widetilde{I_3}$.
D_3 is the lowest level discriminator, its input contains real samples I_3 and the generated $\widetilde{I_3}$.
G_2 not only includes random noise Z_2 as input and also includes the upsampled image of I_3,named as l_2, the output is $\widetilde{h_2}$. Unlike $\widetilde{I_3}$, the $\widetilde{h_2}$ is a residual image, not real image, which will be send to the discriminator D_2, together with the residual of the real image I_2 and the upsampled image l_2.

Such a structure has several advantages:

(1) Approximation and learning for residuals are relatively easy, reducing the amount of content that needs to be learned for each GAN, and thus increasing the learning capacity of the GAN.

(2) Independent training step by step raises the difficulty of the network to simply remember the input samples, which is a problem faced by many high-performance deep networks.

The discriminator still needs to distinguish between real physically meaningful images and generated images, but the generator needs to learn only the residual part, which reduces the learning difficulty.

LAPGAN achieves a more stable convergence performance and higher resolution than DCGAN.

Since most GAN models are based on a dataset consisting of the same class of images sampled to learn the data distribution, and the images themselves are sufficiently self-similar. SinGAN [14] adopts a very similar structure to LAPGAN and can generate high-quality images based only on local image blocks of different resolutions of the same image for learning, which can be used in areas such as data enhancement. The results of generating various images at different scales based on one image are shown in Fig. 5.19.

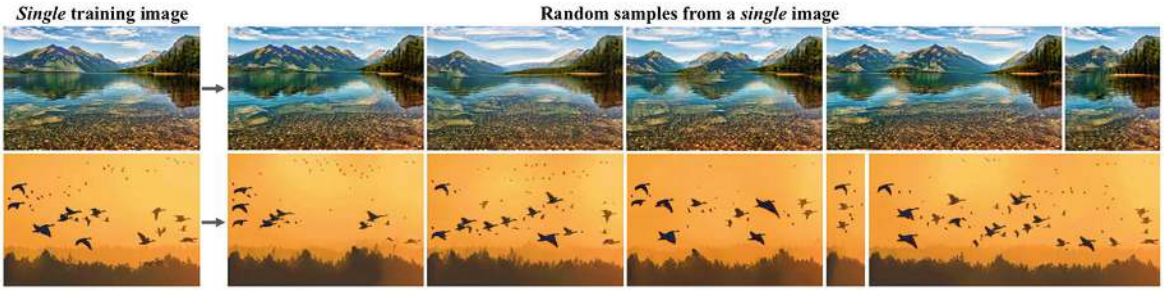


Fig. 5.19 SinGAN-generated image results

Column 1 in Fig. 5.19 shows the training images, and columns 2–6 show the generated images, and it can be seen that the generated images have a high degree of local similarity with the training images.

5.4.2 Progressive GAN

Although Progressive GAN is also a residual and multi-scale-based framework, it is different from the learning approach adopted by LAPGAN, which improves the resolution by gradually adding modules during the training process, and the whole model structure has only one generative network G and one discriminative network D, as shown in Fig. 5.20.

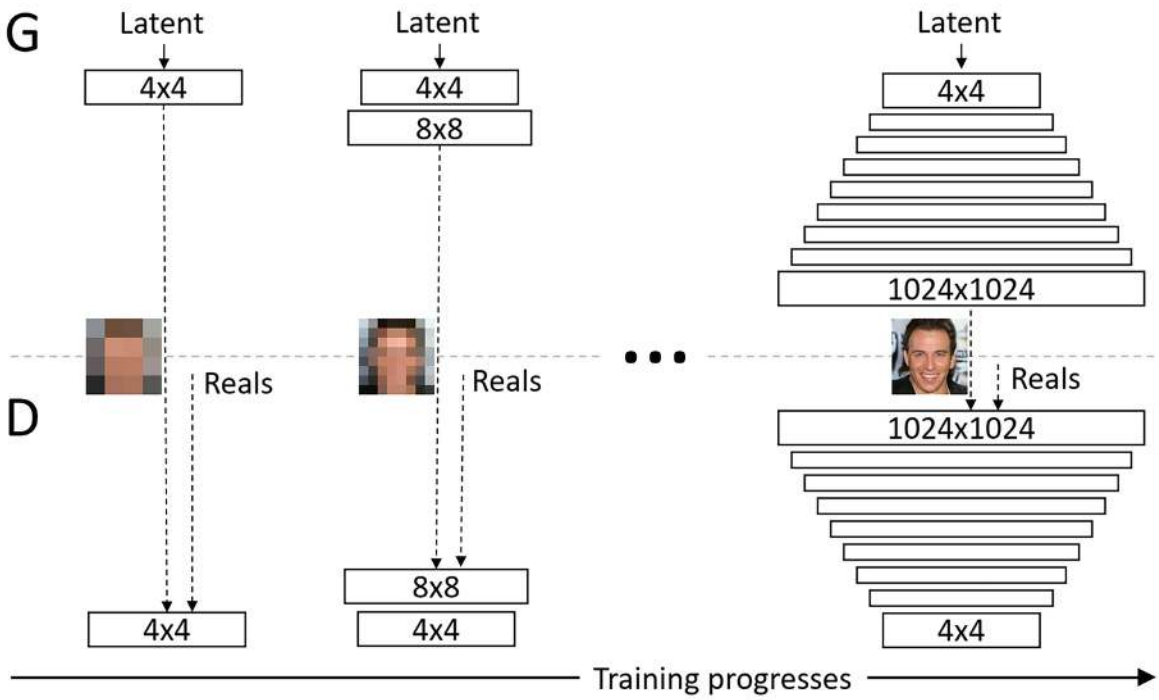


Fig. 5.20 Training process of Progressive GAN

The Progressive GAN uses the higher resolution branches as residual branches, gradually increasing the resolution as shown in Fig. 5.21.

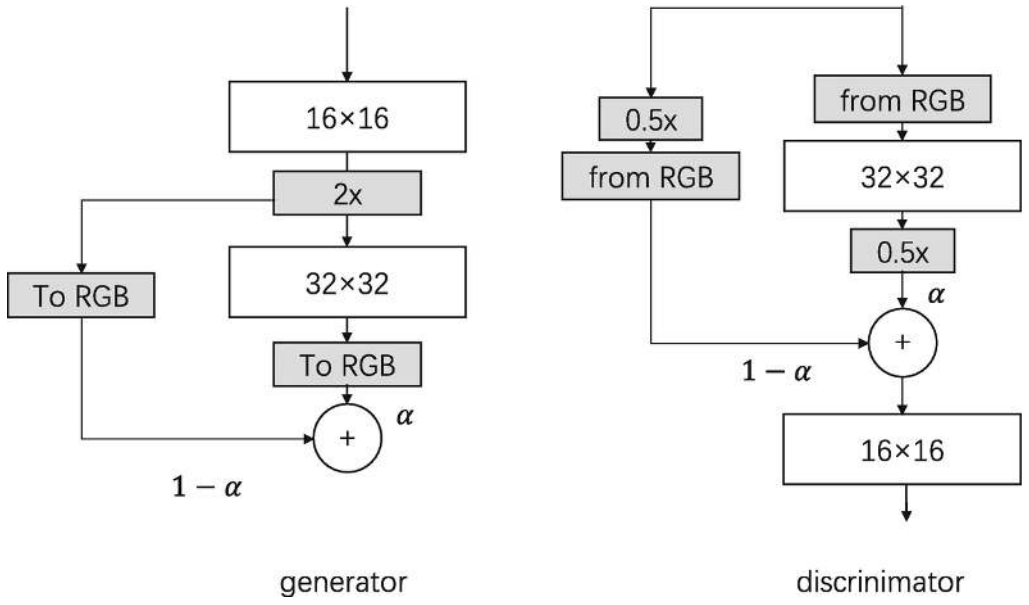


Fig. 5.21 Schematic diagram of residual learning of Progressive GAN

Assuming that the 16 × 16 resolution image generation learning has been completed, the next step is to add a 32 × 32 resolution.

The generator process is shown in the left half of Fig. 5.21, where feature map upsampling is first performed and then divided into two branches, 1 branch is

directly upsampled based on low-resolution features and uses a 1 × 1 ToRGB module to generate the image; the other branch is a residual branch that uses several feature layers that do not change the resolution for learning, and then the results of the two branches are weighted according to the coefficients.

The process of the discriminator is shown in the right half of Fig. 5.21 and is divided into two branches. 1 branch first performs 0.5× downsampling and then extracts features from the image using the from RGB module; the other branch is the residual branch, which first extracts features from the image using the RGB module, then learns using several convolutional layers that do not change the resolution, and finally downsamples the features.

5.5 Attribute GAN

In the previous subsection, we introduced conditional GAN, which controls the generation results by using conditional supervisory information in the generator and discriminator to not only produce label-specific data, but also improve the quality of the generated data. However, the attributes of the images themselves can be very complex, and how to achieve better attribute decoupling is crucial to the control of the generated results, so there are a series of studies dedicated to better learning of attribute vectors to improve the controllability of the generated image attributes, where the methods can be divided into explicit attribute GAN and implicit attribute GAN.

5.5.1 Explicit Attribute GAN

IcGAN [15] uses the idea of CGAN in reverse, first using the encoder to complete the learning from the image to the attribute itself, and then controlling the generated result by changing the attribute, it expects to decouple the attribute by learning the mapping from the image to the attribute through the encoder, and its model structure is shown in Fig. 5.22.

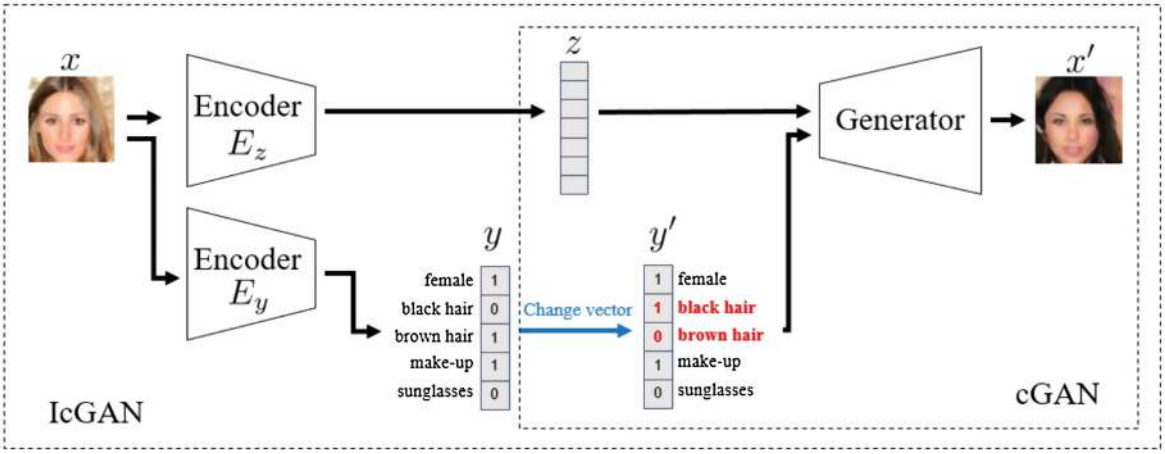


Fig. 5.22 IcGAN generation results

In the IcGAN architecture, the attribute vectors of the specified dimension need to be learned, and the picture x is encoded as feature vector z and attribute vector y by two encoders, respectively.

The training of IcGAN is performed in three steps:

Step 1: Using the attribute labels of the dataset as the conditional vectors of CGAN, by randomly sampling z, a conditional faces generation GAN is trained.
Step 2: Train the feature encoder E_z. The training data are the data x' generated by the face generator in step 1 and the corresponding input vector z.
Step 3: Train the attribute encoder E_y, the training data is the data x' generated by the generator in step 1 and the corresponding label vector y.

The encoder E_z and encoder E_y can be two completely independent encoders, or the model parameters can be further reduced by sharing some underlying features.

The loss functions used for the encoders Z and Y are as follows:

$$L_{ez} = \mathbb{E}_{z\sim p_z, y'\sim p_y} \| z - E_z(G(z,y')) \|_2^2 \tag{5.4}$$

$$L_{ey} = \mathbb{E}_{x,y\sim p_{data}, } \| y - E_y(G(x)) \|_2^2 \tag{5.5}$$

Because the quality of the generated images may not be good, in steps 2 and 3, the real dataset x and the corresponding label y can also be used for training.

When using the model for image generation, the feature vector z can be obtained by encoder E_z and the label vector y by encoder E_y, and then edit y and z together and input them to the generator.

5.5.2 Implicit Attribute GAN

Although IcGAN can learn attribute vectors, many microscopic attributes of the images themselves are difficult to describe quantitatively, and IcGAN can only achieve discrete attribute control, not continuous smooth attribute transformation.

StyleGAN [1] is a more powerful framework that can control the attributes of the generated images. It uses a new generation model, hierarchical attribute control, and Progressive GAN's training strategy to generate 1024 × 1024 resolution face images with precise control and editing of attributes, Fig. 5.23 shows the face images generated in the StyleGAN paper.

Fig. 5.23 Face map generated by StyleGAN

The structure of StyleGAN compared with the traditional generator is shown in Fig. 5.24.

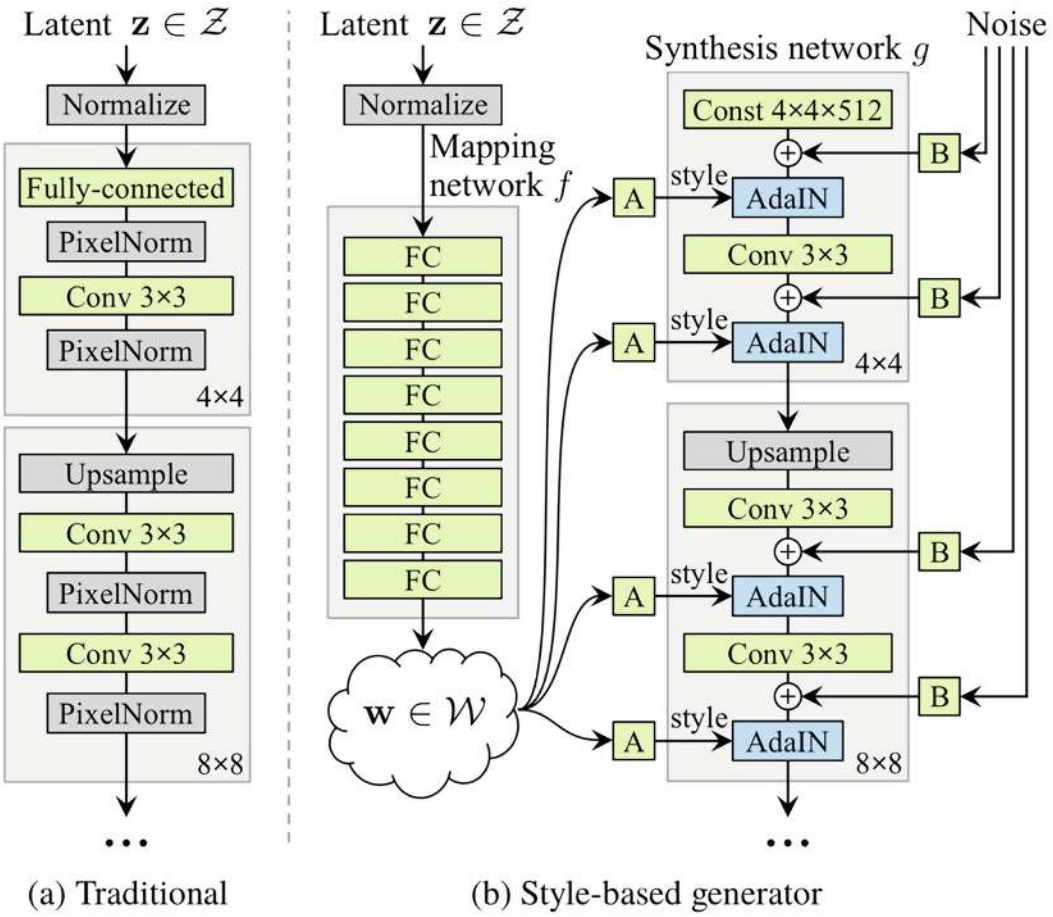


Fig. 5.24 Comparison of StyleGAN generator and traditional generator

Figure 5.24b is a schematic diagram of the structure of StyleGAN, which contains a mapping network *f* and a generative network synthesis network *g*. In the following, we interpret the structure of each part of it.

5.5.2.1 Mapping Network f

The mapping network f has a total of 8 fully connected layers, and the input is a 512-dimensional noise vector *Z*. After 8 fully connected layers, a 512-dimensional latent space vector W is obtained. The advantage of such encoding is to get rid of the input vector being influenced by the distribution of the input dataset, which is illustrated below with reference to a simple case in the paper, as shown in Fig. 5.25.

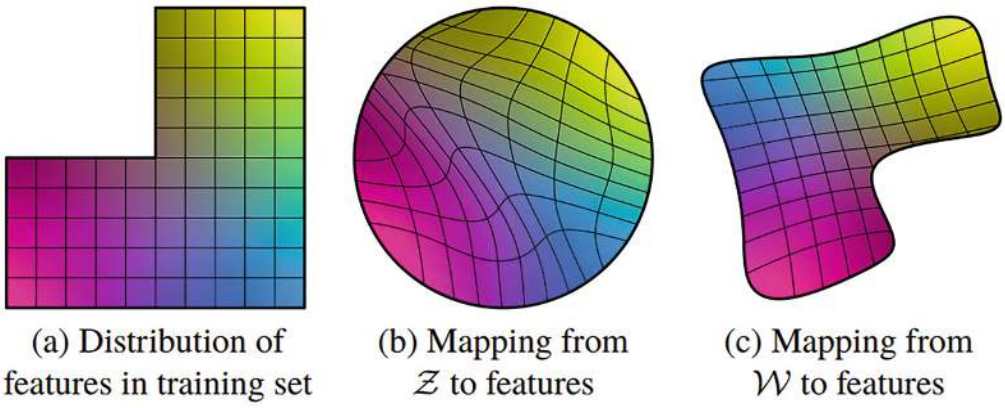


Fig. 5.25 The role of the mapping network *f*

The training dataset is usually biased, for example, in the attributes of human faces, gender includes male and female, and hair includes long and short, where the probability of {male, long hair} attributes appearing together is low, the probability of {male, short hair}, {female, long hair}, {female, short hair} appearing together is high, which is reflected to an uneven distribution in the space, as in Fig. 5.25a.

If we use only the randomly sampled noise vector *Z* to map, because the distribution of the noise *Z* is in the full space, there must be uneven mapping regions in order to fit the training dataset, as in Fig. 5.25b, which increases the difficulty of learning the model from *Z* to the generated image because the coupling relationship between the attributes is very complex.

If *W* is obtained by first mapping *Z* through the mapping network *f*, not only can we ensure a consistent distribution with the training set, but also obtain a more uniform distribution of attributes, and a better linear relationship between the latent vector space *W* and the attributes of the generated images, which is beneficial to the control of the attributes of the generated images, so *W* is more suitable as the input of the generator.

5.5.2.2 Generate Network g

Next, we look at the generative network g, which enables editing of face attributes at different granularities through hierarchical control.

The AdaIN layer is a normalization layer very widely used in the field of generative adversarial networks and stylization, which can replace the batch normalization layer (BN) for better results in style coding tasks, as defined below:

$$\mathrm{AdaIN}(x_i, y) = y_{s,i}\frac{x_i - \mu(x_i)}{\sigma(x_i)} + y_{b,i} \tag{5.6}$$

The specific implementation of AdaIN is to apply a learnable affine transformation to the 512-dimensional vector W and generate a scaling factor $y_{s,i}$ and deviation factor $y_{b,i}$. These two factors are weighted and summed with the output after Instance Normalization (IN), as in Eq. (5.6), and schematically in Fig. 5.26.

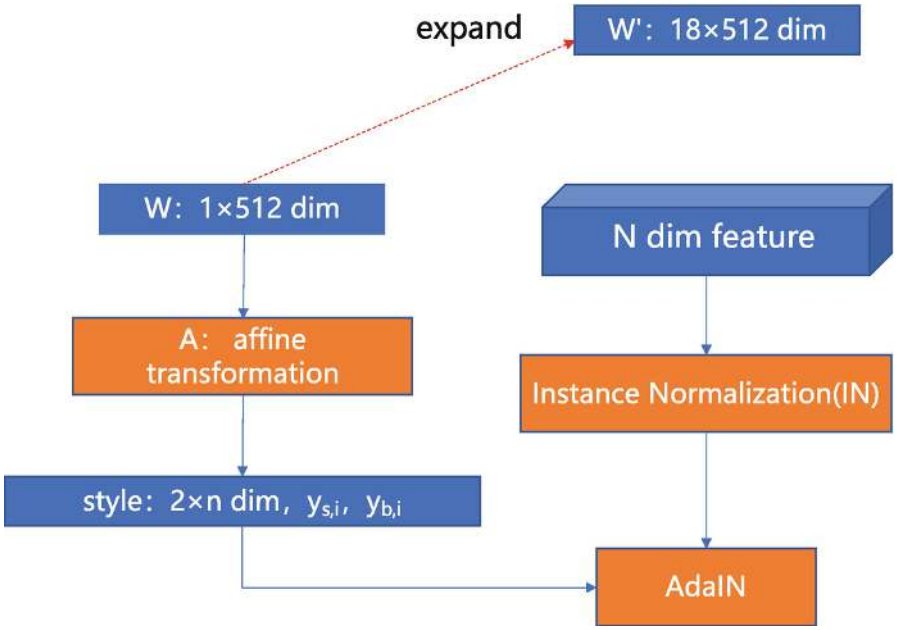


Fig. 5.26 The Schematic diagram of AdaIN

Later researchers found that it was beneficial to use different W vectors for different AdaIN layers, so the dimension of W was expanded to 18 × 512 and called *W'*, where 18 corresponds to the number of AdaIN layers.

Since the instance normalization is calculated separately for each feature map, the dimensions of scale factor $y_{s,i}$ and deviation factor $y_{b,i}$ are also related to the number of feature map channels. By control $y_{s,i}$ and $y_{b,i}$, we can achieve overall style control of the image, so they can be called style vectors.

The synthesis network g is a structure with increasing resolution step by step, with a total of 18 convolutional layers, except for layer 1, which samples one scale on every two layers, and the resolution is increased from 4 × 4 to 1024 × 1024, trained in the same way as Progressive GAN. Each level of resolution has two AdaIN layers, which we can call 1 stylized module, for a total of 9 stylized modules.

Taking the face images generated by StyleGAN as an example, the authors found in the experiments of the paper that the face features can be classified into three levels according to the scale, global features, intermediate features, and detail features, as shown in Fig. 5.27.

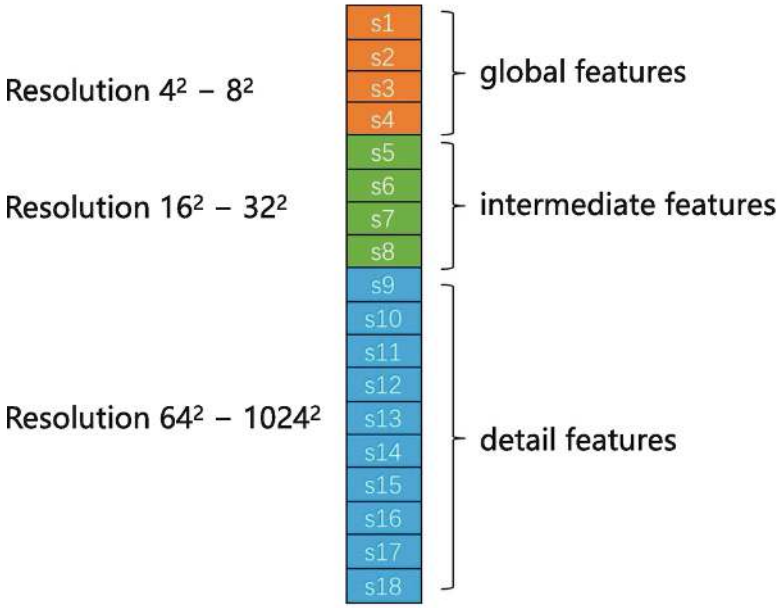


Fig. 5.27 Face layering style expression

The global features are controlled by a stylized module with a resolution of no more than 8 × 8, which mainly includes features such as facial pose, hairstyle, and facial shape.

Intermediate features are controlled by stylized modules with resolutions at 16 × 16 and 32 × 32, mainly including finer facial features, hair styles, eye opening and closing, etc.

Detail features are controlled by stylized modules with resolutions from 64 × 64 to 1024 × 1024, including mainly texture and color details of eyes, hair, and skin.

In addition, Gaussian noise is added after the convolution layer of each 1 stylized module and before the AdaIN layer. The noise input of each channel in each layer is shared, but needs to be multiplied by a learnable weight before adding to the feature map. The addition of noise allows for more fine-grained random control of the generated results and enhances the pattern richness of the generated images, and the related experimental results can be seen in practice in Sect. 5.8.

Because the features of the StyleGAN-generated image are controlled by the W and AdaIN layers, the initial input to the generator no longer requires input noise, but is replaced by a constant value of all ones.

5.5.2.3 Training Techniques

StyleGAN is a very good generative architecture, but just relying on a good architecture is not enough to achieve very high-quality generative results, but also needs some training techniques to assist the training of the model, mainly contains two, style regularization (i.e., mixing regularization) and *W* vector truncation.

In order to reduce the correlation of each level of features in the StyleGAN generator, StyleGAN uses the style regularization (mixing regularization) training technique. It achieves the exchange of two image styles by randomly selecting two

input vectors Z_1 and Z_2 during training, obtaining the intermediate vectors W_1 and W_2 through the mapping network, and then randomly swapping parts of W_1 and W_2.

As in Fig. 5.28, vector a is divided into two segments a_1 and a_2, and vector b is divided into two segments b_1 and b_2, respectively. Combining a1 and b2 into a new vector of the same length as a and b is a common style vector mixing.

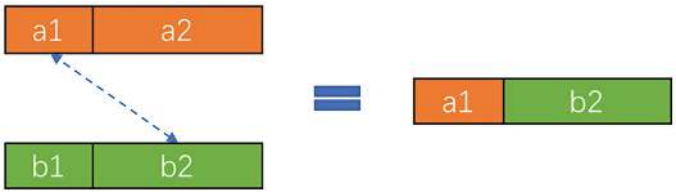


Fig. 5.28 Style vector blending

In the paper, the authors found that when migrating styles from domain B to domain A at a coarse granularity (small scale) of 4–8, the result is that the global information such as hairstyle and face shape from domain B is retained, while the color and texture come from domain A.

When migrating styles from domain B to domain A at a granularity of 16–32, the result is that small-scale facial details such as hair and eyes are retained in domain B, while global information such as posture is derived from domain A.

When migrating styles from domain B at 64–1024 fine granularity (large scale) to domain A, the result will retain some detailed textures and color styles from domain B, and the rest will come from domain A.

Another important technique is the truncation technique for the W vector, which is done by first calculating the statistical mean for the W vector to obtain $\overline{W}$ and then generating a new W vector by truncating the function ψ, as in Eq.

$$W' = \overline{W} + \psi\left(W - \overline{W}\right) \tag{5.7}$$

where the truncation function ψ has the value domain of (−1,1).

The impact of related training techniques on the generated image results can be read by the reader by skipping to the practice section in Sect. 5.8.

5.5.2.4 Evaluation of StyleGAN

We have introduced very many GAN evaluation techniques in the previous chapters, and StyleGAN additionally proposes two new evaluation methods, including perceptual path length and linear separability.

The path length is evaluated as the average distance between endpoints in the latent space Z or W, specifically calculated as the distance between two generated images at adjacent time nodes during the training process, defined in Eq. (5.8) for Z-based and in a similar way for W-based:

(5.8)

$$l_Z = \mathrm{E}\left[\frac{1}{\epsilon^2} d\Big(G(\mathrm{slerp}(z_1, z_2; t), G(\mathrm{slerp}(z_1, z_2; t + \epsilon))\Big)\right]$$

where slerp denotes spherical interpolation, a sampling method in spherical space; d denotes the L1 distance in VGG feature space; t denotes a certain time point, and $\boldsymbol{\epsilon}$ denotes the adjacent time step.

A very good latent space vector should be linearly distributed in space, i.e., along a certain path that can be edited for relevant attributes, and it is most efficient to sample on that path when we want to generate an image of a specific attribute, such as the green dashed path in Fig. 5.29, which can be sampled at any node to generate a "cat" map .

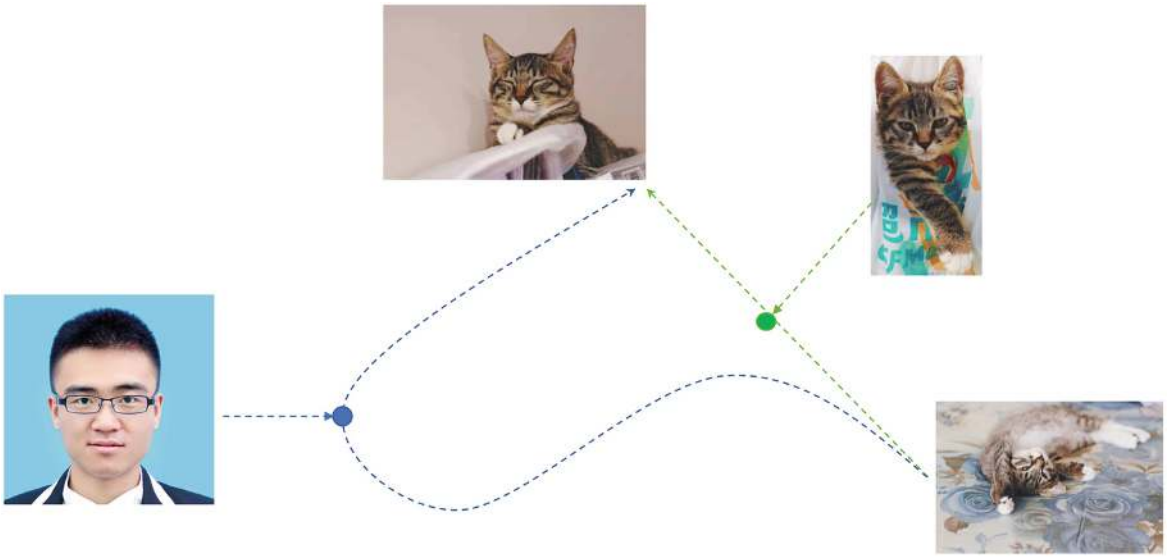


Fig. 5.29 Path length diagram

While the blue dashed line indicates a longer path, although sampling at the end of the path can generate pictures that satisfy the attributes, the vector obtained by sampling its intermediate nodes is not able to generate "cats," so the quality of the blue dashed path is not as good as that of the green dashed line. The visual representation of their quality difference in the graph is the length of the path, i.e., the perceptual path length, and a shorter path indicates a higher quality spatial mapping.

Another evaluation metric, Linear separability, is used to assess whether the Latent vector has sufficient attribute classifiability.

(a) First we generate 200,000 images using the distribution $z \sim P(z)$ and then train 1 CNN image classifier on them to obtain a binary label for a certain attribute, e.g., whether to smile or not.

Next we classify the latent space vector Z or W using SVM and calculate the conditional entropy $H(Y|X)$, where X is the result of the SVM classifier and Y is the result of the CNN picture classifier.

Figure 5.30 shows the two types of samples, smiling and neural, on the left, and the spatial distribution of the attribute vectors on the right.

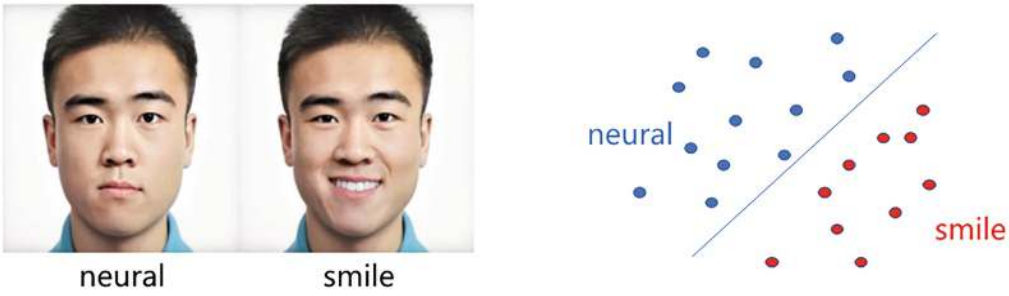


Fig. 5.30 Spatial distribution of samples and attribute vectors

If the attribute vectors in the latent space have better linear divisibility, the smaller $H(Y|X)$ will be, which indicates how much additional information is needed to determine the category, and lower values reflect a more divisible distribution of attribute vectors.

5.5.2.5 Structural Improvements of StyleGAN v2

StyleGAN v1 has an obvious drawback that the generated images all survive containing speckle-like artifacts, which is mainly a problem brought about by the AdaIN layer, as shown in Fig. 5.31.

Fig. 5.31 The defects of StyleGAN v1

StyleGAN v2 [16] improves this problem by modifying the structure of the generator, and a comparison with StyleGAN is shown in Fig. 5.32.

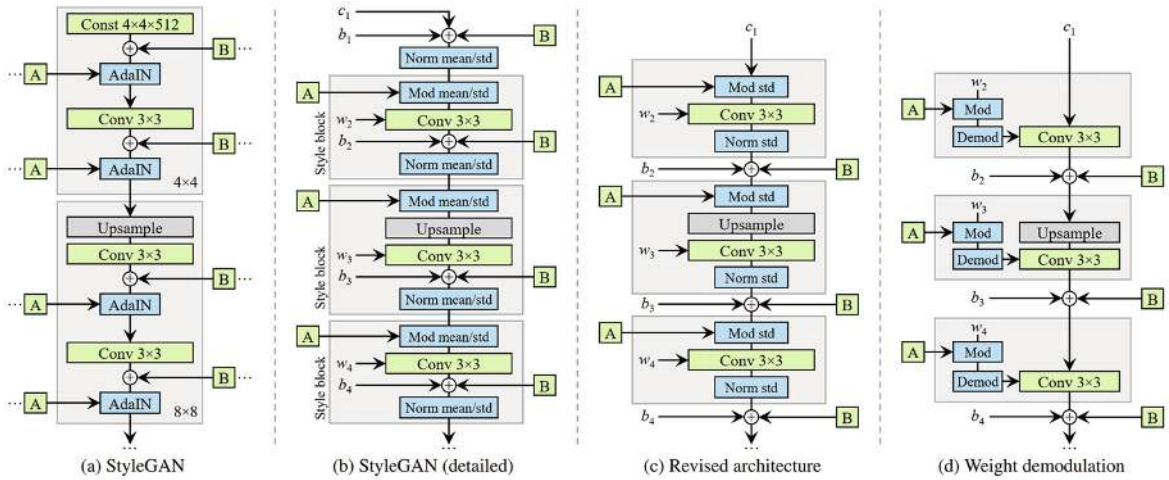


Fig. 5.32 Improvement of StyleGAN v2

The improvements to StyleGAN v2 consist of three parts:

(1) Remove the mean value in the normalization layer and move the noise and the bias of the convolution layer outside the style module.

(2) The noise broadcasting operation is simplified, instead of using a different noise for each 1 feature map, the same noise is used for all feature maps at a certain layer.

(3) Weight normalization: replaces the instance normalization layer with a demodulation layer, which is based on the statistical assumptions of the features rather than the actual content of the feature map, and is a weaker signal modulation compared to the instance normalization layer.

In the StyleGAN v1, A represents the affine transformation learned from *W*, which produces the style vector y, and B represents the broadcast operation of noise.

The input of each 1 style module is the normalized output of the IN layer of the previous style module, followed by modulation using the style vector learned from the affine transform, i.e., adding scaling and biasing, followed by upsampling, convolution, adding noise, and normalization operations.

StyleGAN v2 modifies the operations in the style module: including moving the summation of bias b and noise B outside the style module region and adjusting only the standard deviation of each feature map without modifying the mean value, as in Fig. 5.32c.

The modified structure can be further simplified to two operations, Mod and Demod.

Mod operates as in Eq. (5.9):

$$w'_{ijk} = s_i \bullet w_{ijk} \tag{5.9}$$

The Mod operation replaces each input feature map of the scaled convolution by scaling the convolution weights, where *i* denotes the input feature map, *j* denotes the output feature map, and *k* denotes the spatial location.

The original operation is to multiply the style vector first, followed by the convolution operation; the modified operation is to multiply the convolution kernel at the input channel level, i.e., the weights corresponding to each 1 input channel are multiplied by the style vector.

Demod operates as in Eq. (5.10):

$$w''_{ijk} = w'_{ijk} / \sqrt{\sum_{i,k} {w'_{ijk}}^2 + \epsilon} \tag{5.10}$$

The Demod operation scales the output by dividing the L2 norm of the corresponding weight, which normalizes the convolution kernel at the output channel level, corresponding to the summation of each of the 1 convolution weights connected to output channel j in the denominator of Eq. (5.9).

5.5.2.6 Training Tips for StyleGAN v2

StyleGAN v2 uses some new training techniques, including path regularization, delay regularization, and residual training modules.

(1) Path regularization: In StyleGAN v1, the path length is used to evaluate the results quantitatively, while StyleGAN v2 uses path regularization directly by adding it to the loss function, as defined in Eq. (5.11) below:

$$\mathrm{E}_{w,y\sim\mathbb{N}(0,I)} = \left(\left|\left| J_w^T y \right|\right|_2 - a \right)^2 \tag{5.11}$$

where $J_w^T y = \nabla_w (g(w) \bullet y)$, y is an image with normally distributed pixel values, w ~ f(z), z is uniformly distributed, and J_w is the first-order matrix of the generator over w, which represents the variation of the image over w. a is the long-term exponential moving average of $J_w y$, and the global optimum can be found automatically by training.

(2) Lazy regularization, i.e., the regular term is computed only once every *k* batches, which reduces the computational cost and overall memory usage of the regularized loss term. It also multiplies *k* at computation time to balance the overall size of its gradient. For discriminator, *k* = 16 and generator *k* = 8.

(3) Residual training module, i.e., simultaneous generation at different resolutions by adding skip connections, instead of using the progressive generation strategy in Progressive GAN, which is because the authors found that the progressive growth leads to significantly higher frequency components in the intermediate layers, which impairs the translation invariance of the network, making certain micro-semantic regions do not match the real image

characteristics although the accuracy is high. For example, when the face pose changes, the tooth region does not follow the pose change, thus generating texturally realistic but unrealistic images.

5.6 Multi-discriminator and Generator GAN

The conventional GAN contains only one discriminator and one generator, while adding generators and discriminators is also a class of design ideas that can improve the performance of the GAN model in some aspects, and in this section we briefly introduce the related designs.

5.6.1 Multi-discriminator GAN

Training a discriminator that is too good can damage the performance of the generator, which is one of the difficulties faced by GAN. If multiple discriminators that are not as strong can be trained and then cascaded, good results are expected to be achieved, which is the original design intention of the multi-discriminator structure.

Generative Multi-Adversarial Networks [17] is a structure that includes multiple discriminators a single generator, as shown schematically in Fig. 5.33.

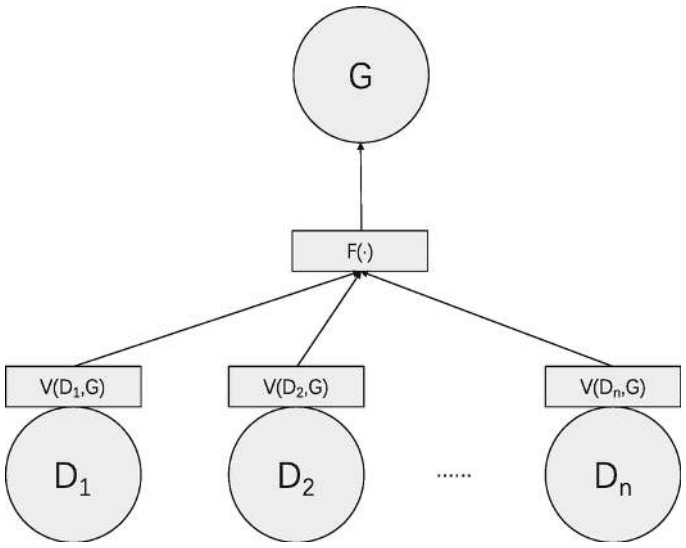


Fig. 5.33 Multi-discriminator single generator GAN

The benefit of using multiple discriminators brings similar advantages to model integration, and we can even apply Dropout techniques to it.

Multiple discriminators can divide the task, for example, in image classification, one for coarse-grained classification and one for fine-grained classification. In speech tasks, each is used for different vocal channels.

5.6.2 Multi-generator GAN

Generally, generators have a more difficult task to accomplish compared to discriminators because it has to complete the fitting of the probability density of the data, while discriminators only need to discriminate, leading to a problem that affects the performance of GAN, named pattern collapse, i.e., the generation of highly similar samples. This problem can be effectively alleviated by using multiple generators single discriminator approach.

Multi-Agent Diverse GAN [18] is a structure containing multiple generators with a single discriminator, as shown in Fig. 5.34.

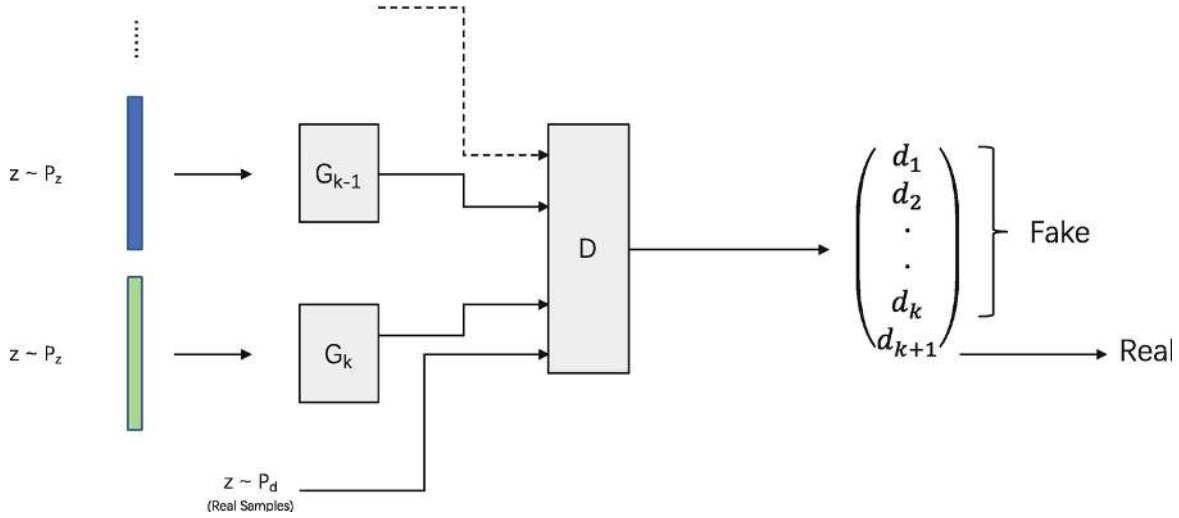


Fig. 5.34 Multi-generator single discriminator GAN

Multi-Agent Diverse GAN contains multiple generative branches with the same structure, these generators share some parameters at a shallow level, and the discriminator needs to be able to encourage a sufficiently rich pattern for these generators.

If there are *k* generators, the optimization objective of the generators is to minimize the following Eq. (5.12):

$$E_{x\sim p_d} \log D_{k+1}(\mathrm{x}) + \sum_{i=1}^{k} E_{x\sim p_{g_i}} \log\left(1 - D_{k+1}(\mathrm{x})\right) \tag{5.12}$$

$D_{k+1}(x)$ is a cross-entropy loss, i.e., determining from which generator the sample comes.

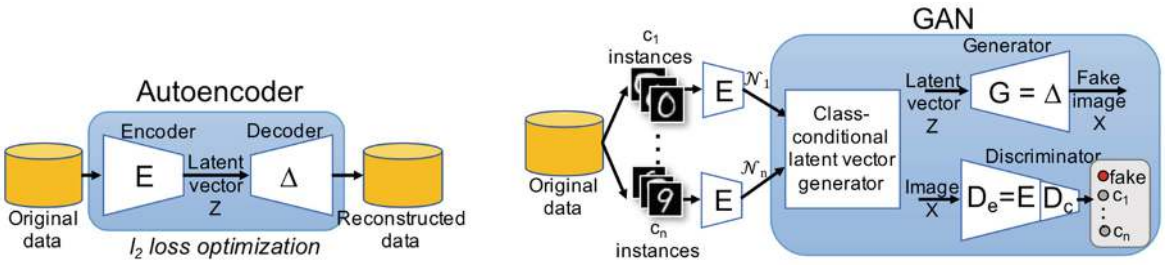
5.7 Data Enhancement and Simulation GAN

The data generation framework is the method, and how to apply it to various tasks is what we are more interested in. We have already introduced the application of GAN for data augmentation and simulation, and next we introduce a relevant

representative framework for each of the two aspects of data augmentation and simulation.

5.7.1 Data Augmentation GAN

GAN is used as an excellent generative framework for data augmentation, and Balancing GAN (BAGAN for short) [19] is one of the representatives, which consists of two steps, as shown in Fig. 5.35.



(a) Autoencoder training. (b) GAN initialization.

Fig. 5.35 Schematic diagram of BAGAN

Step 1: Use the Autoencoder to learn all the original data to get the common features of small sample category and large sample categories, which can avoid the problem of poor training of GAN network due to insufficient features learned from small sample data.
Step 2: Initialize the discriminator of the GAN using the encoder of the Autoencoder learned in step 1, and the decoder of the Autoencoder initializes the generator of the GAN, then train the GAN.

The generator input vector of the GAN is derived from the sampling of the normal distribution, and the mean and variance of the normal distribution are computed statistically from the features obtained by feeding the encoder with the set of samples of the particular class to be generated.

The prediction output vector of the discriminator is n+1 dimensions, where n dimensions are the category classification information and the other dimension is the real/fake sample classification.

5.7.2 Data Simulation GAN

We sometimes use simulated images to train machine learning models, such as training perception models in the field of autonomous driving based on simulated environments, but the generalization ability of the trained models will be affected due to the large difference between simulated and real images.

GAN can be used to enhance the realism of simulation data in addition to regenerating the data. SimGAN [2] is a scheme for optimizing simulation data with a generator G whose input is a synthetic image instead of a random vector.

SimGAN uses real images as supervision, allowing the generator to learn the mapping of synthetic image data distribution to real image data distribution.

It uses adversarial ideas and real pictures without labels for data enhancement of the simulated images, and the schematic diagram of the framework is shown in Fig. 5.36.

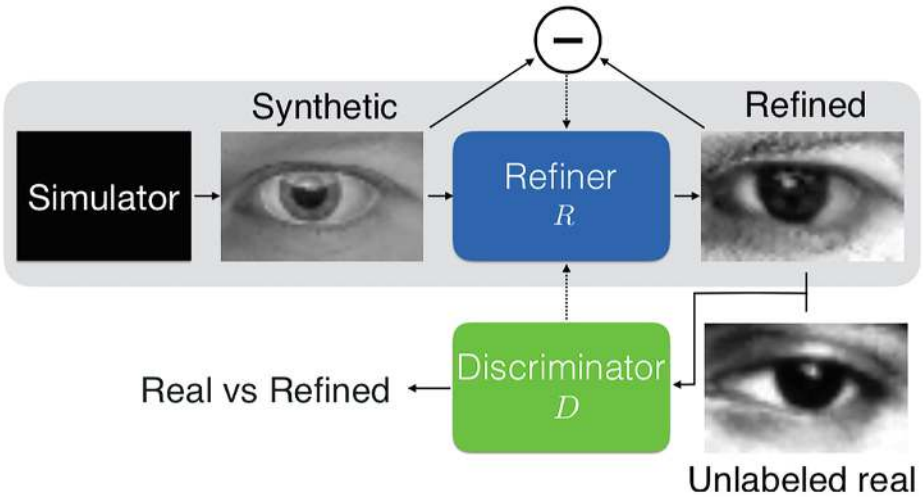


Fig. 5.36 Schematic diagram of SimGAN

The input in Fig. 5.36 is the simulated image, which goes through the refiner to generate a more realistic image.

The generator uses autoregressive loss (self-regularization loss) to ensure that the identity information of the refined images and the simulated images remains unchanged, and uses adversarial loss to discriminate the probability that the sample is a generated image, with the loss defined as follows:

$$L_R(\theta) = \sum_i L_{\text{real}}(\theta; x_i, y) + \lambda L_{\text{reg}}(\theta; x_i) \tag{5.13}$$

L_{real} is the authenticity constrained classification loss, where y denotes the true sample, and x_i denotes the generated sample, as defined in Eq. (5.14) below.$D_\emptyset$ denotes the discriminator, and R_θ is the generator Refiner.

$$L_{\text{real}}(\theta; x_i, y) = -\log\left(1 - D_\emptyset(R_\theta(x_i))\right) \tag{5.14}$$

L_{reg} is the regression loss of identity information, which can use either the L1 distance of the image or the distance in the feature space.

In order to make Refiner focusing on the local features of the image, the authors propose local adversarial loss to ensure that each small image region of the generated image should be real enough, specifically, let the image be divided into H × W local image blocks, and then each image block is discriminated as true or false, instead of the whole image being calculated directly, and finally the results of all regions are averaged, as in Fig. 5.37.

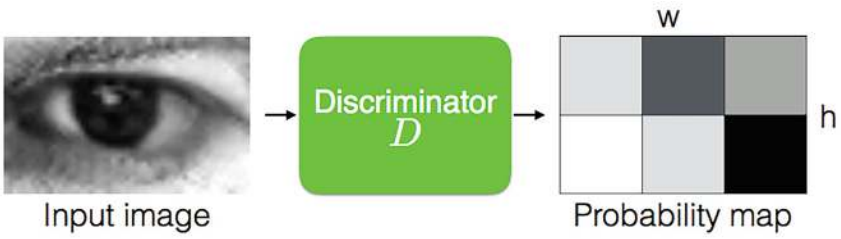


Fig. 5.37 Image block-based discrimination

There is a problem in training SimGAN because the D network uses only the latest generated images for discrimination, while the G network may generate duplicate images, but the D network has no memory of this.

Therefore, in training SimGAN, the authors use a trick that in each batch training, half of the images used for discrimination are from the currently generated batch images and the other half are from the history cache, thus ensuring more stable learning of the D network.

5.8 DCGAN Image Generation in Practice

DCGAN is the first truly fully convolutional generative adversarial network with a generator inputting a 1×100 vector that generates a 64×64 resolution image, which can achieve good results for handwritten digit recognition tasks. Although there are more better GAN models available, DCGAN is still worth learning.

5.8.1 Project Interpretation

Before the experiment, we first read in detail the code of the project, including the dataset and model definition, and optimization module.

5.8.1.1 Data Reading

This time we complete a task of face expression image generation using a dataset including 4358 images, some case diagrams are shown in Fig. 5.38.

Fig. 5.38 Schematic diagram of DCGAN training images

As there is only one class of images, we put them all in one folder, the data is very simple to read, so we directly use torchvision's ImageFolder interface, the core code is as follow:

```
## Read data
dataroot = "mouth/"
dataset = datasets.ImageFolder(root=dataroot.
transform=transforms.Compose([
transforms.Resize(image_size).
ToTensor().
Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)).
]))
dataloader = torch.utils.data.DataLoader(dataset,
batch_size=batch_size.
shuffle=True, num_workers=workers)
```

ImageFolder interface is used to create the datasets class, it needs input the root directory of the images and the data preprocessing function.

5.8.1.2 Discriminator Definition

Next we turn to the definition of a discriminator, which is an image classification model with a slightly different parameter configuration than in the original DCGAN paper.

```
## Discriminator Definition
class Discriminator(nn.Module).
```

```
def __init__(self, ndf=64, nc=3).
super(Discriminator, self). __init__()
self.ndf = ndf
self.nc = nc
self.main = nn.Sequential(
# input image size (nc) x 64 x 64, output (ndf) x 32 x 32,
convolution kernel size 4 x 4, stride size 2
nn.Conv2d(nc, ndf, 4, 2, 1, bias=False).
nn.LeakyReLU(0.2, inplace=True).

# Input (ndf) x 32 x 32, Output. (ndf*2) x 16 x 16,
convolution kernel size 4 x 4, stride size 2
nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False).
nn.BatchNorm2d(ndf * 2).
nn.LeakyReLU(0.2, inplace=True).

# input (ndf*2) x 16 x 16, output (ndf*4) x 8 x 8,
convolution kernel size 4 x 4, stride size 2
nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False).
nn.BatchNorm2d(ndf * 4).
nn.LeakyReLU(0.2, inplace=True).

# input (ndf*4) x 8 x 8, output (ndf*8) x 4 x 4, convolution
kernel size 4 x 4, stride size 2
nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False).
nn.BatchNorm2d(ndf * 8).
nn.LeakyReLU(0.2, inplace=True).

# input (ndf*8) x 4 x 4, output 1 x 1 x 1, convolution
kernel size 4 x 4, stride size 1
nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False).
nn.Sigmoid())

def forward(self, input).
return self.main(input)
```

The above code contains five convolutional layers, where the first four convolutional layers have a convolutional kernel size of 4 × 4, the width and height stride equal to 2, and the boundary filling value is 1. Each convolutional layer is followed by a batch normalization layer and a lrelu layer.

Suppose the input space scale is F_{in}, the convolution kernel scale is k, the boundary filling value is p, and the stride size is s. For the convolution layer, [−] denotes the downward rounding operation.

The spatial scale variation relationship of the input and output is Eq. (5.15):

(5.15)

$$F_o = \left[\frac{F_{in} - k + 2p}{s} + 1 \right]$$

Since the stride of the first four convolution layers are all equal to 2, it can be calculated by Eq. (5.15) that the image size is reduced to 1/2 of the original after each convolution.

The output layer is also a convolutional layer with an input feature map size space size of 4 × 4 and a convolutional kernel size of 4 × 4, so the output space layer dimension is 1. Using the sigmoid activation function, the output is a probability value between 0 and 1.

The structure of the discriminator after visualization using the Netron tool is shown in Fig. 5.39:

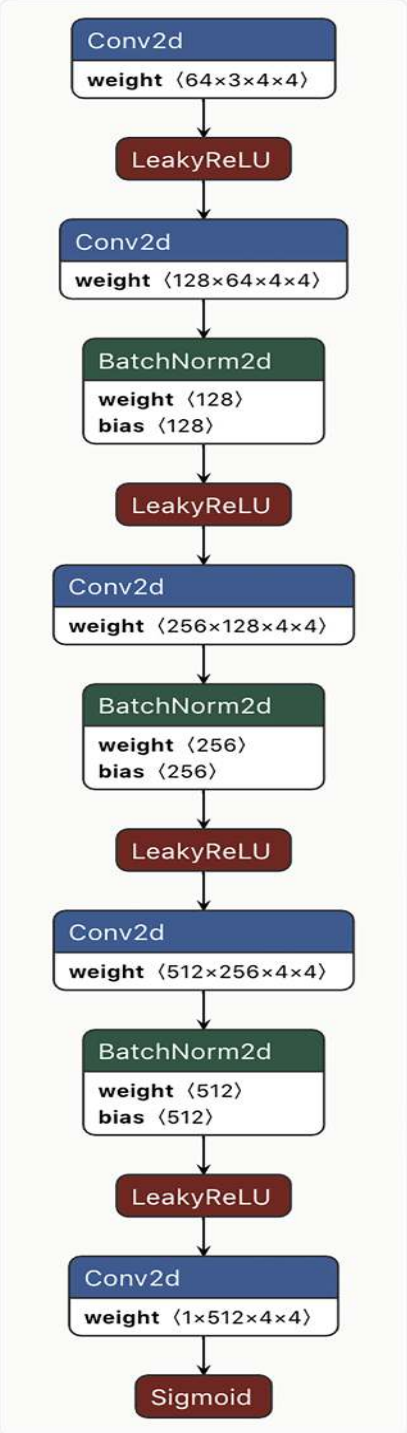


Fig. 5.39 DCGAN discriminator model visualization

5.8.1.3 Generator Definition

Next we turn to the definition of the generator, which inputs a one-dimensional noise vector and outputs a two-dimensional image, with a slightly different parameter configuration than in the original DCGAN paper.

```
## Generator Definition
class Generator(nn.Module).
def __init__(self, nz=100, ngf=64, nc=3).
super(Generator, self). __init__()
self.ngf = ngf
self.nz = nz
self.nc = nc
self.main = nn.Sequential(
# input nz x 1 x 1 , output (ngf*8) x 4 x 4, convolution
kernel size 4 x 4, stride size 1
nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False).
nn.BatchNorm2d(ngf * 8).
nn.ReLU(True).

# input (ngf*8) x 4 x 4, output (ngf*4) x 8 x 8, convolution
kernel size 4 x 4, stride size 2
nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False).
nn.BatchNorm2d(ngf * 4).
nn.ReLU(True).

# input (ngf*4) x 8 x 8, output (ngf*2) x 16 x 16,
convolution kernel size 4 x 4, stride size 2
nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False).
nn.BatchNorm2d(ngf * 2).
nn.ReLU(True).

# input(ngf*2) x 16 x 16, output(ngf) x 32 x 32, convolution
kernel size 4 x 4, stride size 2
nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False).
nn.BatchNorm2d(ngf).
nn.ReLU(True).

# input(ngf) x 32 x 32, output(nc) x 64 x 64, convolution
kernel size 4 x 4, stride size 2
nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False).
nn.Tanh())

def forward(self, input).
return self.main(input)
```

It can be seen that a total of five upsampling layers are included, each of which uses a transposed convolution with a convolution kernel size of 4 × 4, while the original authors used a fractional convolution of size 5 × 5.

The spatial scale variation relation of the transposed convolutional input and output is Eq. 5.16:

$$F_o = [(F_{in} - 1)s + k - 2p + p_o] \tag{5.16}$$

Unlike the padding in the convolution process, the transpose convolution has an additional output padding parameter p_o, which is an operation to add elements on one side of the feature map after convolution, while the input padding p is an operation to remove elements on the bilateral boundary. The other parameters in Eq. 5.16 have the same meaning as the corresponding parameters in Eq. (5.15).

According to Eq. (5.16), the first transposed convolutional layer upsamples the input 1D noise vector to generate a 4 × 4 size feature map, and the next four transposed convolutional layers are upsampled by a factor of 2.

The first four transposed convolutional layers are followed by BN and ReLU layers, and the last transposed convolutional layer is followed by the Tanh activation function.

The structure of the generator after visualization using the Netron tool is shown in Fig. 5.40:

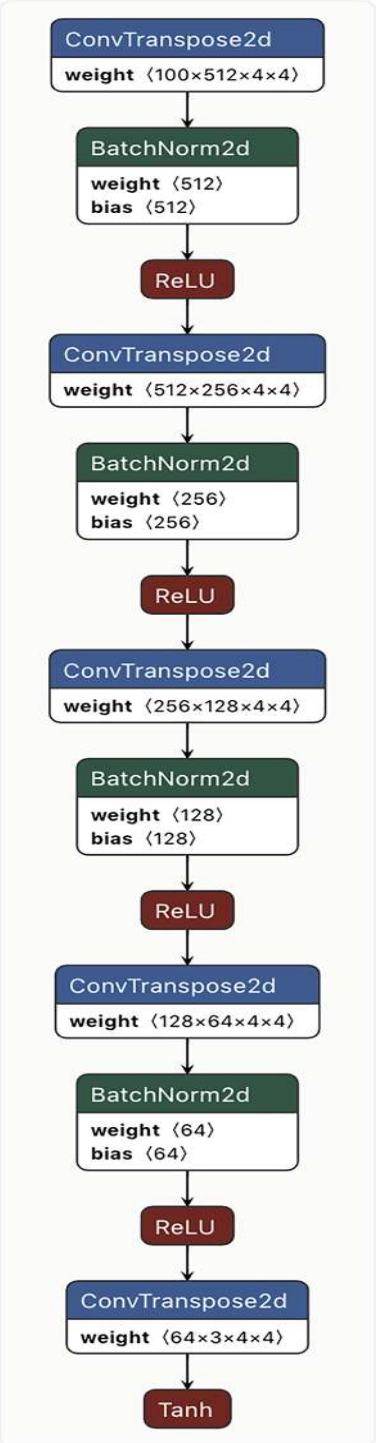


Fig. 5.40 DCGAN generator model visualization

5.8.1.4 Loss Function and Optimization Method Definition

The loss function uses a BCE cross-entropy loss with real and fake labels of 1 and 0, respectively.

```
criterion = nn.BCELoss()
real_label = 1. # "real" label
fake_label = 0. # "fake" label
```

Both the discriminator and the generator use the Adam method as an optimizer and use the same configuration, defined as follows:

```
lr = 0.0003
beta1 = 0.5
optimizerG = torch.optim.Adam(netG.parameters(), lr=lr,
betas=(beta1, 0.999))
optimizerD = torch.optim.Adam(netD.parameters(), lr=lr,
betas=(beta1, 0.999))
```

5.8.1.5 Training Parameters Configuration

Then we configure the training-related parameters, including the training input map size, batch size, feature map size, number of training iterations, etc., as follows:

```
# Batch size
batch_size = 64
# Training image size
image_size = 64
# Training image channels
nc = 3
# The length of the noise vector z
nz = 100
# Generators feature map quantity units
ngf = 64
# Discriminator feature map quantity units
ndf = 64
# training epochs
num_epochs = 100
```

5.8.1.6 Training Core Code

The steps for each training iteration are as follows:

```
for epoch in range(num_epochs).
lossG = 0.0
lossD = 0.0
for i, data in enumerate(dataloader, 0).
############################
## (1) Update D network: maximize log(D(x)) + log(1 -
D(G(z)))
###########################
```

```
## Training real pictures
netD.zero_grad()
real_data = data[0].to(device)
b_size = real_data.size(0)
label = torch.full((b_size,), real_label, device=device)

output = netD(real_data).view(-1)
## Calculate real image loss, gradient backpropagation
errD_real = criterion(output, label)
errD_real.backward()

## Training to generate images
## Generate latent vectors
noise = torch.randn(b_size, nz, 1, 1, device=devic
## Using G to generate images
fake = netG(noise)
label.fill_(fake_label)
output = netD(fake.detach()).view(-1)

## Calculate generation image loss, gradient backpropagation
errD_fake = criterion(output, label)
errD_fake.backward()

## Accumulation error, parameter update
errD = errD_real + errD_fake
optimizerD.step()

############################
# (2) Update G network: maximize log(D(G(z)))
###########################
netG.zero_grad()
label.fill_(real_label) # Assign a label to the generated
image

## Since the discriminator has been updated, discriminate
the generated graph once more
output = netD(fake).view(-1)

## Calculate generation image loss, gradient backpropagation
errG = criterion(output, label)
errG.backward()
optimizerG.step()

## Storage loss
lossG = lossG + errG.item()
```

```
lossD = lossD + errD.item()

iters += 1

writer.add_scalar('data/lossG', lossG, epoch)
writer.add_scalar('data/lossD', lossD, epoch)
torch.save(netG.state_dict(),'models/netG.pt')
```

5.8.2 Experimental Results

Once you have completed the definition of the above modules, you are ready for training.

5.8.2.1 Training Results

The training result curves are shown in Fig. 5.41.

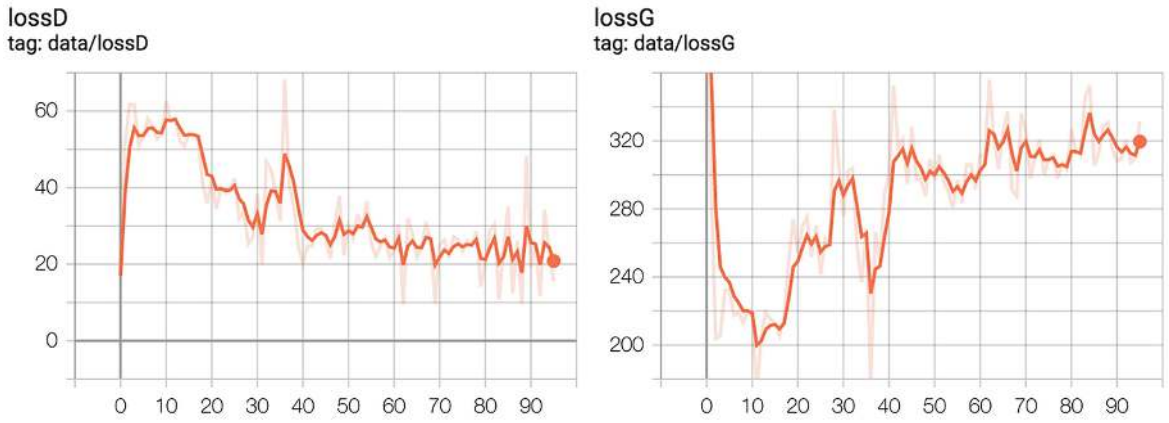


Fig. 5.41 DCGAN lip generation loss curve

Since the discriminators and generators of the generative adversarial network are trained alternately against each other, their respective loss curves are not likely to be at a level that keeps falling until very low, as is usually the case for image classification tasks, but have a process of falling and then rising.

For the discriminator, there is no learning at first, so the performance is poor, and as training proceeds, the loss of the discriminator decreases. But because the performance of the generator is improving, the loss of the discriminator starts to oscillate again after some time.

For the generator, there is no learning at first and the performance is poor, and as training proceeds, the performance gets better.

The two compete against each other until a better balance is reached, but it is still difficult to tell how well the model performs just from the loss curve itself, so we also have to look at the actual samples generated, as shown in Fig. 5.42.

Fig. 5.42 Generated results, from left to right, for the 0th, 10th, and 100th epoch

Figure 5.41 shows the generation results of the 0th, 10th, and 100th epochs from left to right, respectively.

From the results in Fig. 5.42, as the training progresses, many meaningful and very realistic samples are gradually generated. At 10 epochs, the generated images have obvious flaws, and by 100 epochs, some realistic samples have started to be generated. However, the final generated images still have some poor results, which is due to the model performance limitation of DCGAN itself, and we can use a better model to improve it.

5.8.2.2 Test Results

After the model has been trained, our next goal is to use it for inference, the code is as follow:

```
import torch
import torch.nn as nn
import torchvision.utils as vutils
import matplotlib.pyplot as plt

from net import Generator
device = torch.device("cuda:0" if (torch.cuda.is_available()
and ngpu > 0) else "cpu")
netG = Generator().to(device)

modelpath = sys.argv[1] ## modelpath
savepath = sys.argv[2] ## store path
netG.load_state_dict(torch.load(modelpath,map_location=lambda
storage,loc: storage))
netG.eval() ## Set the inference mode so that network layers
such as dropout and batchnorm switch between train and val
modes
torch.no_grad() ## Stop the autograd module from working to
speed up and save memory
nz = 100 ## Input noise vector dimension
```

```
for i in range(0,100).
noise = torch.randn(64, nz, 1, 1, device=device)
fake = netG(noise).detach().cpu()
rows = vutils.make_grid(fake, padding=2, normalize=True)
fig = plt.figure(figsize=(8, 8))
plt.imshow(np.transpose(rows, (1, 2, 0)))
plt.axis('off')
plt.savefig(os.path.join(savepath,"%d.png" % (i)))
plt.close(fig)
```

The core code for inference is to load the generator model using the torch.load function and then input a random noise vector to get the generated results. Figure 5.43 shows some of the generated sample plots.

Fig. 5.43 More lip image generation results

As we can see in Fig. 5.42, the overall generation results are still good, but there is still much room for improvement in this task, including but not limited to: (1) doing more data augmentation. (2) Improving the model. These will be left to the reader to experiment.

5.9 StyleGAN Face Image Generation in Practice

StyleGAN is a very important framework, and in this section we focus on the interpretation of the model code at the core of StyleGAN using pre-trained models for testing.

5.9.1 Project Profile

This project is a PyTorch replication of StyleGAN. On the one hand, because training the StyleGAN model requires many resources, most readers are not necessarily able to reproduce it; on the other hand because the model itself is very good and is used as a pre-training model by many studies, we focus here on the use of the model and not reproduce the whole model training like DCGAN.

The referenced project address is https://github.com/rosinality/style-based-gan-pytorch, we made some minor modifications in use, but did not change the core

functional code.

5.9.2 Model Interpretation

Next we start with a detailed interpretation of the model definition code.

5.9.2.1 Generator Definition

First, let's look at the definition of synthesis network:

```
## synthesis network definition
class Generator(nn.Module).
def __init__(self, code_dim, fused=True).
super(). __init__()
## 9 scale convolution blocks from 4×4 to 64×64 using
bilinear upsampling; from 64×64 to 1024×1024 using
transposed convolution for upsampling
self.progression = nn.ModuleList(
[
StyledConvBlock(512, 512, 3, 1, initial=True), ##4×4
StyledConvBlock(512, 512, 3, 1, upsample=True), ## 8×8
StyledConvBlock(512, 512, 3, 1, upsample=True), ## 16×16
StyledConvBlock(512, 512, 3, 1, upsample=True), ## 32×32
StyledConvBlock(512, 256, 3, 1, upsample=True), ## 64×64
StyledConvBlock(256, 128, 3, 1, upsample=True, fused=fused),
## 128×128
StyledConvBlock(128, 64, 3, 1, upsample=True, fused=fused),
## 256×256
StyledConvBlock(64, 32, 3, 1, upsample=True, fused=fused),
## 512×512
StyledConvBlock(32, 16, 3, 1, upsample=True, fused=fused),
## 1024×1024
]
)
## 9 scales of 1×1 to_rgb convolution layers, outputting
feature maps as RGB images, corresponding to 9 style modules
self.to_rgb = nn.ModuleList(
[
EqualConv2d(512, 3, 1).
EqualConv2d(512, 3, 1).
EqualConv2d(512, 3, 1).
EqualConv2d(512, 3, 1).
EqualConv2d(256, 3, 1).
EqualConv2d(128, 3, 1).
EqualConv2d(64, 3, 1).
EqualConv2d(32, 3, 1).
```

```
EqualConv2d(16, 3, 1).
]
)

def forward(self, style, noise, step=0, alpha=1,
mixing_range=(-1, 1)).
out = noise[0] ## Take the noise vector as input

if len(style) < 2: ## input only 1 style vector, means no
style blending, object_index=10
inject_index = [len(self.progression) + 1]
else.
## More than one style vector can be trained for style
mixing, generating a sequence of style mixing intersections
of length len(style) - 1)), whose value size does not exceed
step
inject_index = sorted(random.sample(list(range(step)),
len(style) - 1))

crossover = 0 ## position for style blending

for i, (conv, to_rgb) in enumerate(zip(self.progression,
self.to_rgb)).
if mixing_range == (-1, 1).
## Determine the index of the style mix based on the random
number generated earlier
if crossover < len(inject_index) and i >
inject_index[crossover].
crossover = min(crossover + 1, len(style))

style_step = style[crossover] ##Get the start point of the
crossover style

else.
## The interval to feel the style mixing according to
mixing_range, mixing_range[0] <= i <= mixing_range[1] take
style[1], others take style[0]
if mixing_range[0] <= i <= mixing_range[1].
style_step = style[1] ## Take the 2nd sample style
else.
style_step = style[0] ## take the 1st sample style

if i > 0 and step > 0.
out_prev = out
```

```
## Input noise and style vectors into the style module
out = conv(out, style_step, noise[i])

if i == step: ## last 1 level of resolution, output image
out = to_rgb(out) ## 1×1 convolution

## Whether the final result is alpha fused
if i > 0 and 0 <= alpha < 1.
skip_rgb = self.to_rgb[i - 1](out_prev) ##Get the result of
the previous level of resolution for 2x upsampling
skip_rgb = F.interpolate(skip_rgb, scale_factor=2,
mode='nearest')
out = (1 - alpha) * skip_rgb + alpha * out

break

return out
```

First of all, you can see that there are nine style modules, i.e., StyledConvBlock, of which the first style module does not need to be upsampled and the remaining eight modules need to be upsampled. Each style module corresponds to a to_rgb convolution layer, which can output the current resolution image.

The input to the style module consists of noise vectors and style vectors, and we next decode the style module:

```
## Style module layers, including two convolutional, two
AdaIN layers
class StyledConvBlock(nn.Module).
def __init__(
self.
in_channel.
out_channel.
kernel_size=3.
padding=1.
style_dim=512.
initial=False.
upsample=False.
fused=False.
).
super(). __init__()

## 1st style layer, initialize 4×4×512 feature map
if initial.
self.conv1 = ConstantInput(in_channel)
```

```
else.
if upsample: ## upsampling layer
if fused: ##Use transposed convolutional upsampling for
resolutions of 128 and above
self.conv1 = nn.Sequential(
FusedUpsample(
in_channel, out_channel, kernel_size, padding=padding
).
Blur(out_channel),## filtering operation
)

else.
## For resolution less than 128, use nearest neighbor
upsampling
self.conv1 = nn.Sequential(
nn.Upsample(scale_factor=2, mode='nearest').
EqualConv2d(
in_channel, out_channel, kernel_size, padding=padding
).
Blur(out_channel),## filtering operation
)

else: ##Non upsampling layer
self.conv1 = EqualConv2d(
in_channel, out_channel, kernel_size, padding=padding
)

self.noise1 = equal_lr(NoiseInjection(out_channel)) ##noise
module1
self.adain1= AdaptiveInstanceNorm(out_channel, style_dim)
##AdaIN module 1
self.lrelu1 = nn.LeakyReLU(0.2)

self.conv2 = EqualConv2d(out_channel, out_channel,
kernel_size, padding=padding)
self.noise2 = equal_lr(NoiseInjection(out_channel)) ##noise
module 2
self.adain2= AdaptiveInstanceNorm(out_channel, style_dim)
##AdaIN module 2
self.lrelu2 = nn.LeakyReLU(0.2)

def forward(self, input, style, noise).
out = self.conv1(input)
out = self.noise1(out, noise)
out = self.lrelu1(out)
```

```
out = self.adain1(out, style)

out = self.conv2(out)
out = self.noise2(out, noise)
out = self.lrelu2(out)
out = self.adain2(out, style)

return out
```

As can be seen, except for the first style layer which outputs a constant feature map of size 4 × 4 × 512 with a value of all 1s, all others need to be upsampled, using transposed convolutional upsampling for resolutions of 128 and above, and nearest neighbor upsampling for resolutions below 128.

where ConstantInput is defined as follows:

```
class ConstantInput(nn.Module).
def __init__(self, channel, size=4).
super(). __init__()
self.input = nn.Parameter(torch.randn(1, channel, size,
size))

def forward(self, input).
batch = input.shape[0]
out = self.input.repeat(batch, 1, 1, 1)
return out
```

Transposed convolutional upsampling is defined as follows:

```
## transpose convolutional upsampling, where the weight
parameter is self-defined
class FusedUpsample(nn.Module).
def __init__(self, in_channel, out_channel, kernel_size,
padding=0).
super(). __init__()

weight = torch.randn(in_channel, out_channel, kernel_size,
kernel_size)
bias = torch.zeros(out_channel)

fan_in = in_channel * kernel_size * kernel_size ## Number of
neurons
self.multiplier = sqrt(2 / fan_in)

self.weight = nn.Parameter(weight)
self.bias = nn.Parameter(bias)
```

```
self.pad = padding

def forward(self, input).
weight = F.pad(self.weight * self.multiplier, [1, 1, 1, 1])
weight = (
weight[:, :, 1:, 1:]
+ weight[:, :, :-1, 1:]
+ weight[:, :, 1:, :-1]
+ weight[:, :, :-1, :-1]
) / 4

out = F.conv_transpose2d(input, weight, self.bias, stride=2,
padding=self.pad)

return out
```

The noise module is defined as follows, which is summed and fused by weights and images:

```
## Add noise, noise weights can be learned
class NoiseInjection(nn.Module).
def __init__(self, channel).
super(). __init__()
self.weight = nn.Parameter(torch.zeros(1, channel, 1, 1))

def forward(self, image, noise).
return image + self.weight * noise
```

The AdaIN module is defined as follows, which controls the style through scaling and bias factors:

```
## Adaptive IN layer
class AdaptiveInstanceNorm(nn.Module).
def __init__(self, in_channel, style_dim).
super(). __init__()
self.norm = nn.InstanceNorm2d(in_channel) ## Create IN layer
self.style = EqualLinear(style_dim, in_channel * 2) ## Fully
connected layer, turning W vectors into AdaIN layer
coefficients S
self.style.linear.bias.data[:in_channel] = 1
self.style.linear.bias.data[in_channel:] = 0

def forward(self, input, style).
## input style is style vector W, length 512; after
self.style to get output style matrix S, the number of
```

```
channels is equal to 2 times the number of input channels
style = self.style(style).unsqueeze(2).unsqueeze(3)
gamma, beta = style.chunk(2, 1) ## Get scaling and bias
coefficients, divided into 2 parts by 1 axis (channel)
out = self.norm(input) ##IN normalized
out = gamma * out + beta

return out
```

The Style vector needs to be learned from the W vector by affine transformation, EqualLinear is defined as follows:

```
## Fully connected layer
class EqualLinear(nn.Module).
def __init__(self, in_dim, out_dim).
super(). __init__()

linear = nn.Linear(in_dim, out_dim)
linear.weight.data.normal_()
linear.bias.data.zero_()

self.linear = equal_lr(linear)

def forward(self, input).
return self.linear(input)
```

The input dimension of the EqualLinear layer is style_dim, i.e., 512, and the output is in_channel * 2, where multiplying by 2 is because the scaling and bias coefficients are to be generated in two copies, and in_channel corresponds to the number of channels to be acted upon.

In the above code, we can see that whether it is a convolutional layer or a fully connected layer, the equal_lr function needs to be called to normalize the weights, which is one of the training engineering techniques of StyleGAN. It normalizes the weights according to the number of neurons in the current layer, so as to achieve the effect of making each layer have an equal learning rate.

```
## Normalized learning rate
class EqualLR.
def __init__(self, name).
self.name = name

def compute_weight(self, module).
weight = getattr(module, self.name + '_orig')
## Number of input neurons, number of convolution kernels
per layer = Nin*Nout*K*K.
```

```
fan_in = weight.data.size(1) * weight.data[0][0].numel()
return weight * sqrt(2 / fan_in)

@staticmethod
def apply(module, name).
fn = EqualLR(name)

weight = getattr(module, name)
del module._parameters[name]
module.register_parameter(name + '_orig',
nn.Parameter(weight.data))
module.register_forward_pre_hook(fn)

return fn

def __call__(self, module, input).
weight = self.compute_weight(module)
setattr(module, self.name, weight)

def equal_lr(module, name='weight').
EqualLR.apply(module, name)

return module
```

The complete generator is defined as follows:

```
## Full generator definition
class StyledGenerator(nn.Module).
def __init__(self, code_dim=512, n_mlp=8).
super(). __init__()

self.generator = Generator(code_dim) ## synthesis network

## mapping network definition, contains 8 fully connected
layers, n_mlp=8
layers = [PixelNorm()]
for i in range(n_mlp).
layers.append(EqualLinear(code_dim, code_dim))
layers.append(nn.LeakyReLU(0.2))

## mapping network f for generating Latent vector W (i.e.
style vector) from noise vector Z
self.style = nn.Sequential(*layers)

def forward(
self.
```

```
input, ## input vector Z
noise=None, ## noise vector, optional
step=0, ## upsampling factor
alpha=1, ## fusion factor
mean_style=None, ## Average style vector W
style_weight=0, ##Style vector weight
mixing_range=(-1, 1), ##Mixing interval variables
).
styles = [] ## style vector W
if type(input) not in (list, tuple).
input = [input]

for i in input.
styles.append(self.style(i)) ## call mapping network,
generate the i-th style vector W

batch = input[0].shape[0] ## batchsize size

if noise is None.
noise = []

for i in range(step + 1): ## 0 to 8, total 9 levels of noise
size = 4 * 2 ** i ## scale of each layer, the first layer is
4 * 4, each layer of the individual channels share noise
noise.append(torch.randn(batch, 1, size, size,
device=input[0].device))

## Get the full style vector based on the average style
vector and the currently generated style vector
if mean_style is not None.
styles_norm = [] ## styles_array [1*512]

for style in styles.
styles_norm.append(mean_style + style_weight * (style -
mean_style))

styles = styles_norm

return self.generator(styles, noise, step, alpha,
mixing_range=mixing_range)
```

The above is the main code of the generator, next we will look at the definition of the discriminator.

5.9.2.2 Discriminator Definition

The discriminator also adopts the progressive discriminant structure in Progressive GAN, defined as follows:

```
class Discriminator(nn.Module).
def __init__(self, fused=True, from_rgb_activate=False).
super(). __init__()
self.progression = nn.ModuleList(
[
ConvBlock(16, 32, 3, 1, downsample=True, fused=fused), ##
512×512
ConvBlock(32, 64, 3, 1, downsample=True, fused=fused), ##
256×256
ConvBlock(64, 128, 3, 1, downsample=True, fused=fused), ##
128×128
ConvBlock(128, 256, 3, 1, downsample=True, fused=fused), ##
64×64
ConvBlock(256, 512, 3, 1, downsample=True), ## 32×32
ConvBlock(512, 512, 3, 1, downsample=True), ## 16×16
ConvBlock(512, 512, 3, 1, downsample=True), ## 8×8
ConvBlock(512, 512, 3, 1, downsample=True), ## 4×4
ConvBlock(512, 512, 3, 1, 4, 0).
]
)

## Conversion from RGB images to probability
def make_from_rgb(out_channel).
if from_rgb_activate.
return nn.Sequential(EqualConv2d(3, out_channel, 1),
nn.LeakyReLU(0.2))

else.
return EqualConv2d(3, out_channel, 1)

self.from_rgb = nn.ModuleList(
[
make_from_rgb(16).
make_from_rgb(32).
make_from_rgb(64).
make_from_rgb(128).
make_from_rgb(256).
make_from_rgb(512).
make_from_rgb(512).
make_from_rgb(512).
make_from_rgb(512).
]
```

```
)

self.n_layer = len(self.progression)
self.linear = EqualLinear(512, 1)

def forward(self, input, step=0, alpha=1).
for i in range(step, -1, -1).
index = self.n_layer - i - 1

if i == step: ## top level, input image
out = self.from_rgb[index](input)

if i == 0.
out_std = torch.sqrt(out.var(0, unbiased=False) + 1e-8)
mean_std = out_std.mean()
mean_std = mean_std.expand(out.size(0), 1, 4, 4)
out = torch.cat([out, mean_std], 1)

out = self.progression[index](out)

## Adjacent layer fusion for discriminators
if i > 0.
if i == step and 0 <= alpha < 1.
skip_rgb = F.avg_pool2d(input, 2)
skip_rgb = self.from_rgb[index + 1](skip_rgb)
out = (1 - alpha) * skip_rgb + alpha * out

out = out.squeeze(2).squeeze(2)
out = self.linear(out)

return out
```

First of all, you can see that a total of nine convolutional modules, i.e., ConvBlock, are included, of which the 9th style module does not need to be downsampled and the remaining eight modules need to be downsampled. Each style module corresponds to a make_from_rgb convolutional layer, which can output the real/fake prediction probability based on the current resolution of the image.

The ConvBlock module is defined as follows:

```
class ConvBlock(nn.Module).
def __init__(
self.
in_channel.
out_channel.
kernel_size.
```

```
padding.
kernel_size2=None.
padding2=None.
downsample=False.
fused=False.
).
super(). __init__()

pad1 = padding
pad2 = padding
if padding2 is not None.
pad2 = padding2

kernel1 = kernel_size
kernel2 = kernel_size
## last layer kernel_size2=4, other layers input is none
if kernel_size2 is not None.
kernel2 = kernel_size2

self.conv1 = nn.Sequential(
EqualConv2d(in_channel, out_channel, kernel1, padding=pad1).
nn.LeakyReLU(0.2).
)

if downsample.
if fused: ## For resolutions of 128 and above, use a
convolution with a step size of 2
self.conv2 = nn.Sequential(
Blur(out_channel).
FusedDownsample(out_channel, out_channel, kernel2,
padding=pad2).
nn.LeakyReLU(0.2).
)

else: ## For resolutions of 64 and below, use average
pooling
self.conv2 = nn.Sequential(
Blur(out_channel).
EqualConv2d(out_channel, out_channel, kernel2,
padding=pad2).
nn.AvgPool2d(2).
nn.LeakyReLU(0.2).
)

else.
```

```
self.conv2 = nn.Sequential(
EqualConv2d(out_channel, out_channel, kernel2,
padding=pad2).
nn.LeakyReLU(0.2).
)

def forward(self, input).
out = self.conv1(input)
out = self.conv2(out)

return out
```

Similar to the strategy of using different upsampling methods for different resolution modules in the generator, downsampling is performed using convolution with stride for resolutions of 128 and above, and downsampling is performed using average pooling for resolutions below 128, for which the reader is invited to read the full project for the detailed code.

5.9.3 Use of Pre-trained Models

Next we perform face image generation experiment, first we need to download the relevant pre-training model according to the hints in the open source project, this time we download the 1024 resolution generation model, and then use the pre-training model to generate the image.

5.9.3.1 Face Generation

First we build the predictor and generate the face, the core inference code is as follows:

```
## Building Predictors
class Predictor().
def __init__ (self,modelpath).
self.device = torch.device("cuda" if
torch.cuda.is_available() else "cpu")
self.generator = StyledGenerator(512).to(self.device) ##
Define the generator

## Load the trained model weights
weights = torch.load(modelpath,map_location=self.device)
self.generator.load_state_dict(weights["generator"])
self.generator.eval() ## Set the inference mode

## Get the average style vector
self.mean_style = get_mean_style(self.device)

## Prediction functions
```

```
def predict(self, seed, output_path).
torch.manual_seed(seed) ## set seed for CPU to generate
random numbers, making the result deterministic
step = int(math.log(SIZE, 2)) - 2
nsamples = 15
img = self.generator(
torch.randn(nsamples, 512).to(self.device).
step=step.
alpha=1.
mean_style=self.mean_style.
style_weight=0.7.
)
utils.save_image(img, output_path, normalize=True)

if __name__ == '__main__'.
modelpath = "checkpoints/stylegane-1024px-new.model"
predictor = Predictor(modelpath)
## Run 10 times to get the generated results based on
different random seeds
for i in range(0,10).
predictor.predict(i,'results/'+str(i)+'.png')
```

The generator is defined in the initialization function init, where the av erage style vector is obtained, and the g is called in the predict function to generates the image.
where the average style vector is obtained from the following function:

```
## Average style vector acquisition
@torch.no_grad()
def get_mean_style(generator, device).
mean_style = None
for i in range(100).
## From the random vector Z, after mapping network to get W
style = generator.mean_style(torch.randn(1024, 512).mean(0,
keepdim=True).to(device))
if mean_style is None.
mean_style = style
else.
mean_style += style

mean_style /= 100
return mean_style
```

In one iteration, the mean_style function of generator takes a 1 × 512-dimensional random vector Z as input and produces a 1 × 512-dimensional vector W. The average

of a total of 100 times iteration is the mean_style vector.

The generator generates *n*_samples at a time, and the input parameters include the random vector *Z*, step, alpha, and style_weight.

Where step is the upsampling count factor, which is equal to 8 when the resolution of the generated image is 1024. Since the input is a 4 × 4 sized graph, it has to be upsampled $2^8 = 256$ times.

Alpha is a layer-skip connection fusion factor for fusing features with different resolutions in different layers, and the default is 1, which means no fusion is performed.

style_weight is the truncation weight. The larger the weight, the more the generated image deviates from the average face, and a weight of 0 will generate the average face.

Figure 5.44 shows the generation result when the truncation weight is 0. It

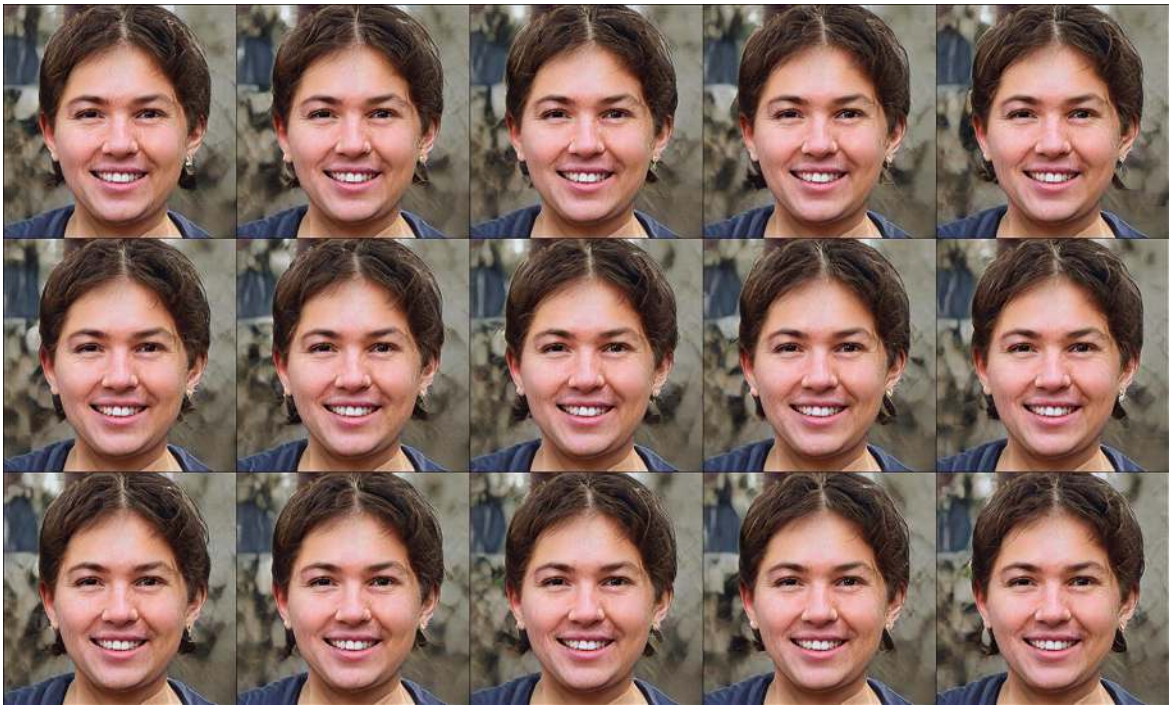


Fig. 5.44 Generated results with truncated weights of 0

can be seen that the generated body does not change, there is only slight changes in the background, which comes from the effect of the input noise added in the synthesis network. If we want to generate exactly the same face, we can fix the random seeds.

Figure 5.45 shows the generation result when the truncation weight is 0.7,

Fig. 5.45 Generated results with truncation weight of 0.7

and it can be seen that the generated subject has changed and can be generated into various real human faces.

5.9.3.2 Style Mixing Editing

StyleGAN uses style blending to provide regularization during training, and we next view the results of style blending with the following core code.

```
## Style Mixing
@torch.no_grad()
def style_mixing(generator, step, mean_style, n_source,
n_target, device).
## Two style vectors
source_code = torch.randn(n_source, 512).to(device)
target_code = torch.randn(n_target, 512).to(device)

shape = 4 * 2 ** step ##1024 resolution
alpha = 1

images = [torch.ones(1, 3, shape, shape).to(device) * -1]

## Source Domain Map
source_image = generator(
source_code, step=step, alpha=alpha, mean_style=mean_style,
style_weight=0.7
```

```
)
## Target Domain Map
target_image = generator(
target_code, step=step, alpha=alpha, mean_style=mean_style,
style_weight=0.7
)

images.append(source_image) ## Store the source domain image

## Style Mixing
for i in range(n_target).
image = generator(
[target_code[i].unsqueeze(0).repeat(n_source, 1),
source_code].
step=step.
alpha=alpha.
mean_style=mean_style.
style_weight=0.7.
mixing_range=(0, 1).
)
images.append(target_image[i].unsqueeze(0)) ## store the
target domain map
images.append(image) ##Store blended style images

images = torch.cat(images, 0)
```

In the above code, the graphs of source and target domains are first generated based on n_source, n_target, and then the respective style vectors are blended one by one.

The way of mixing is determined by the mixing_range, and you can see in the code of 5.9.2 that there are two ways of mixing. When mixing_range = (−1, 1), it is a random mixing method, i.e., the exchange point of the two vectors is chosen randomly.

When a valid range is specified, blending is done according to the valid range. For 1024 resolution images, there are nine stylized layers in total, corresponding to 4, 8, 16, 32, 64, 128, 256, 512, 1024 for a total of nine levels of resolution. Therefore, the valid range of style blending is between 0 and 8. We take (0,1) for the next style blending experiment, and according to the code we can know that it indicates that the features of 4, 8 resolutions are taken from the second style vector, and the features of 16, 32, 64, 128, 256, 512, 1024 resolutions are taken from the first style vector.

Figure 5.46 shows a graph of the results of style blending

Fig. 5.46 StyleGAN style blending results

The first row in Fig. 5.45 represents the source domain map, the first column represents the target domain map, and the other represents the style blending result map of the source domain map and the target domain. It can be seen that the style blending map retains the macro attributes such as pose, hair style, and face shape in the source domain map, and retains the micro features such as skin tone, eyes, and hair texture in the target domain map, which achieves realistic style blending and is consistent with the face feature hierarchy in Sect. 5.5.2.

5.9.4 Summary

In this section, we introduce the StyleGAN project based on the PyTorch framework implementation, experimenting with StyleGAN image generation with different parameters, and style mixing results. The core code of the generative and discriminative models of StyleGAN is explained, and the reader can refer to the project for training replication.

StyleGAN is a very important framework for image generation and also a very important framework for image editing. When projected from the image to the latent space, various attributes of the image can be edited, such as the age and expression of the face, etc. It is a core technology in current face editing applications, and it is worthwhile for readers to master it in-depth, and we will also focus on it in the later chapters.

References

1. Karras T, Laine S, Aila T. A style-based generator architecture for generative adversarial networks[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2019: 4401-4410.

2. Shrivastava A, Pfister T, Tuzel O, et al. Learning from Simulated and Unsupervised Images through Adversarial Training[C]. Computer vision and pattern recognition, 2017: 2242-2251.

3. Sbai O, Elhoseiny M, Bordes A, et al. DeSIGN: Design Inspiration from Generative Networks [J]. arXiv preprint arXiv:1804.00921, 2018.

4. Elgammal A, Liu B, Elhoseiny M, et al. CAN: Creative adversarial networks, generating "art" by learning about styles and deviating from style norms [J]. arXiv preprint arXiv:1706.07068, 2017.

5. Radford A, Metz L, Chintala S. Unsupervised representation learning with deep convolutional generative adversarial networks [J]. arXiv preprint arXiv:1511.06434, 2015.

6. Mirza M, Osindero S. Conditional generative adversarial nets [J]. arXiv preprint arXiv:1411.1784, 2014.

7. Chen X, Duan Y, Houthooft R, et al. Infogan: Interpretable representation learning by information maximizing generative adversarial nets [C]// Advances in neural information processing systems. 2016: 2172-2180.

8. Odena A. Semi-supervised learning with generative adversarial networks[J]. arXiv preprint arXiv:1606.01583, 2016.

9. Odena A, Olah C, Shlens J. Conditional image synthesis with auxiliary classifier gans[C]//International conference on machine learning. PMLR, 2017: 2642-2651.

10. Mao X, Li Q, Xie H, et al. AlignGAN: Learning to Align Cross-Domain Images with Conditional Generative Adversarial Networks [J]. arXiv: Computer Vision and Pattern Recognition, 2017.

11. Zhang H, Xu T, Li H, et al. StackGAN: Text to Photo-Realistic Image Synthesis with Stacked Generative Adversarial Networks[C]. international conference on computer vision, 2017: 5908-5916.

12. Denton E L, Chintala S, Fergus R. Deep generative image models using a laplacian pyramid of adversarial networks [C]//Advances in neural information processing systems. 2015: 1486-1494.

13. Karras T, Aila T, Laine S, et al. Progressive growing of gans for improved quality, stability, and variation [J]. arXiv preprint arXiv:1710.10196, 2017

14. Shaham T R, Dekel T, Michaeli T. Singan: Learning a generative model from a single natural image[C]//Proceedings of the IEEE/CVF international conference on computer vision. 2019: 4570-4580.

15. Perarnau G, De Weijer J V, Raducanu B, et al. Invertible Conditional GANs for image editing.[J]. arXiv: Computer Vision and Pattern Recognition, 2016.

16. Karras T, Laine S, Aittala M, et al. Analyzing and Improving the Image Quality of StyleGAN.[J]. arXiv: Computer Vision and Pattern Recognition, 2019.

17. Durugkar I, Gemp I, Mahadevan S. Generative multi-adversarial networks [J]. arXiv preprint arXiv:1611.01673, 2016.

18. Ghosh A, Kulharia V, Namboodiri V P, et al. Multi-agent diverse generative adversarial networks[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018: 8513-8521.

19. Mariani G, Scheidegger F, Istrate R, et al. BAGAN: Data Augmentation with Balancing GAN.[J]. arXiv: Computer Vision and Pattern Recognition, 2018.

P. Long, X. Guo, *Generative Adversarial Network*
https://doi.org/10.1007/978-981-96-9404-4_6

6. Image Translation

Peng Long[1] ✉ and Xiaozhou Guo[2]
(1) Beijing YouSan Educational Technology, Beijing, China
(2) • China Electronics Technology Group Corporation No. 54 Research Institute, Shijiazhuang, China

Abstract
This chapter systematically introduces image translation tasks, covering fundamental concepts, classifications, and core GAN-based methodologies. Image translation is defined as transforming images between domains (e.g., RGB to anime). Tasks are categorized into global (e.g., stylization, segmentation) and local (e.g., facial attribute editing), as well as supervised (paired data) and unsupervised (unpaired data) paradigms. Key supervised models include Pix2Pix (using U-Net generators and PatchGAN discriminators with L1 reconstruction loss), Pix2PixHD (enhancing resolution via multi-scale generators/discriminators), and Vid2Vid (video translation with optical flow constraints). Unsupervised approaches focus on domain alignment (e.g., CycleGAN with cyclic consistency loss) and latent space sharing (e.g., UNIT combining VAE and GAN). A practice on image coloring is demonstrated for deep understanding the details of Pix2Pix framework.

Keywords Image translation – Conditional GAN – Pix2Pix – CycleGAN – Domain adaptation

Image translation does not refer to a specific research area, but a collective term for a series of research fields, which commonly includes tasks such as image stylization. In this chapter, we will introduce the classical models and core techniques of image translation GAN.

6.1 Basics of Image Translation

Firstly, let's take a look at the basic concepts and applications of image translation, as well as the classification of image translation models.

6.1.1 What Is Image Translation?

Before introducing image translation, we first need to understand a basic concept of domain: a series of images with the same style collection. What image translation needs to achieve is the transformation from one domain to another, which is closely related to the domain migration task (Domain Transfer), for example, Fig. 6.1 shows the migration from RGB image domain to anime image domain.

Fig. 6.1 Example of different domain images

Since image translation is a transformation from one image to another, all kinds of image processing algorithms can be referred to as image translation algorithms, and Fig. 6.2 shows a series of classic

examples.

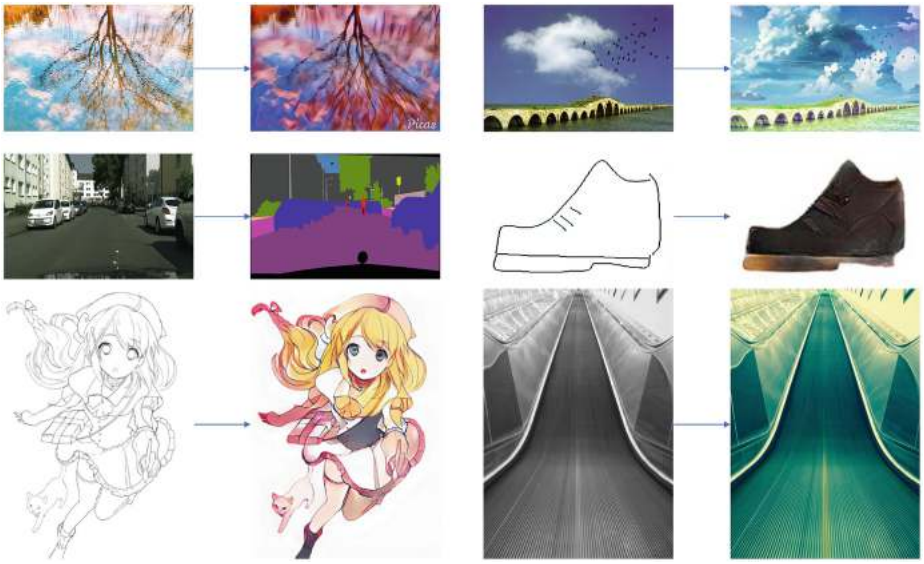


Fig. 6.2 Common image translation tasks

Figure 6.2 includes image stylization, image segmentation, image coloring, and conversion from contour map to RGB image, which are common image translation tasks.

6.1.2 Types of Image Translation Tasks

In the field of image translation, image translation tasks can be divided into global and local image translation according to the regions where the images are edited. Moreover, Image translation tasks can be classified into supervised and unsupervised tasks on the basis of whether the image translation model requires pairwise matching of images from different domains for training.

6.1.2.1 Global and Local Image Translation Tasks

Many classical image processing problems are global image translation tasks. For example, image segmentation is a transformation from the original image to a mask, and edge detection is a transformation from the original image to a binary edge.

Another example is image stylization, which is a very broad and recreational application, often used to create specific stylized images, such as different weather in photography, conversion from normal work to oil painting. Figure 6.3 shows the effect of different stylization of an image.

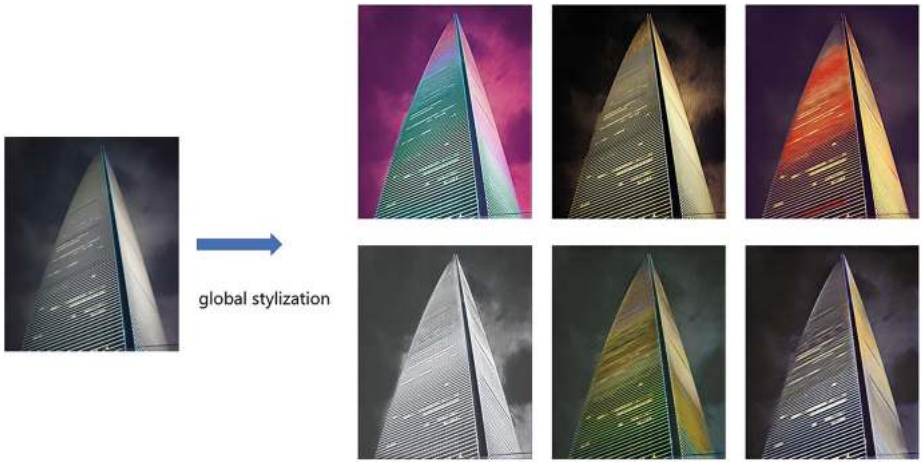


Fig. 6.3 Global image translation (image stylization)

In addition to general image styling, there are also special styling needs in some vertical fields, such as face cartoonization whose aim is to generate cartoon images from real human face images, which can be used in entertainment and social fields.

On the other hand, sometimes we only need to edit some attributes in the image, and its operation on the image is usually localized, one of the more common applications is the attribute editing of faces. Two typical local face editing tasks are shown in Fig. 6.4, which are face expression editing and face makeup.

Fig. 6.4 Partial image editing

6.1.2.2 Supervised and Unsupervised Image Translation Tasks

For some classical tasks, the prediction results must be unique, such as image segmentation, edge detection, and it is also very easy for us to obtain some paired sample data to train a supervised image translation model, as shown in Fig. 6.5.

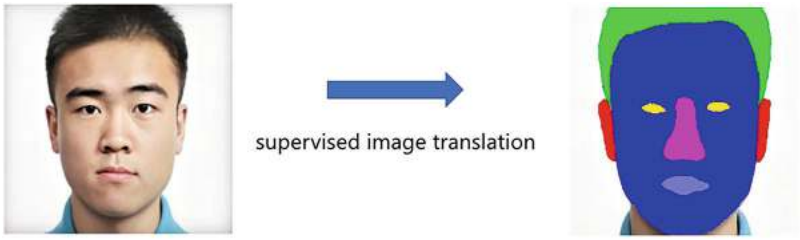


Fig. 6.5 Typical supervised image translation task (image segmentation)

However, there are some tasks where the prediction results are not unique, and even we would have preferred a richer output from the model and we cannot obtain paired data, so we need to investigate unsupervised image translation models, such as the face animation stylization in Fig. 6.6.

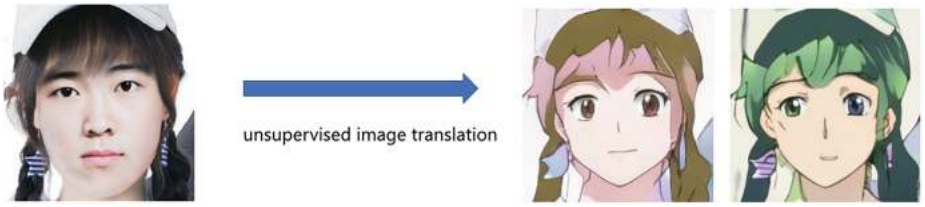


Fig. 6.6 Typical unsupervised image translation task (face stylization)

Next, we will introduce various image translation models based mainly on supervised and unsupervised classification, which are all essentially conditional GANs. Moreover, core improvement techniques will be introduced.

6.2 Supervised Image Translation Model

For supervised image translation models, the input and output images of the model are one-to-one, which also requires the datasets composed of pairs of images during training, and the representative model is Pix2Pix.

6.2.1 Pix2Pix

Pix2Pix [1] is a typical supervised image translation model GAN, which uses pairs of images to accomplish image to image translation.

In the Pix2Pix framework, the input to the generator is not random noise but a real image to be transformed and the output is the result of the transformation. In order to achieve pairwise relationships, the input and output of the generator G are fed together into the discriminator.

The architecture of Pix2Pix is shown in Fig. 6.7.

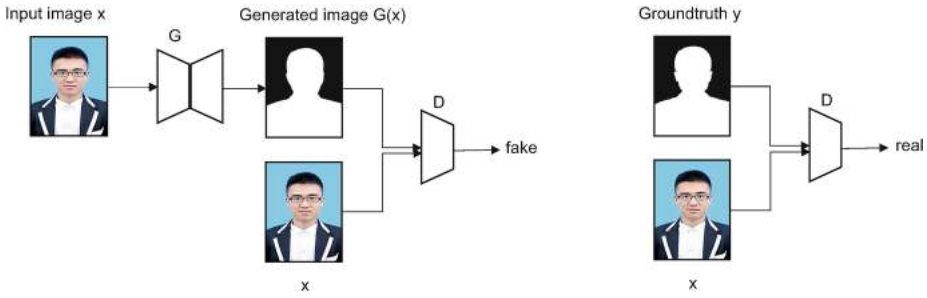


Fig. 6.7 Pix2Pix architecture

The generator uses the UNet structure. It preserves pixel-level detail information at different resolutions, which is very important to obtain good results.

The discriminator is PatchGAN, which does not directly predict a true/false probability for the whole image, but predicts subgraphs of $N \times N$ region size instead, and finally averages the predictions of the subgraphs, Fig. 6.8 shows the schematic diagram of PatchGAN.

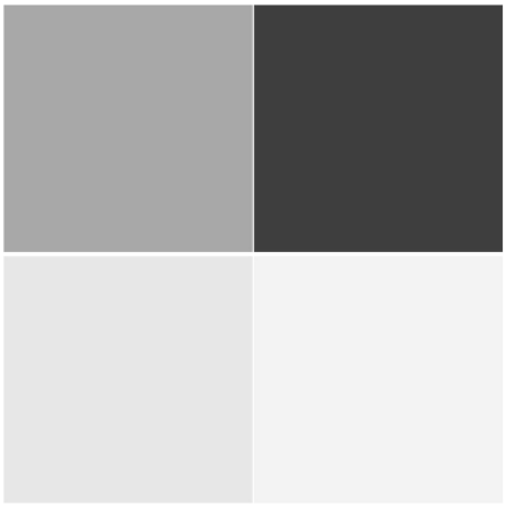


Fig. 6.8 PatchGAN prediction schematic, the original graph is divided into 2 × 2 regions for prediction

With the use of PatchGAN, GAN can supervise more high-frequency image detail information. In the content of Pix2Pix, it compares the generation results using different size image blocks, and for a 256 × 256 input image, the image block size of 70 × 70 has better details than using the whole image as discriminator input.

GAN can encourage the generation of high-frequency components, while the generation of low-frequency components is modeled by using reconstruction loss, and the full loss function of Pix2Pix is composed of a standard CGAN loss and the L1 reconstruction loss, as shown in Eq. (6.1):

$$G^* = \arg\min_G \max_D \mathscr{L}_{cGAN}(G, D) + \lambda \mathscr{L}_{L1}(G) \tag{6.1}$$

where the reconstruction loss is the L1 loss, as in Eq. (6.2), which is more beneficial to reduce the blurring of the reconstructed image than the L2 loss:

$$\mathscr{L}_{L1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x,z)\|_1] \tag{6.2}$$

The conditional GAN loss is shown in Eq. (6.3):

$$\mathscr{L}_{cGAN}(G, D) = \mathbb{E}_{x,y}[\mathrm{logD(x, y)}\ ] + \mathbb{E}_{x,z}[\log\ (1 - \mathrm{D(x, G(x, z))}\ ] \tag{6.3}$$

z is the noise, which does not appear in the model structure schematic diagram of Pix2Pix. It can be used as an additional input variable as a new input after channel stitching with the input image. Additionally, the author of Pix2Pix achieved a similar effect by using the Dropout layer.

The Pix2Pix framework can be used for most image translation tasks, including style migration, grayscale and color map conversion, image segmentation, edge detection, image enhancement, etc.

6.2.2 Pix2PixHD

Pix2PixHD (High Precision Pix2Pix) [2] is an improvement of Pix2Pix, which uses multi-scale generators and discriminators, etc. so as to generate 2048 × 1024 high-resolution images, solving the problem of unstable high-resolution image generation, and its structure is shown in Fig. 6.9.

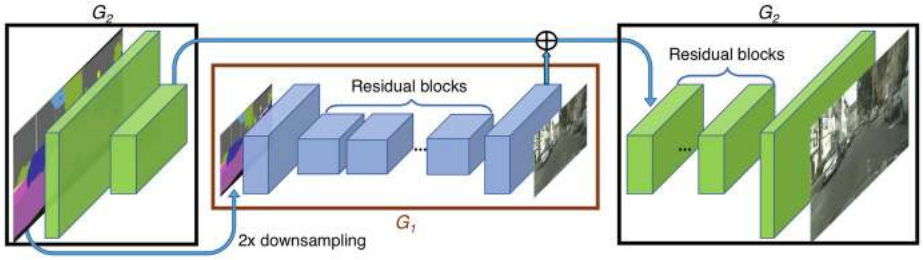


Fig. 6.9 pix2pixHD structure

The generator consists of two parts, G1 and G2. There is no difference between G1 and the Pix2Pix generator, which is a standard U-Net architecture.

G2, on the other hand, is an upsampling model whose left half part extracts features and then sums and fuses them with the previous layer of features in the output layer of G1, feeding the fused information into the second half of G2 to output a high-resolution image. G1 is trained first, and then G1 and G2 are trained jointly.

The discriminator also applies the PatchGAN architecture, but uses a multi-scale discriminator, i.e., discriminates the generator at three different scales and then averages the results.

The three scales used for discrimination are: the original image, 1/2 downsampled image of the original image, and 1/4 downsampled image of the original image. The lower the resolution of the graph, the larger the perceptual field, and the more concerned about the global consistency of the image.

The loss function contains two components, as shown in Eq. (6.4), which are GAN loss and Feature matching loss, respectively.

$$\min_{G}\left(\left(\max_{D_1,D_2,D_3}\sum_{k=1,2,3}\mathscr{L}_{GAN}(G, D_k)\right)\right) + \lambda\sum_{k=1,2,3}\mathscr{L}_{FM}(G, D_k) \tag{6.4}$$

where GAN loss$\mathscr{L}_{GAN}$ is the same as Pix2Pix, while for the feature matching loss$\mathscr{L}_{FM}$, the generated samples and the real samples are fed into the discriminator separately to extract features, and then the L1 distance is calculated for the features, and this loss is beneficial to make the training process more stable.

In addition, an additional content loss can be added to further improve the generation quality: that is, the generated and real samples are fed into the VGG16 model separately to extract the middle layer features, and then the L1 distance is calculated.

Feature matching loss and content loss have the same form, except that the networks used to compute them differ in that they can constrain the overall content of the model-generated and input graphs to be consistent, while the details are learned by the GAN.

6.2.3 Vid2Vid

Although GAN has achieved good results in the field of image translation, however, there are still many problems in the field of video translation. The main reason is that the generated video is difficult to

guarantee the consistency of the front and back frames and is prone to jitter. For this problem, the most intuitive idea is to add the optical flow information of the front and back frames as a constraint.

Vid2Vid [3] is the video version of Pix2PixHD, which adds optical flow information to the discriminator, models foreground and background separately, and focuses on solving the inconsistency of front and back frames in the video-to-video conversion process.

Vid2Vid is built on the pix2pixHD model and therefore allows high-resolution video generation.

6.3 Unsupervised Image Translation Model

One-to-one image translation requires high-quality one-to-one datasets, which is often not available, so we need unsupervised models. In this section, we will introduce several representative models.

6.3.1 Unsupervised Model Based on Domain Migration and Domain Alignment

Let's first introduce the unsupervised models based on domain migration and domain alignment, both of which at their core are about transforming images into a uniform feature space.

6.3.1.1 Domain Migration Network

Domain Transfer Network (DTN) [4] adopts the idea of style migration to complete the transformation from one domain to another, and the model maps the input to the same domain no matter what domain it comes from.

The network structure is shown in Fig. 6.10. Given two domains S and T, we hope to learn a generating function G, which consists of an encoder f and a decoder g. f is used to extract features from the input image to obtain a feature vector. The input of g is the output of f, and the output is a target-style image. The goal to be achieved is that whether the input of f comes from the domain S or T, f can encode it into the information of the target domain.

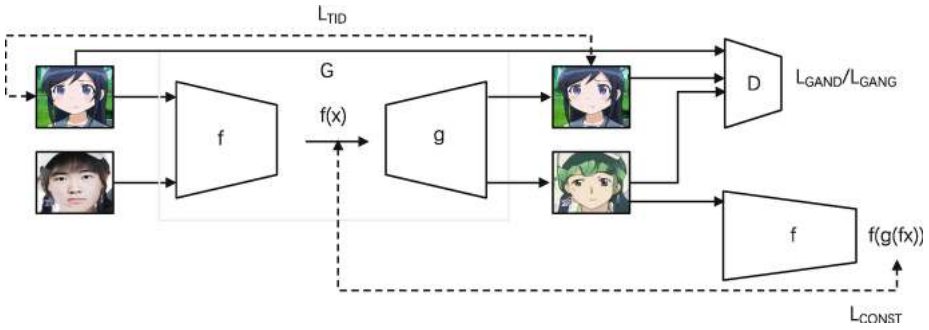


Fig. 6.10 Schematic diagram of DTN structure

The original image and the target image do not need to correspond to each other during training, and the original image dataset and the target-style image dataset can be used for training, respectively.

For the image in the source domain, it is desired that the feature vectors extracted by the input f and the feature vectors generated by the original image through the generative network G and then extracted by f are as similar as possible, constructing a loss function as L_{CONST}, where $x \in$ s indicates that the image x is the original image and s is the set of original images.

$$L_{\text{CONST}} = \sum_{x \in s} d\Big(f(x), f(g(f(x))) \tag{6.5}$$

For the image in the target domain, when it is input to the generative network G, the output should also be the same image, i.e., the generative network plays the role of constant mapping (identity matrix) for the target image, constructing the loss function L_{TID}, where $x \in t$, means the image x is the target image and t are the set of target images.

(6.6)

$$L_{\mathrm{TID}} = \sum_{x \in t} d_2(x, G(x))$$

DTN network also contains a discriminant network D. The role of the discriminant network is to discriminate whether the input is a generated image (fake) or an input image (real), and what needs to be discriminated includes the generated image of the original image, the target image and the generated image of the target image, and the loss function is Pay attention to the size and format of parentheses:

$$L_D = -\sum_{x \in S} \log D_1\left(g(f(x))\right) - \sum_{x \in t} \log D_2\left(g(f(x))\right) - \sum_{x \in t} \log D_3(x) \tag{6.7}$$

where D_1 is used to discriminate the generated image of the original image which is passing through the generation network G, and D_2 is used to discriminate the generated image of the target image passing through the generative network G, and D_3 for discriminating the target image, where D is a triple classifier.

Adding a TV smoothing loss, the total generating network G has a loss function as shown in Eq. (6.8):

$$L_G = L_{\mathrm{GANG}} + \alpha L_{\mathrm{CONST}} + \beta L_{\mathrm{TID}} + \gamma L_{\mathrm{TV}} \tag{6.8}$$

Here $L_{\mathrm{GANG}} = -\sum_{x \in S} \log D_3\big(g(f(x))\big) - \sum_{x \in t} \log D_3\big(g(f(x))\big)$

To train the specific model, we first pre-train f, which is a classification network in the source and target domains, and after training, a excellent feature extraction network is obtained, and then the whole network is trained.

Unsup-Im2Im [5] is also a similar framework with a three-step training process, as shown in Fig. 6.11.

(1) First train a GAN network.

(2) Then fix the generator G of GAN and train an encoder E.

(3) Transformation with the trained encoder E and generator G.

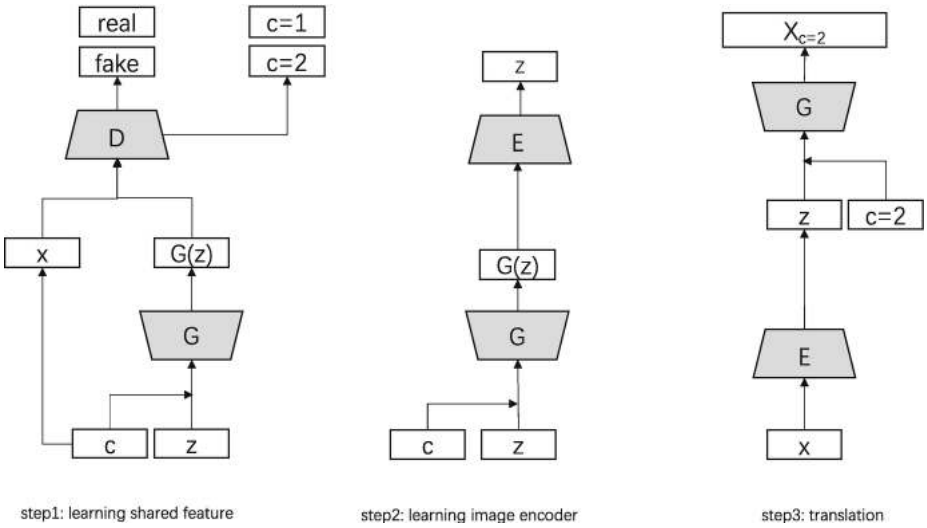


Fig. 6.11 Unsup-Im2Im Framework

The objective functions of the discriminator and generator are

$$L_D = -\log\left(P(s|D(X_{\mathrm{real}}))\right) + \log\left(1 - P(s|D(X_{\mathrm{fake}}))\right) + \log\left(P(c|D(X_{\mathrm{real}}))\right) \tag{6.9}$$

$$L_G = \log\left(P(s|D(X_{\mathrm{fake}}))\right) + \log\left(P(c|D(X_{\mathrm{fake}}))\right) \tag{6.10}$$

where s denotes the score of images as real images, c denotes the category, X_{real} is the real image, and X_{fake} is the generated image, and it can be seen that the classification loss function is added to the GAN.

The input of generator G is the category information and random vector *Z*. The input of encoder E is the image generated by generator *G*, and the output is the input vector expected to be reconstructed out of generator *G*. This allows extracting the feature vector of an image when it is given and the semantic information represented by that feature vector is the same as the semantic information expressed by the *Z* vector.

After training the encoder *E* and the generator *G* network, given a image to be converted by the encoding feature *Z* extracted by the encoder, and then input *Z* together with the target label that we want to convert into the generator *G* to generate the target converted image.

6.3.1.2 Domain Alignment Network

To realize the conversion from domain *D*1 to domain *D*2, the feature encoder e1 corresponding to domain *D*1 is first used for encoding, and then the feature decoder *d*2 corresponding to domain *D*2 is used for decoding. However, if each of them is learned independently, the feature encoder *e*1 and the feature encoder *e*2 may be relatively different, for example, different channels and convolutional layers are used to extract the same attributes, and then they are prone to decode wrong results when fed to their respective decoders.

Couple-GAN [6] employs a weight sharing constraint strategy to solve this problem, and its structure is shown in Fig. 6.12.

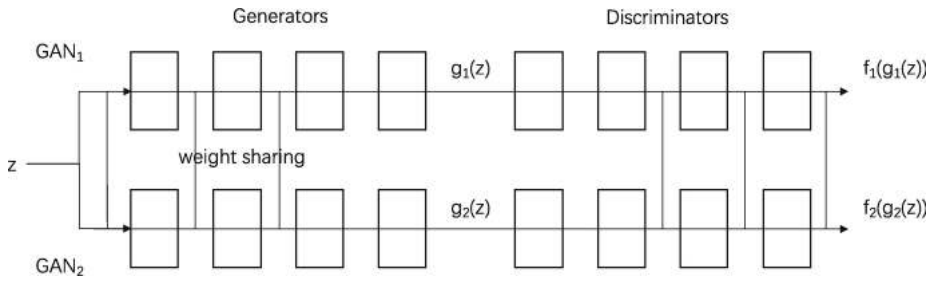


Fig. 6.12 Couple-GAN

In traditional domain adaption, we need to learn or train a domain adapter, and the training process requires supervised learning with images from the source domain and the corresponding target domain. While Couple-GAN can learn the joint distribution between them in an unsupervised manner without the existence of corresponding images in both domains.

In Fig. 6.12, GAN1 and GAN2 are GANs for two domains, respectively, which have the same structure. Meanwhile, both of their respective generators and discriminators share the weights of several network layers.

The shared weight of the generator (encoder) is near the front end of the network, because the encoder generates details step by step, and it is at the front end of the network that higher level semantic information, such as target contours, is obtained, while details such as edge textures are obtained at the back end of the network. The shared weight of the discriminator (decoder) is near the back end of the network because it is at the back end of the network that higher level semantic information is acquired for discriminating.

In contrast to training these two GANs directly, Couple-GAN obtains the joint distribution of the two domains instead of the inner product of the two marginal distributions. The specific optimization objectives are the same as those of most GANs. Later the authors extended Couple-GAN by proposing the UNIT framework [7], the principle of which is shown in Fig. 6.13.

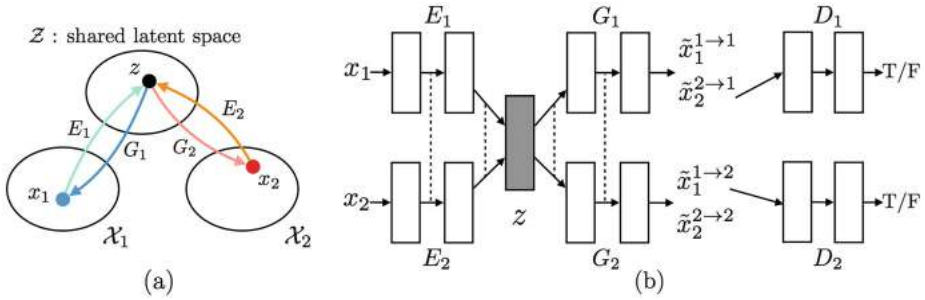


Fig. 6.13 UNIT Framework

The UNIT framework consists of two encoders, two generators, and two discriminators. The two encoders first map *X*1 and *X*2 to a shared latent space, and then input them to the generator and discriminator, respectively.

Figure 6.13a shows a pair of images (*x*1, *x*2) from two different image domains that can be mapped into the same latent code *z* in a shared latent space *Z*.

Figure 6.13b in *E*1 and *E*2 are two encoders responsible for encoding the image into latent code *z*. *G*1 and *G*2 are two generating functions responsible for converting the latent code *z* into an image.

The UNIT framework is identical to the Couple-GAN architecture and uses a weight sharing strategy to share the latent space by sharing the latter layers of *E*1 and *E*2 and sharing the first layers of *G*1 and *G*2, both for extracting high-level features.

UNIT is a framework that incorporates VAE and GAN, where *E* and *G* form the VAE and *G* and *D* form the GAN, requiring the following optimization objectives for the solution, including VAE reconstruction loss, GAN loss, and loop loss, as in Eq. (6.10).

$$\min_{E_1,E_2,G_1,G_2,D_1,D_2} \max \mathcal{L}_{VAE_1}(E_1,G_1) + \mathcal{L}_{GAN_1}(E_1,G_1,D_1) + \mathcal{L}_{CC_1}(E_1,G_1,E_2,G_2) + \mathcal{L}_{VAE_2}(E_2,G_2) + \mathcal{L}_{GAN}$$

where the loss of VAE is defined in Eqs. (6.11) and (6.12) below:

$$\mathcal{L}_{VAE_1}(E_1,G_1) = \lambda_1 KL(q_1(z_1|x_1)\|p_\eta(z)) - \lambda_2 \mathbb{E}_{z_1\sim q_1(z_1|x_1)}[log\ p_{G_1}(x_1|z_1)] \tag{6.12}$$

$$\mathcal{L}_{VAE_2}(E_2,G_2) = \lambda_1 KL(q_2(z_2|x_2)\|p_\eta(z)) - \lambda_2 \mathbb{E}_{z_2\sim q_2(z_2|x_2)}[log\ p_{G_2}(x_2|z_2)] \tag{6.13}$$

The first of these terms is the KL divergence, which is the divergence of the posterior distribution with respect to the prior distribution, and the latter term is actually the image reconstruction loss.

The loss of GAN is defined as follows:

$$\mathcal{L}_{GAN_1}(E_1,G_1,D_1) = \lambda_0 \mathbb{E}_{x_1\sim Px_1}[\log D_1(x_1)] + \lambda_0 \mathbb{E}_{z_2\sim q_2(z_2|x_2)}[l\,\mathrm{og}(1 - D_1(G_1(z_2)))] \tag{6.14}$$

$$\mathcal{L}_{GAN_2}(E_2,G_2,D_2) = \lambda_0 \mathbb{E}_{x_2\sim Px_2}[\log D_2(x_2)] + \lambda_0 \mathbb{E}_{z_1\sim q_1(z_1|x_1)}[\log (1 - D_2(G_2(z_1)))] \tag{6.15}$$

CC loss is defined as follows:

$$\mathcal{L}_{CC_1}(E_1,G_1,E_2,G_2) = \lambda_3 KL(q_1(z_1|x_1)\|p_\eta(z)) + \lambda_3 KL\left(q_2\left(z_2\middle|x_1^{1\to2}\right)p_\eta(z)\right) - \lambda_4 \mathbb{E}_{z_2\sim q_2}\left(z_2\middle|x_1^{1\to2}\right)[\log p_{G}$$

Formula 6.16 corresponds to the transformation from domain 2 to domain 1, i.e. $F_{2\to1}(x_2) = G_1(z_2\sim q_2(z_2|\,x_2)$

$$\mathcal{L}_{CC_2}(E_2,G_2,E_1,G_1) = \lambda_3 KL(q_2(z_2|x_2)\|p_\eta(z)) + \lambda_3 KL\left(q_1\left(z_1\middle|x_2^{2\to1}\right)p_\eta(z)\right) - \lambda_4 \mathbb{E}_{z_1\sim q_1}\left(z_1\middle|x_2^{21}\right)[log\ p_{G_2}$$

Formula 6.17 corresponds to the transformation from domain 1 to domain 2, $F_{1\to2}(x_1) = G_2(z_1\sim q_1(z_1|\,x_1)$The training is conducted using a two-step strategy:

Step 1: fix E1, E2, G1, G2, train D1, D2;
Step 2: fix D1, D2, train E1, E2, G1, G2

6.3.2 Unsupervised Model Based on Circular Consistency Constraints

CycleGAN [8] is an image translation framework that allows one-to-one mapping between source and target domains without establishing a one-to-one mapping between training data.

The method performs a two-step transformation of the source domain image: first attempting to map it to the target domain, and then returning to the source domain to obtain a secondary generated image, which is a cyclic structure, hence called CycleGAN, and the schematic diagram of the framework is shown in Fig. 6.14.

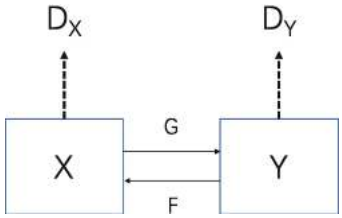


Fig. 6.14 Schematic diagram of Cycle Gan principle

From Fig. 6.14 we can see that CycleGAN is actually two one-way GANs with opposite directions, which share two generators and then each has a discriminator, adding up to two discriminators and two generators. A one-way GAN has two losses, and CycleGAN has four losses.

X and *Y* denote the images of two domains, respectively, and two generators *G* and *F* are needed for the generation from *X* to *Y* and *Y* to *X* to generation, respectively, containing two discriminators, D_x and D_y. The complete loss is as follows:

$$L(G,F,D_X,D_Y)=L_{\text{GAN}}(G,D_Y,X,Y)+L_{\text{GAN}}(F,D_X,X,Y)+\lambda L_{\text{CYC}}(F,G) \tag{6.18}$$

where $L_{\text{GAN}}(G,D_Y,X,Y)$, $L_{\text{GAN}}(F,D_X,X,Y)$ is the loss of the ordinary GAN, and $L_{\text{CYC}}(F,G)$ is as follows:

$$L_{\text{CYC}}(F,G)=E_{x\sim p_{\text{data}(x)}}[||F(G(x))-x||_1]+E_{y\sim p_{\text{data}(y)}}[||G(F(y))-y||_1] \tag{6.19}$$

The meaning behind this is that after a sample is transformed from one space to another, it can conversely be transformed back, i.e.,

$$\text{x}\rightarrow G(x)\rightarrow F(G(x))\approx x \tag{6.20}$$

Cyclic loss is necessary because with a large enough sample size, the network can map the same set of input images to any random arrangement of images in the target domain, so that there is no guarantee that the input *X* can be mapped to the desired output *Y* theoretically. For example, mapping a horse in one pose to a zebra in another pose is not what we hope because we only hope to change the texture style and not improve the hors's pose.

The authors of CycleGAN found that the discriminator is not stable to train if it is logarithmic loss, so they used a mean squared loss LSGAN, which is defined as follows:

$$L_{\text{LSGAN}}(G,D_Y,X,Y)=E_{y\sim p_{\text{data}(y)}}[||D_Y(y)-1||_2]+E_{x\sim p_{\text{data}(x)}}[||D_Y(G(x)||_2] \tag{6.21}$$

In addition, when we input *X* into *F*, or *Y* into *G*, the input image and the target image are in the same domain, no style conversion should be performed at this point, and the input image content and style should be maintained as much as possible, so we define the identity consistency loss, which is shown as follows:

$$L_{\text{identity}}(G,F)=E_{y\sim p_{\text{data}(y)}}[||G(y)-y||_1]+E_{x\sim p_{\text{data}(x)}}[||F(x)-x||_1] \tag{6.22}$$

In addition to cycleGAN, there are several very similar networks from the same period, including DualGAN [9], DiscoGAN [10], and XGAN [11].

DiscoGAN uses the same loss function as CycleGAN, but the specific generator structure and discriminator structure are different. Instead of using the U-Net structure in Pix2Pix as the generator and the PatchGAN structure as the discriminator, it uses a simple structure.

The difference between DualGAN and CycleGAN lies in the loss function, which is used in WGAN instead of the cross-entropy used in standard GAN.

XGAN is also similar to CycleGAN, named for its model structure like "X", as shown in Fig. 6.15.

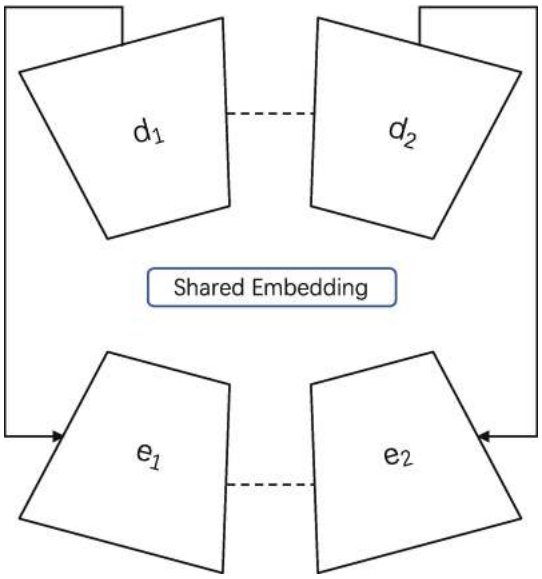


Fig. 6.15 XGAN model structure

In Fig. 6.15, e1 and e2 denote the two encoders, d1 and d2 denote the two decoders, and the shared Embedding in the middle denotes the shared feature representation. If we use $D1$ and $D2$ to represent two domains, the operation which is required to achieve the transformation from domain $D1$ to $D2$ is to first encode using e_1 and then decode using d_2.

The strategy of weight sharing is used in the last layers of the encoder and the first layers of the decoder, thus making it possible to retain as much information as possible about the high-level features of the image during the conversion process, and the converted result retains the basic features of the input image to the maximum extent. In the conversion experiments of real faces and cartoon faces done by the authors in the paper, the features such as hair, nose, and eyes of the converted faces essentially do not change too much.

XGAN uses semantic consistency loss to retain feature information at the semantic level of the image, while CycleGAN focus on pixel-level consistency loss, so XGAN can retain higher level feature information and generate images with better results.

The design of the XGAN model loss function contains five terms:

$$L_{\mathrm{XGAN}} = L_{\mathrm{rec}} + \omega_d L_{\mathrm{dann}} + \omega_s L_{\mathrm{sem}} + \omega_g L_{\mathrm{gan}} + \omega_t L_{\mathrm{teach}} \tag{6.23}$$

Reconstruction loss term L_{rec}: It is used to constrain the encoder to learn meaningful feature representations. It is desired that the reconstructed image obtained by encoding the feature information from a domain into the decoder is as close as possible to the input image.

$$L_{\mathrm{rec},1} = E_{x\sim p_{D1}}[\|F(d_1(e_1(x)) - x)\|_2] \tag{6.24}$$

Domain adversarial loss L_{dann}: It makes the embedded feature information learned by e_1, e_2 distributed in the same subspace. If the images processed by each encoder can still be discriminated by the classifier, it means that the encoding contains domain information rather than just feature information; on the contrary, if it cannot be discriminated, it means that the encoding is all feature information common to both domains. Therefore, the classifier hopes to maximize the classification accuracy, and e_1 and e_2 hope to minimize it, and a confrontation is formed between them.

$$L_{\text{dann}} = E_{x\sim p_{D1}} l(c_{\text{dann}}(e_1(x)), 1) + E_{x\sim p_{D2}} l(c_{\text{dann}}(e_2(x)), 2) \tag{6.25}$$

Here $l(\cdot)$ is the classification loss; it usually employs the cross-entropy. L_{dann} allows the model to accelerate the convergence of the model at the beginning of training by bringing the encoding of different domains closer together.

Semantic consistency loss L_{sem}: It requires that the encoding should be consistent at the semantic feature level so that the identity information of the face can be preserved, where the loss in the direction from domain 1 to 2 is defined as follows. This constraint has a higher semantic level and is easier to optimize compared to the pixel-level loss as follows:

$$L_{\text{sem},1\to 2} = E_{x\sim p_{D1}} \| (e_1(x) - e_2(g_{1\to 2}(x)) \|_2 \tag{6.26}$$

Guided Loss L_{teach}: This is an optional term that allows the use of prior knowledge to accelerate the training of the model and can be seen as a way of regularizing the learned Shared Embedding features, which is expressed as follows:

$$L_{\text{teach}} = E_{x\sim p_{D1}} \| (T(x) - e_1(x)) \|_2 \tag{6.27}$$

where T denotes the feature vector obtained based on the output layer of an already learned model. For instance, when a face recognition model is used, the guiding loss is equivalent to migrating the knowledge learned by the face recognition model to the encoder.

6.4 Key Improvements to the Image Translation Model

We have introduced the supervised and unsupervised classical image translation models, but these models cannot meet the needs of all kinds of image translation tasks, for example, Pix2Pix and CycleGAN can only achieve the conversion between two domains. In addition, in order to better edit the local details of the image, we often need some prior knowledge input to the model in order to obtain more ideal image translation results. These are the key improvement techniques for image translation models, and this section introduces the important parts of them.

6.4.1 Multi-Domain Transformation Network GAN

If we use Pix2Pix and CycleGAN to achieve interconversion between C domains, we need to learn $C \times (C-1)$ models, which is very inefficient, and in this section we introduce more suitable multi-domain conversion networks.

6.4.1.1 StarGAN v1

StarGAN [12] presents a better solution for conversion between multiple domains for a single model, and a comparison with multiple models is shown schematically in Fig. 6.16, which is also referred to as StarGAN v1.

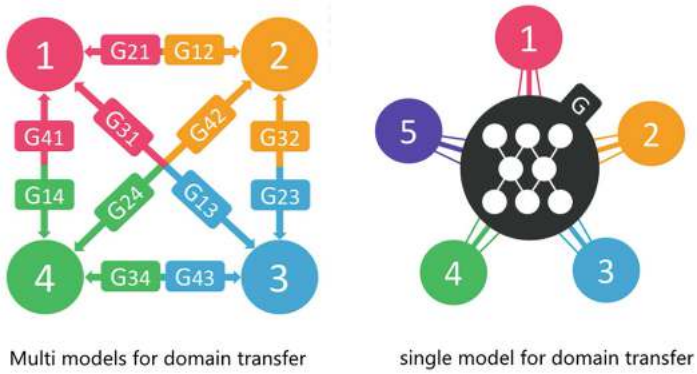


Fig. 6.16 Comparison of migration between multi-model (left) and single-model (right) implementations of multiple domains

StarGAN v1 implements migration between multiple domains by adding control information of the domains as conditional inputs. In the network structure design, the discriminator not only needs to learn to identify whether the sample is real or not, but also to judge which domain the real image comes from, and the StarGAN model architecture is shown in Fig. 6.17.

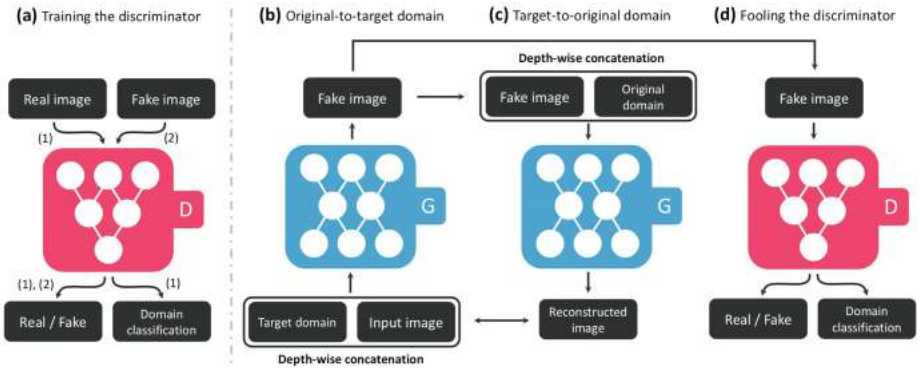


Fig. 6.17 StarGAN network structure

The processing flow of the whole network consists of three parts.

Part 1: The input image *x* and the target domain vector *c* are stitched at the channel level and fed into the generative network *G* to obtain the generative graph, with c often being a unique thermal encoding vector.
Part 2: The generated and real images are fed separately to the discriminator *D*. D needs to determine whether the picture is real or not and also which domain it comes from.
Part 3: Similar to CycleGAN, the generated generative image and the domain information c' of the original image are stitched at the channel level and input to the generator *G*. This requires the ability to reconstruct the original input picture *x*, i.e., to achieve the consistency constraint.

In addition to the basic GAN loss, domain classification loss and reconstruction loss are also included. The domain classification loss using the real image in the discriminator is defined as follows:

$$L_{cls}^{r} = E_{x,c'}\left[-\log D_{cls}\left(c'|x\right)\right] \tag{6.28}$$

In the generator, the domain classification loss of the generated image is used, defined as follows:

$$L_{cls}^{f} = E_{x,c}[-\log D_{cls}(c|G(x,c))] \tag{6.29}$$

In order to make the generator G change only the attribute information related to the domain instead of the content of the image, a reconstruction consistency loss needs to be added, which is defined as follows:

$$\mathrm{L}_{rec} = E_{x,c,c'}\left[x - G\left(G(x,c),c'\right)_1\right] \tag{6.30}$$

6.4.1.2 StarGAN v2

StarGAN v1 can only rely on explicit labels to control style migration, but the style of the data itself is very complex, and there are also coupling relationships between various attributes.

StarGAN v2 [13] refers the idea of stylized module in StyleGAN on the basis of StarGAN to achieve more complex multi-domain migration by adding style networks and removing attribute labels, and its overall model structure is shown in Fig. 6.18.

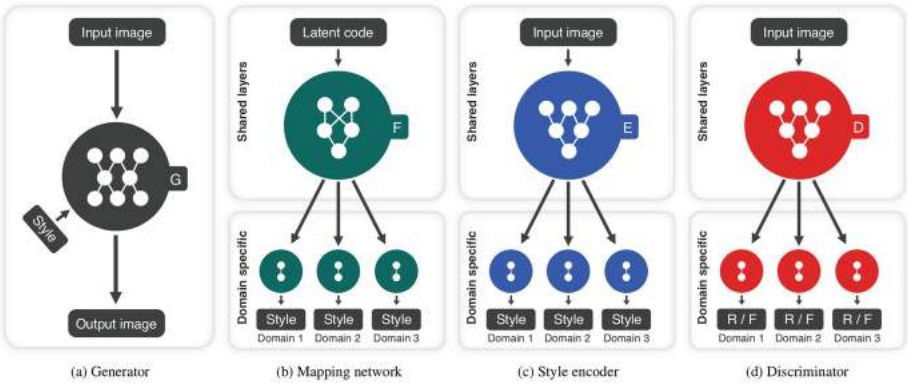


Fig. 6.18 StarGAN V2 network structure

Four sections are included in Fig. 6.18 as follows:

(1) Generator. The input consists of two parts, one is the image and the other is the style vector (style code) of a domain. The generator network structure consists of an encoder and a decoder, where the encoder extracts the image features and the decoder is responsible for fusing these image features with the data distribution of the style code and outputting $G(x; s)$.

(2) mapping network (mapping network), the output contains multiple branches. It encodes random Gaussian noise and generates a style vector of multiple domains, $s = F_y(z)$, sampling one domain y at a time at random for training.

(3) Style encoder (Style encoder), the output contains multiple branches. It performs style extraction on the input image to obtain the style vector corresponding to the domain, $s = E_y(x)$, and selects only the corresponding branch for training each time.

(4) Discriminator, the output contains multiple branches, each of which determines whether the sample belongs to the current domain.

The entire optimization objective consists of four parts:

(1) Contrast loss. Suppose the input image is x, the domain label is y, and s is the style based on the domain label acquisition, the loss is as follows. The discriminator classifies the domain correctly, while the generator learns the confusion discriminator based on s and x.

$$L_{\text{adv}} = E_{x,y}[\log D_y(x)] + E_{x,\tilde{y},z}\left[\log\left(1 - D_{\tilde{y}}(\mathrm{G}(x,\tilde{s}))\right)\right] \tag{6.31}$$

(2) Style reconstruction loss, allowing style encoders to learn the encoding of styles.

$$L_{sty} = E_{x,\tilde{y},z}\left[\|\tilde{s} - E_{\tilde{y}}(G(x,\tilde{s}))\right] \tag{6.32}$$

(3) Loss of style diversity. Randomly sampled vectors z1 and z2 are encouraged to produce widely varying styles, ensuring the richness of the generated results.

$$L_{ds} = E_{x,\tilde{y},z_1,z_2}[\|\mathrm{G}(\mathrm{x},\widetilde{s_1}) - \mathrm{G}(\mathrm{x},\widetilde{s_2})\|] \tag{6.33}$$

(4) Cyclic consistency loss. It is used to constrain some features that should not be changed as the domain changes, such as identity features, pose features.

$$L_{cyc} = E_{x,y,\tilde{y},z}\left[\left|\left|x - G\left(G(x,\tilde{s}),\hat{s}\right)\right|\right|\right] \tag{6.34}$$

6.4.2 Enriching the Generation Mode of Image Translation Model

The Pix2Pix framework can accomplish the transformation from one domain to another, but the result is unique, i.e., different steganographic features are mapped to the same output, and no texture style-rich results can be obtained, which can also be called pattern collapse because the multimodal mapping from high-dimensional input to high-dimensional output distributions is very challenging.

If the model hopes to generate richer results, it needs to learn a low-dimensional latent code for each possible output that is not present in the input image and thus is not constrained by the input image. When used, the generator inputs the image and a randomly sampled latent code, thus generating pattern-rich results.

Next we present two representative frameworks.

6.4.2.1 MUINT Framework

The MUINT framework [14] implements independent learning of content and style by using two independent encoders. It is assumed that the latent space in which the image is located can be decomposed into content space and style space, and the content code is a high-dimensional spatial feature map with complex distribution properties, while the style code is a low-dimensional vector with Gaussian distribution properties. The content of the content space is directly shared by different domains, but the content in the style space is unique to each particular domain, and the framework is shown in Fig. 6.19.

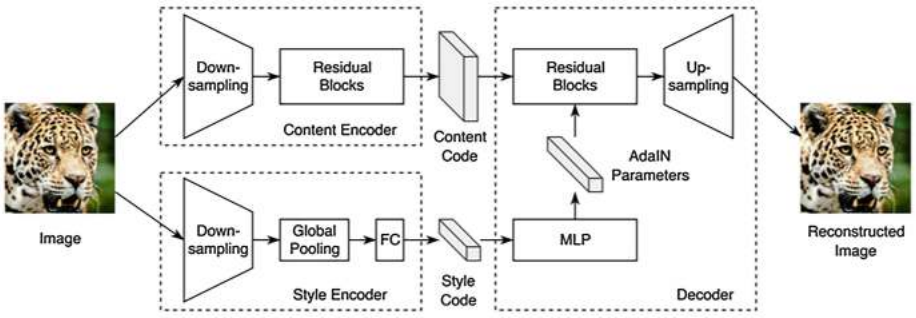


Fig. 6.19 Schematic diagram of MUINT framework

The two corresponding pairs of images (x_1, x_2) are generated as follows:

$x_1 = G_1(c,s_1)$, $x_2 = G_2(c,s_2)$, where c,s_1,s_2 are from some specific distributions and G_1, G_2 are latent generators.

In the specific implementation, the Instance Normalization layer is added after all the convolution layers of Content encoder. Because the Instance Normalization layer removes style information from the original image features, the Instance Normalization layer is not used in the Style encoder.

The decoder adds an Adaptive Instance Normalization (AdaIN) layer after each Residual Blocks, the parameters of which are learned by the MLP.

6.4.2.2 Augmented CycleGAN framework

CycleGAN can only achieve one-to-one mapping, and although we can influence the output by adding random noise perturbations, the circular consistency constraint will still make the generated results of pattern less rich.

Augmented CycleGAN [15] is improved based on CycleGAN and enables many-to-many mapping by using latent encoding as explicit input, which is shown schematically in Fig. 6.20 in comparison with CycleGAN.

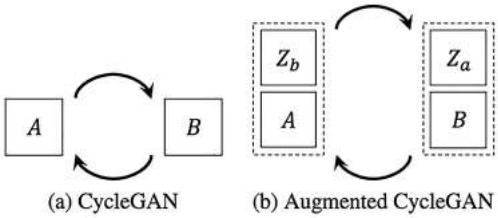


Fig. 6.20 Schematic diagram of CycleGAN compared with Augmented CycleGAN framework

The Augmented CycleGAN framework takes sample A of the source domain and latent code Z_b as input, and then outputs sample B of the target domain and latent code Z_a. A and Z_b together form the augmented source domain, so it is called Augmented CycleGAN.

The model includes eight sub-networks.

(1) Two generators G_{AB} and G_{BA}. G_{AB} input (A,Z_b), output B domain image. G_{BA} input (B,Z_a), output A domain image, they both add the vector Z as input to the image, which can be regarded as a conditional GAN.

(2) Two encoders E_A and E_B. E_A input (A,B), output A domain latent code Z_a, E_B input (A,B), output B domain latent code Z_b.

The expression is as follows:

$$\tilde{b} = G_{AB}(a, z_b) \ , \quad \widetilde{z_a} = E_A\left(a, \tilde{b}\right) \tag{6.35}$$

$$\tilde{a} = G_{BA}(b, z_a) \ , \quad \widetilde{z_b} = E_B(b, \tilde{a}) \tag{6.36}$$

The specific learning relationships are shown in Fig. 6.21.

(3) Four discriminators, corresponding to the image and encoding of domain A, and the image and encoding of domain B, respectively.

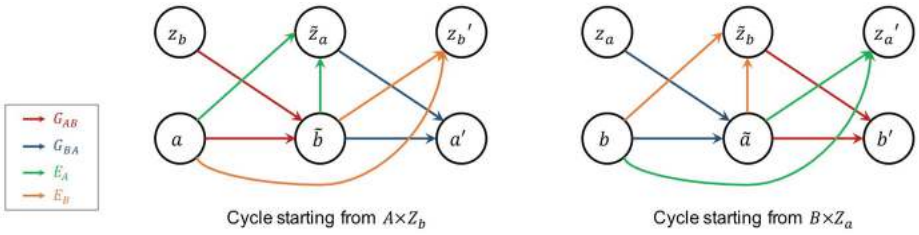


Fig. 6.21 Augmented CycleGAN cycle learning schematic

Please read the original paper to understand the specific optimization goals of MUINT framework and Augmented CycleGAN, here we focus on the model design ideas.

6.4.3 Adding Supervisory Information to the Model

The previously introduced frameworks do not consider the differences between different image regions when performing image translation, and we actually need to use an attention mechanism to allow the model to focus on the really important semantic regions of the image in order to obtain better results. This is common in face-related editing tasks, and common forms of supervision include keypoint heat maps, facial segmentation masks, and here we present Landmark-Assisted CycleGAN as a representative framework.

Landmark-Assisted CycleGAN [16] is a face cartoon image generation framework that uses facial keypoint information as supervision to constrain the stylized distribution of the facial organs to reasonably satisfy a priori knowledge, and Fig. 6.22 shows the generator schematic of the framework.

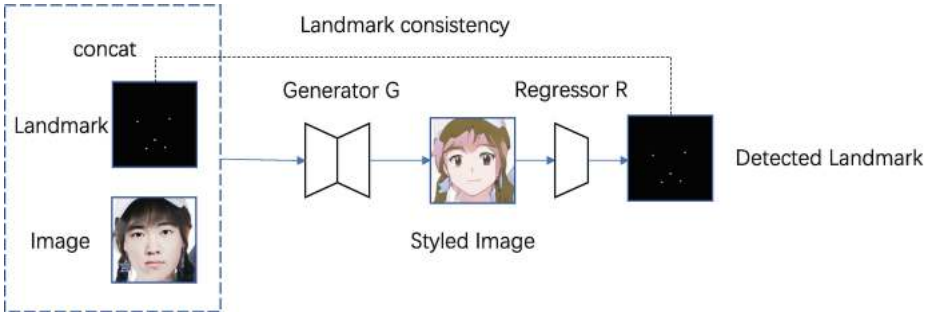


Fig. 6.22 Key point-based face supervision

The input of the generator includes RGB map and keypoint heat map, and the style map is obtained after the generator G. Then the style map is detected using the pre-trained keypoint regressor R to obtain the keypoint heat map, and the loss is calculated with the keypoint heat map of the input RGB map. This constrains the consistency of the distribution of the facial organs after stylization, without changing the identity information, and the key point loss is L2 distance.

The discriminators include global and local discriminators, where the global discriminators can be divided into conditional and unconditional discriminators according to whether keypoint information is input, conditional discriminators use the style map and keypoint heat map stitching as input, and unconditional discriminators use only the style map as input.

The difference between the local discriminator and the global discriminator is that it discriminates the semantic sub-regions of the face by multiple discriminators, including nose, eyes, and mouth in three parts, which are used to constrain the local authenticity of the generated images.

The Landmark-Assisted CycleGAN framework uses keypoint information to globally constrain face organs, and keypoints are also crucial for tasks such as face expression transformation [17].

There are also frameworks that use finer segmentation masks [18, 19] for semantic awareness, which is crucial for improving the quality of the generated images.

6.5 Pix2Pix Model-Based Image Coloring Practice

Previously, we have introduced the classic framework for image translation, of which Pix2Pix is the earliest important work. In this subsection, we practice image coloring practice based on Pix2Pix, and Pytorch is chosen as the framework. Moreover, we refer to the original author's open source code, which can be found at https://github.com/junyanz/pytorch- CycleGAN-and-pix2pix.

6.5.1 Data Processing

This time we have carried out three kinds of image coloring task for practice, namely portrait image, plant image, and architectural image, and we will introduce the related data processing work below.

6.5.1.1 Dataset

The portrait coloring task utilizes a high-definition face dataset, Celeba-HQ [20], which was released in 2019 and contains 30,000 high-definition face images including different attributes, where the image sizes are all 1024 × 1024.

The plant coloring task uses the publicly available dataset Oxford 102 Flowers Dataset, which was published in 2008 and contains 102 categories with a total of 102 species of flowers, with each category containing between 40 and 258 images.

The architectural coloring task dataset was obtained from crawlers crawling from search engines and photography websites, in which contains 8564 images.

6.5.1.2 Data Reading

For the image coloring task, it will be better in CIELab color space than in RGB color space because the L channel in CIELab color space has only grayscale information, while the A and B channels have only color information, achieving the separation of luminance and color. Therefore, in the data reading module, it is necessary to convert RGB images to CIELab color space and then construct pairs of data.

Let's look at the core function functions in the data reading class, including the initialization function __init__ and the data iterator __getitem__.

```
## The data class is defined as follows
class ColorizationDataset(BaseDataset).
def __init__(self, opt).
BaseDataset.__init__(self, opt)
self.dir = os.path.join(opt.dataroot, opt.phase)
self.AB_paths = sorted(make_dataset(self.dir, opt.max_dataset_size))
assert(opt.input_nc == 1 and opt.output_nc == 2 and opt.direction == 'AtoB')
self.transform = get_transform(self.opt, convert=False)

def __getitem__(self, index).
path = self.AB_paths[index]
im = Image.open(path).convert('RGB') ## Read the RGB image
im = self.transform(im) ## Preprocessing
im = np.array(im)
lab = color.rgb2lab(im).astype(np.float32) ## Convert RGB map to CIELab map
lab_t = transforms.ToTensor()(lab)
L = lab_t[[0], ...] / 50.0 - 1.0 ## Normalize the value of the L channel
(index=0) to between -1 and 1
AB = lab_t[[1, 2], ...] / 110.0 ## Normalize the values of channels A, B
(index=1,2) to between 0 and 1
return {'A': L, 'B': AB, 'A_paths': path, 'B_paths': path}
```

In the __getitem__ function shown above, the image is first read by using the PIL package and then preprocessed and converted into CIELab space. The value range of the L channel after being read is between 0 and 100, which is normalized to between -1 and 1 by processing. The values of A and B channels after being read range from 0 to 110, which are normalized to between 0 and 1 by processing.

In addition, the __init__ function is preprocessed by calling the get_transform function, which mainly contains operations such as image scaling, random cropping, random flipping, and subtracting the mean divided by the variance. Since they are relatively general operations, the key code is not interpreted here.

6.5.2 Interpreting the Model Code

Let's first interpret the key code of the model, including the preprocessing of data and the configuration of the model.

6.5.2.1 Generator Network

U-Net structure is adopted as the generator, the residual structure, and the UNet structure are provided in the open source project for choice, and we use U-Net to complete the experiments.

```
## The UNet generator is defined as follows
class UnetGenerator(nn.Module).
def __init__(self, input_nc, output_nc, num_downs, ngf=64,
norm_layer=nn.BatchNorm2d, use_dropout=False).
super(UnetGenerator, self). __init__()
unet_block = UnetSkipConnectionBlock(ngf*8,ngf*8, input_nc=None,
submodule=None, norm_layer=norm_layer, innermost=True) # add the innermost
layer
for i in range(num_downs - 5).
unet_block=UnetSkipConnectionBlock(ngf*8,ngf*8,input_nc=None,
submodule=unet_block, norm_layer=norm_layer, use_dropout=use_dropout)
## Gradually reduce the number of channels from ngf * 8 to ngf
unet_block=UnetSkipConnectionBlock(ngf*4,ngf*8,input_nc=None,
submodule=unet_block, norm_layer=norm_layer)
```

```
unet_block=UnetSkipConnectionBlock(ngf*2,ngf*4,input_nc=None,
submodule=unet_block, norm_layer=norm_layer)
unet_block=UnetSkipConnectionBlock(ngf,ngf*2,input_nc=None,
submodule=unet_block, norm_layer=norm_layer)
self.model=UnetSkipConnectionBlock(output_nc,ngf,input_nc=input_nc,
submodule=unet_block, outermost=True, norm_layer=norm_layer) ## outermost

def forward(self, input).
"""Standard forward"""
return self.model(input)
```

As shown in the above code, UnetGenerator mainly consists of the skip connection module UnetSkipConnectionBlock.

Among the important input parameters, input_nc is the number of input channels, output_nc is the number of output channels, num_downs is the number of downsampling, it controls how many skip connected modules are in the middle except for the innermost and outermost jump layer connected modules, ngf is the number of output channels of the last convolutional layer of the generator, and norm_layer is the normalization layer.

The UnetSkipConnectionBlock is defined as follows:

```
class UnetSkipConnectionBlock(nn.Module).
def __init__(self, outer_nc, inner_nc, input_nc=None.
submodule=None,outermost=False,innermost=False, norm_layer=nn.BatchNorm2d,
use_dropout=False).
super(UnetSkipConnectionBlock, self). __init__()
self.outermost = outermost
if type(norm_layer) == functools.partial.
use_bias = norm_layer.func == nn.InstanceNorm2d
else.
use_bias = norm_layer == nn.InstanceNorm2d
if input_nc is None.
input_nc = outer_nc
downconv = nn.Conv2d(input_nc, inner_nc, kernel_size=4.
stride=2, padding=1, bias=use_bias)
downrelu = nn.LeakyReLU(0.2, True)
downnorm = norm_layer(inner_nc)
uprelu = nn.ReLU(True)
upnorm = norm_layer(outer_nc)

if outermost: ## outermost
upconv = nn.ConvTranspose2d(inner_nc * 2, outer_nc.
kernel_size=4, stride=2.
padding=1)
down = [downconv]
up = [uprelu, upconv, nn.Tanh()]
model = down + [submodule] + up
elif innermost: ## innermost
upconv = nn.ConvTranspose2d(inner_nc, outer_nc.
kernel_size=4, stride=2.
padding=1, bias=use_bias)
down = [downrelu, downconv]
up = [uprelu, upconv, upnorm]
model = down + up
else: ## Intermediate layer
upconv = nn.ConvTranspose2d(inner_nc * 2, outer_nc.
kernel_size=4, stride=2.
```

```
padding=1, bias=use_bias)
down = [downrelu, downconv, downnorm]
up = [uprelu, upconv, upnorm]

## Whether to use dropout
if use_dropout.
model = down + [submodule] + up + [nn.Dropout(0.5)]
else.
model = down + [submodule] + up
self.model = nn.Sequential(*model)

def forward(self, x).
if self.outermost: ## Outermost output directly
return self.model(x)
else: ## Add a jump layer
return torch.cat([x, self.model(x)], 1)
```

outer_nc is the number of outer channels, inner_nc is the number of inner channels, input_nc is the number of input channels, submodule is the middle submodule, outermost is used to determine if it is the outermost layer, innermost is used to determine if it is the innermost layer, norm_layer is used to specify the normalized layer category, and user_ dropout is used to specify whether to use dropout or not.

For the pix2pix model, nn.BatchNorm2d is used as the default normalization layer, which is equivalent to InstanceNorm when batch = 1.

The visualization results of converting the trained generator to ONNX format are shown in Fig. 6.23.

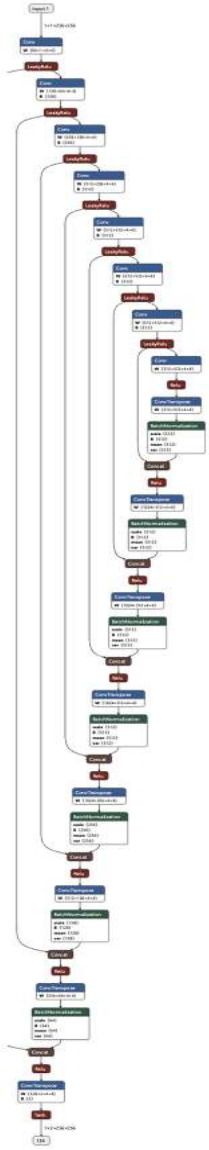


Fig. 6.23 Visualization result of generator structure

6.5.2.2 Discriminator Network

A discriminator is a classification model whose output is not a single predicted probability value, but a probability map of a certain size, and average pooling will perform on it, i.e., summing and averaging the probability map to obtain the final probability, as defined below.

```
## PatchGAN is defined as follows
class NLayerDiscriminator(nn.Module).
def __init__(self, input_nc, ndf=64, n_layers=3, norm_layer=nn.BatchNorm2d).
super(NLayerDiscriminator, self). __init__()
if type(norm_layer) == functools.partial.
use_bias = norm_layer.func == nn.InstanceNorm2d
else.
use_bias = norm_layer == nn.InstanceNorm2d

kw = 4 ## Convolution kernel size
padw = 1 ## Fill size
```

```
## First convolutional layer with a step size of 2
sequence = [nn.Conv2d(input_nc, ndf, kernel_size=kw, stride=2,
padding=padw), nn.LeakyReLU(0.2, True)]

## Multiple convolutional layers stacked, each with a step size of 2
nf_mult = 1
nf_mult_prev = 1
for n in range(1, n_layers): ## Gradually increase the channel width,
expanding it by twice as much each time
nf_mult_prev = nf_mult
nf_mult = min(2 ** n, 8)
sequence += [
nn.Conv2d(ndf * nf_mult_prev, ndf * nf_mult, kernel_size=kw, stride=2,
padding=padw, bias=use_bias).
norm_layer(ndf * nf_mult).
nn.LeakyReLU(0.2, True)
]

## Last convolutional layer with a step size of 1
nf_mult_prev = nf_mult
nf_mult = min(2 ** n_layers, 8)
sequence += [
nn.Conv2d(ndf * nf_mult_prev, ndf * nf_mult, kernel_size=kw, stride=1,
padding=padw, bias=use_bias).
norm_layer(ndf * nf_mult).
nn.LeakyReLU(0.2, True)
]

## Output single channel prediction result graph
sequence += [nn.Conv2d(ndf * nf_mult, 1, kernel_size=kw, stride=1,
padding=padw)]
self.model = nn.Sequential(*sequence)

def forward(self, input).
return self.model(input)
```

where input_nc is the input map channel, ndf is the number of output channels of the first convolutional layer, n_layers is the number of convolutional layers except the first and the last convolutional layer, the default PatchGAN corresponds to n_layers=3, the whole model consists of five convolutional layers, where the step of first four convolutional layers is 2, the step of the last convolutional layer is 1, and the global step size is 16. norm_layer is the type of the normalization layer , and the default type is BN layer.

The results of converting the trained discriminator to ONNX formatted visualization are shown in Fig. 6.24.

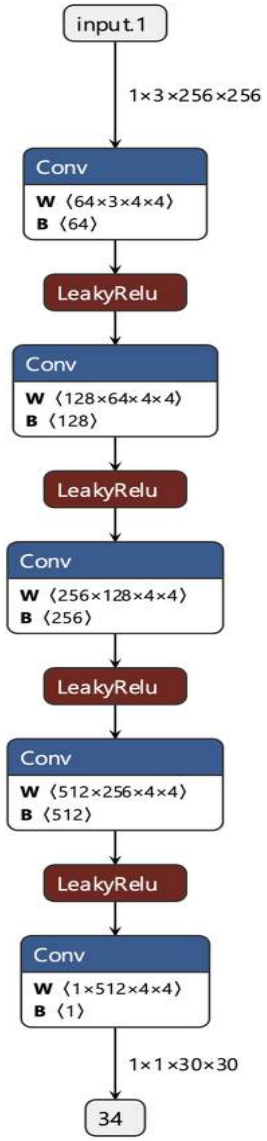


Fig. 6.24 Visualization results of discriminator structure

From Fig. 6.24, we can see that a discriminator is composed of five convolutional layers, and the size of final output probability map is 30 × 30.

6.5.2.3 Loss Function

Next, we turn to the definition of the loss function.

```
class GANLoss(nn.Module).
def __init__(self, gan_mode, target_real_label=1.0, target_fake_label=0.0).
## GAN loss type, support original loss, lsgan loss
super(GANLoss, self). __init__()
self.register_buffer('real_label', torch.tensor(target_real_label))
self.register_buffer('fake_label', torch.tensor(target_fake_label))
self.gan_mode = gan_mode
if gan_mode == 'lsgan'.
self.loss = nn.MSELoss()
elif gan_mode == 'vanilla'.
```

```
self.loss = nn.BCEWithLogitsLoss()
elif gan_mode in ['wgangp'].
self.loss = None
else.
raise NotImplementedError('gan mode %s not implemented' % gan_mode)

## Convert labels to the same size as the predicted result graph
def get_target_tensor(self, prediction, target_is_real).
if target_is_real.
target_tensor = self.real_label
else.
target_tensor = self.fake_label
return target_tensor.expand_as(prediction)

## Loss calls
def __call__(self, prediction, target_is_real).
if self.gan_mode in ['lsgan', 'vanilla'].
target_tensor = self.get_target_tensor(prediction, target_is_real)
loss = self.loss(prediction, target_tensor)
elif self.gan_mode == 'wgangp'.
if target_is_real.
loss = -prediction.mean()
else.
loss = prediction.mean()
return loss
```

The above code is mainly the definition of the adversarial loss of GAN, which can support the logarithmic loss of the original GAN, the loss of LSGAN.

6.5.2.4 Complete Model

After defining the discriminator and generator, we take a look at the definition of the complete PixPix model, which is shown as follows:

```
class Pix2PixModel(BaseModel).
## Configure default parameters
def modify_commandline_options(parser, is_train=True).
## default use batchnorm, network structure is unet_256, use paired
(aligned) image dataset
parser.set_defaults(norm='batch', netG='unet_256', dataset_mode='aligned')
if is_train.
parser.set_defaults(pool_size=0, gan_mode='vanilla')## Use classical GAN
loss
parser.add_argument('--lambda_L1', type=float, default=100.0, help='weight
for L1 loss') ## L1 loss weight is 100
return parser

def __init__(self, opt).
BaseModel.__init__(self, opt)
self.loss_names = ['G_GAN', 'G_L1', 'D_real', 'D_fake'] ## loss
self.visual_names = ['real_A', 'fake_B', 'real_B'] ## Intermediate result
graph
if self.isTrain.
self.model_names = ['G', 'D']
else: # during test time, only load G
self.model_names = ['G']

## Generator and discriminator definitions
```

```
self.netG = networks.define_G(opt.input_nc, opt.output_nc, opt.ngf,
opt.netG, opt.norm,not opt.no_dropout, opt.init_type, opt.init_gain.
self.gpu_ids)
## Discriminator definition, input RGB map and generator map stitching
if self.isTrain.
self.netD = networks.define_D(opt.input_nc + opt.output_nc, opt.ndf,
opt.netD, opt.n_layers_D, opt.norm, opt.init_type, opt.init_gain, self.
gpu_ids)

if self.isTrain.
## Loss function definition, GAN standard loss and L1 reconstruction loss
self.criterionGAN = networks.GANLoss(opt.gan_mode).to(self.device)
self.criterionL1 = torch.nn.L1Loss()
## Optimizer, using Adam
self.optimizer_G = torch.optim.Adam(self.netG.parameters(), lr=opt.lr,
betas=(opt.beta1, 0.999))
self.optimizer_D = torch.optim.Adam(self.netD.parameters(), lr=opt.lr,
betas=(opt.beta1, 0.999))
self.optimizers.append(self.optimizer_G)
self.optimizers.append(self.optimizer_D)

def set_input(self, input).
## Input preprocessing, set A, B according to different directions
AtoB = self.opt.direction == 'AtoB'
self.real_A = input['A' if AtoB else 'B'].to(self.device)
self.real_B = input['B' if AtoB else 'A'].to(self.device)
self.image_paths = input['A_paths' if AtoB else 'B_paths']

## Generator forward propagation
def forward(self).
self.fake_B = self.netG(self.real_A) #G(A)

## Discriminator loss
def backward_D(self).
## False sample loss
fake_AB = torch.cat((self.real_A, self.fake_B), 1)
pred_fake = self.netD(fake_AB.detach())
self.loss_D_fake = self.criterionGAN(pred_fake, False)
## True sample loss
real_AB = torch.cat((self.real_A, self.real_B), 1)
pred_real = self.netD(real_AB)
self.loss_D_real = self.criterionGAN(pred_real, True)
## True and false sample loss averaging
self.loss_D = (self.loss_D_fake + self.loss_D_real) * 0.5
self.loss_D.backward()

## Generator loss
def backward_G(self).
## GAN loss
fake_AB = torch.cat((self.real_A, self.fake_B), 1)
pred_fake = self.netD(fake_AB)
self.loss_G_GAN = self.criterionGAN(pred_fake, True)
## Reconstruction loss
self.loss_G_L1 = self.criterionL1(self.fake_B, self.real_B) *
self.opt.lambda_L1
## Loss-weighted average
self.loss_G = self.loss_G_GAN + self.loss_G_L1
```

```
self.loss_G.backward()

def optimize_parameters(self).
self.forward() # Calculate G(A)
## Update D
self.set_requires_grad(self.netD, True)
self.optimizer_D.zero_grad() ## D gradient clear
self.backward_D() ## Calculate the D gradient
self.optimizer_D.step() ## Update D weights
## Update G
self.set_requires_grad(self.netD, False) ## Optimize G without optimizing D
self.optimizer_G.zero_grad() ## G gradient clear
self.backward_G() ## Calculate G gradient
self.optimizer_G.step() ## Update G weights
```

The interpretation of the core code in the project is completed as shown above, and the next step is to conduct model training and testing.

6.5.3 Model Training and Testing

Next, we train and test the model.

6.5.3.1 Model Training

Model training is to complete the model definition, data loading, visualization, storage, etc. The core training code is as follows:

```
if __name__ == '__main__'.
opt = TrainOptions().parse() ## Get some training parameters
dataset = create_dataset(opt) ## Create dataset
dataset_size = len(dataset) ## dataset size
print('The number of training images = %d' % dataset_size)

model = create_model(opt) ## Create model
model.setup(opt) ## Model initialization
visualizer = Visualizer(opt) ## visualizer function
total_iters = 0 ## Number of iterations of batch

for epoch in range(opt.epoch_count, opt.niter + opt.niter_decay + 1).
epoch_iter = 0 ## current epoch iteration batch number
for i, data in enumerate(dataset): ## inner loop per epoch
visualizer.reset()
total_iters += opt.batch_size ## Total iterations of batch
epoch_iter += opt.batch_size
model.set_input(data) ## input data
model.optimize_parameters() ## iterative update

if total_iters % opt.display_freq == 0: ## visdom visualization
save_result = total_iters % opt.update_html_freq == 0
model.compute_visuals()
visualizer.display_current_results(model.get_current_visuals(), epoch,
save_result)

if total_iters % opt.print_freq == 0: ## Store information such as losses
losses = model.get_current_losses()
t_comp = (time.time() - iter_start_time) / opt.batch_size
visualizer.print_current_losses(epoch, epoch_iter, losses, t_comp, t_data)
if opt.display_id > 0.
```

```
visualizer.plot_current_losses(epoch, float(epoch_iter) / dataset_size,
losses)

if total_iters % opt.save_latest_freq == 0: ## Store the model
print('saving the latest model (epoch %d, total_iters %d)' % (epoch,
total_iters))
save_suffix = 'iter_%d' % total_iters if opt.save_by_iter else 'latest'
model.save_networks(save_suffix)

if epoch % opt.save_epoch_freq == 0: ## store model per opt.save_epoch_freq
epoch
model.save_networks('latest')
model.save_networks(epoch)

model.update_learning_rate()## Update the learning rate after each epoch
```

Some of the important training parameters are configured as follows:
input_nc=1, which means the generator input is 1-channel image, i.e., L channel.
output_nc=2, which means the generator output is 2-channel image, i.e., AB channel.
ngf=64, which means the output channel of the last one convolutional layer of the generator is 64.
ndf=64, indicating that the last one convolutional layer output channel of the discriminator is 64.
n_layers_D=3, indicating the use of the default PatchGAN, which is equivalent to discriminating blocks of 70 × 70 size images.
norm = batch, batch_size = 1, indicating use batch normalization.
load_size = 286, indicating the size of the loaded image.
crop_size = 256, indicating image crop size (i.e., training).

6.5.3.2 Model Testing

After judging that the model is already converged through the intermediate results and losses of training, we use the trained model for test, and the complete code is as follows:

```
import os
from PIL import Image
import torchvision.transforms as transforms
import cv2
import numpy as np
import torch
from models.networks import define_G

model_path = "checkpoints/portraits_pix2pix/latest_net_G.pth"
# parameters
input_nc = 3
output_nc = 3
ngf = 64
netG = 'unet_256'
norm = 'batch'

modelG = define_G(input_nc, output_nc, ngf, netG, norm,True)
params = torch.load(model_path,map_location='cpu') #load model
modelG.load_state_dict(params)

if __name__ == '__main__'.
imagedir = "myimages"
imagepaths = os.listdir(imagedir)
## Preprocessing functions
transform_list = []
transform_list.append(transforms.Resize((256,256),Image.BICUBIC))
```

```
transform_list += [transforms.ToTensor()]
transform_list += [transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]
Compose(transform_list)
for imagepath in imagepaths.
img = Image.open(os.path.join(imagedir,imagepath))
img = img.convert('RGB')
img = transform(img)
img = np.array(img)
lab = color.rgb2lab(img).astype(np.float32)
lab_t = transforms.ToTensor()(lab)
L = lab_t[[0], ...] / 50.0 - 1.0
L = L.unsqueeze(0)
AB = modelG(L)
AB2 = AB * 110.0
L2 = (L + 1.0) * 50.0
Lab = torch.cat([L2, AB2], dim=1)
Lab = Lab[0].data.cpu().float().numpy()
Lab = np.transpose(Lab.astype(np.float64), (1, 2, 0))
rgb = (color.lab2rgb(Lab) * 255).astype(np.uint8)
```

In the above code, since the Batchsize=1 is used for training, we do not use the stored batchsize parameter for testing, i.e., we do not turn on the modelG.eval() option, the reader can compare the difference after turn on the option.

After reading the RGB image, it is first converted to CIELab space, and then the L channel is input to the generator to obtain the coloring result.

The results of the portrait, plant, and building drawings are shown in Figs. 6.25, 6.26 and 6.27, respectively.

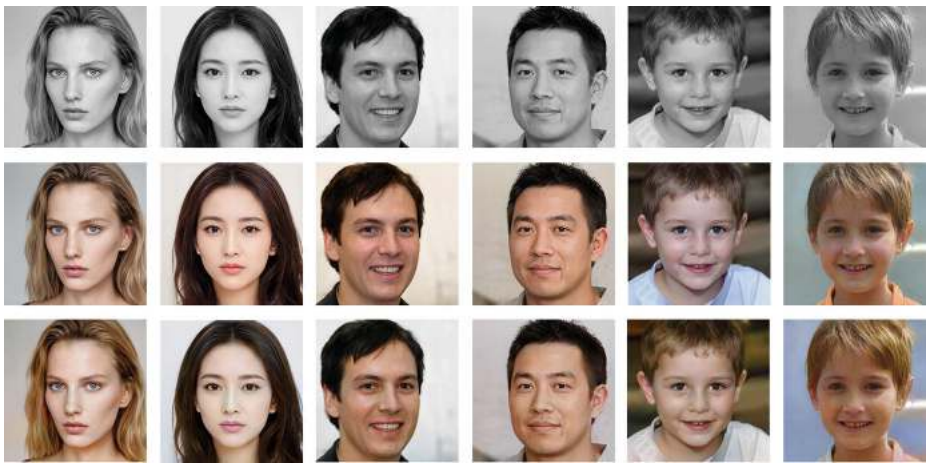


Fig. 6.25 Portrait image coloring results

Fig. 6.26 Plant image coloring results

Fig. 6.27 Architectural image coloring results

The first row of images in Figs. 6.25, 6.26 and 6.27 are the L channel images which are the input of generator, the second row is the real RGB images, and the third row is the result of image coloring. Among them, none of the input images are in the training set, the face images are generated by StyleGAN, part of the plant and building images are from the author's photography, and part of them are randomly taken from the validation set.

The results of the above three images show that although the coloring results are not exactly the same as those of the original RGB images, all three types of images obtain a more realistic coloring effect. The skin tones of human faces, the skies of architectural drawings, and the subjects of plants do not show colors that obviously violate a priori common sense, indicating that the model has indeed learned the semantic information of the target.

For all the test images, a smooth local and global coloring style was obtained, with no obvious discontinuous flaw, and a good result was obtained for some of the difficult test images. For instance, the 2nd image of the plant figure where the main body and background are both yellow. Although the coloring result is not consistent with the original image, the consistency of the coloring structure of the main body is very good, and the background has a richer color distribution. For the 6th image of the architectural figure, the input RGB image is also a monochrome style map, and the coloring result keeps the style well without discontinuous color defects, which verifies the robustness of the model.

6.5.4 Summary

In this chapter, we have completed several types of common image coloring tasks by using the supervised Pix2Pix image translation framework to verify the practical effectiveness of the model in image coloring tasks, and there are some experiments available for interested readers to extend their practice, including:

(1) Compare the results of training models in RGB color space, HSV color space, and CIELab color space to verify whether CIELab color space, HSV color space will give better results compared to RGB color space.

(2) Train a unified coloring model to colorize three types of images and compare it with a model trained specifically for a single task to see if the latter has better results.

References

1. Isola P, Zhu J Y, Zhou T, et al. Image-to-image translation with conditional adversarial networks[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2017: 1125-1134.
2. Wang T, Liu M, Zhu J, et al. High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs[C]. Computer vision and pattern recognition, 2018: 8798-8807.
3. Wang T, Liu M, Zhu J, et al. Video-to-video synthesis [C]. neural information processing systems, 2018: 1144-1156.1
4. Taigman Y, Polyak A, Wolf L. Unsupervised cross-domain image generation [J]. arXiv preprint arXiv:1611.02200, 2016.
5. Dong H, Neekhara P, Wu C, et al. Unsupervised image-to-image translation with generative adversarial networks [J]. arXiv preprint arXiv:1701.02676, 2017.
6. Liu M, Tuzel O. Coupled Generative Adversarial Networks[C]. neural information processing systems, 2016: 469-477.
7. Liu M, Breuel T M, Kautz J, et al. Unsupervised Image-to-Image Translation Networks[C]. neural information processing systems, 2017: 700-708.
8. Zhu J Y, Park T, Isola P, et al. Unpaired image-to-image translation using cycle-consistent adversarial networks[C]//Proceedings of the IEEE international conference on computer vision. 2017: 2223-2232.
9. Kim T, Cha M, Kim H, et al. Learning to Discover Cross-Domain Relations with Generative Adversarial Networks [J]. arXiv: Computer Vision and Pattern Recognition, 2017.
10. Yi Z, Zhang H, Tan P, et al. Dualgan: Unsupervised dual learning for image-to-image translation [J]. arXiv preprint, 2017.
11. Royer A, Bousmalis K, Gouws S, et al. Xgan: Unsupervised image-to-image translation for many-to-many mappings [J]. arXiv preprint arXiv:1711.05139, 2017.
12. Choi Y, Choi M, Kim M, et al. Stargan: Unified generative adversarial networks for multi-domain image-to-image translation [C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018: 8789-8797.
13. Choi Y, Uh Y, Yoo J, et al. Stargan v2: Diverse image synthesis for multiple domains[C]//Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2020: 8188-8197.
14. Huang X, Liu M, Belongie S, et al. Multimodal Unsupervised Image-to-Image Translation [C]. european conference on computer vision, 2018: 179-196.
15. Almahairi A, Rajeshwar S, Sordoni A, et al. Augmented cyclegan: Learning many-to-many mappings from unpaired data[C]//International conference on machine learning. PMLR, 2018: 195-204.
16. Wu R, Gu X, Tao X, et al. Landmark Assisted CycleGAN for Cartoon Face Generation.[J]. arXiv: Computer Vision and Pattern Recognition, 2019.
17. Song L, Lu Z, He R, et al. Geometry guided adversarial facial expression synthesis[C]//Proceedings of the 26th ACM international conference on Multimedia, 2018: 627-635.
18. Jiang S, Tao Z, Fu Y. Segmentation Guided Image-to-Image Translation with Adversarial Networks[J]. arXiv preprint arXiv:1901.01569, 2019.
19. Lee CH, Liu Z, Wu L, et al. MaskGAN: towards diverse and interactive facial image manipulation [J]. arXiv preprint arXiv:1907.11922, 2019.
20. Karras T, Aila T, Laine S, et al. Progressive growing of gans for improved quality, stability, and variation[J]. arXiv preprint arXiv:1710.10196, 2017

P. Long, X. Guo, *Generative Adversarial Network*
https://doi.org/10.1007/978-981-96-9404-4_7

7. Face Image Editing

Peng Long[1] ✉ and Xiaozhou Guo[2]
(1) Beijing YouSan Educational Technology, Beijing, China
(2) • China Electronics Technology Group Corporation No. 54 Research Institute, Shijiazhuang, China

Abstract
This chapter explores GAN-based facial image editing, addressing tasks like expression, age, pose, style, makeup, and identity swapping. Expression editing uses keypoint-guided models (e.g., G2-GAN with cycle and identity loss). Age editing leverages latent space models (e.g., CAAE with adversarial autoencoders). Pose editing integrates 3DMM models (e.g., FFGAN and FaceID-GAN using face recognition features). Style transfer employs attention mechanisms (e.g., UGATIT with adaptive normalization). Makeup transfer (e.g., BeautyGAN) combines cycle consistency and histogram matching. Face swapping relies on geometric deformation or deep feature disentanglement. A unified framework (StyleGAN) enables attribute manipulation via latent vector editing. Finally, A very detailed practical project based on StyleGAN for face editing was demonstrated to help readers grasp the details.

Keywords Face editing – 3DMM – CycleGAN – StyleGAN – Attribute preservation

Previously, we introduced the basic framework of GAN in image generation and image translation tasks. As an emerging technology, GAN has a wide range of applications in many face image tasks, and in this chapter we introduce the typical technical framework of GAN in face editing.

7.1 Face Expression Editing

Face expression editing can be widely used in entertainment and social fields, and normalization of expressions also helps to improve the performance of algorithms such as key point localization and face recognition of face images under large expressions. In this section, we introduce the face expression editing problem with a typical framework.

7.1.1 Expression Editing Issues

Face expression editing, that is, changing the expression properties of the face, as shown in Fig. 7.1, realizes the editing of neutral expression to smiling expression. The expression unit of the face mainly includes areas such as lips, nose, and eyes, so the expression editing mainly modifies these areas.

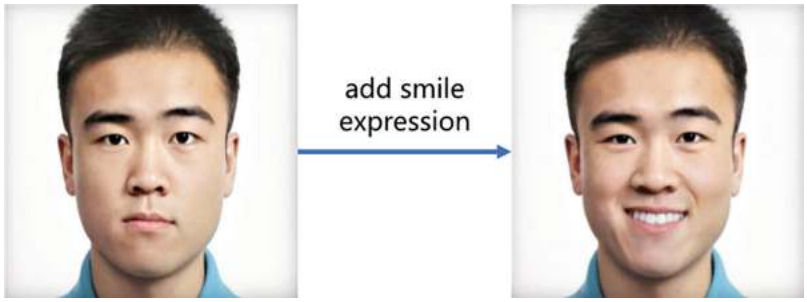


Fig. 7.1 Face smile expression editing

In the expression editing task, it is not only necessary to complete the attribute editing of expressions, i.e., to deal with the presence or absence of a certain expression. It is also necessary to complete the

magnitude editing to achieve continuous and smooth expression transformation, which is a much more difficult problem.

7.1.2 Key Point Controlled Expression Editing Model

Since the expression of human face, which is mainly determined by the geometric deformation of facial units such as lips, nose, and eyes, is related to the face keypoint task, this section introduces a keypoint-controlled face expression editing model G2-GAN [1], which uses the heat map of face keypoints as a condition to control the generation and removal of expressions, with a generator structure shown in Fig. 7.2, and a discriminator shsown in Fig. 7.3.

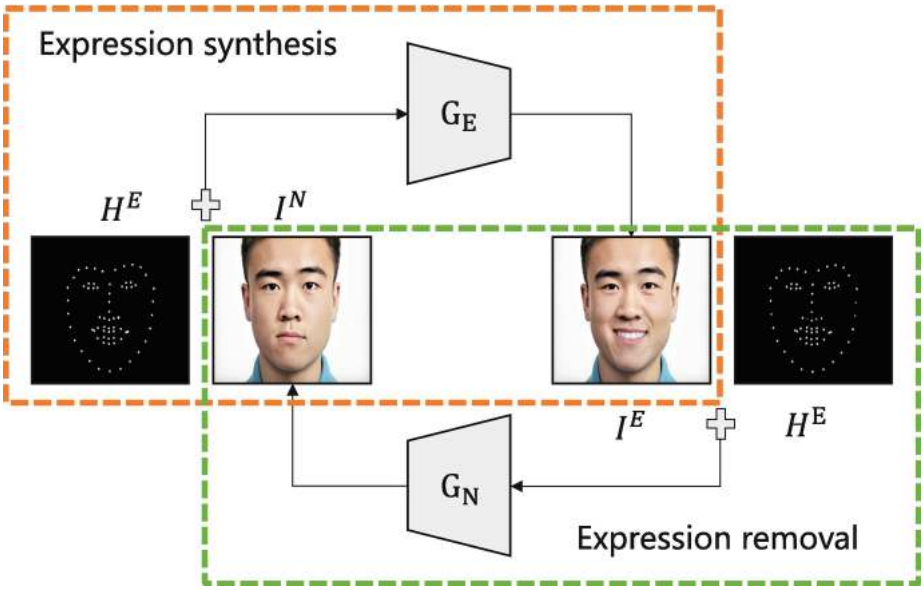


Fig. 7.2 G2-GAN generator structure

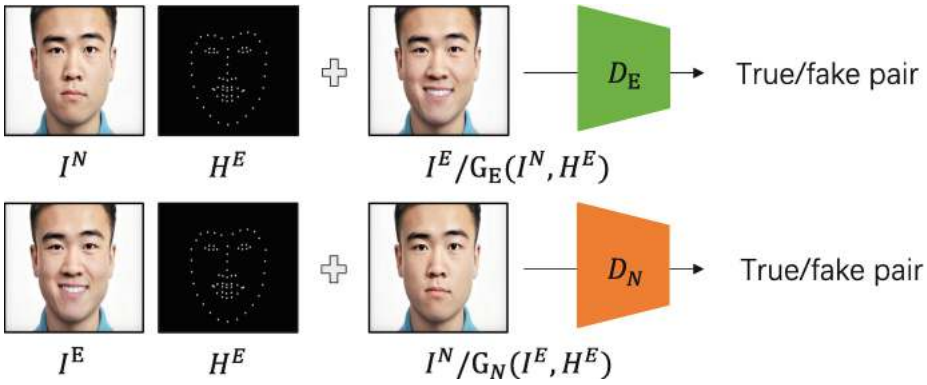


Fig. 7.3 G2-GAN discriminator structure

The model consists of two generators and two discriminators. Generator G_E is an expression generator that inputs a keypointed heat map H^E with expression and an expressionless positive face I to generate an expressive image I^E. Generator G_N, on the contrary, inputs a keypointed heat map H^E with expression and an expressive positive face I^E to generate an expressionless image I^N. In the expression generation task, H^E plays the role of controlling the magnitude, while in the expression removal task, H^E plays a similar role as the annotation role.

The discriminator D_E is used to discriminate between real triples (I^N,H^E,I^E) and generated triples (I^N,H^E,G_E (I^N,H^E)), and D_N is similar.

The overall loss function consists of four components.

The first part is the standard GAN loss and it will not be repeated here.

The second component is the pixel loss, which is used to constrain the pixel smoothing of the generated and input images, which is defined as follows:

$$L_{\text{pixel}} = E_{I,H,I' \sim P(I,H,I')} \| I' - G(I,H) \|_1 \tag{7.1}$$

The third part is the same cycle loss as CycleGAN, which forms a closed loop from the input image after passing through two generators, which is shown in Eq. (7.2), where *G* and *G*' are generators in opposite directions to each other.

$$L_{\text{cyc}} = E_{I,H \sim P(I,H)} \| I - G'(G(I,H)) \|_1 \tag{7.2}$$

The fourth part is the attribute retention loss, which is used to constrain the identity information from being changed, which is shown in Eq. (7.3), where F denotes the feature extractor for face recognition.

$$L_{\text{idendity}} = E_{I,H \sim P(I,H)} \| F(I) - F(G(I,H)) \|_1 \tag{7.3}$$

7.2 Face Age Editing

The variation of face age poses a challenge for algorithms such as face recognition. Age editing can be used not only in the entertainment and social fields, but normalization of age can help improve the performance of algorithms such as face recognition. In this section, we introduce the face age editing problem and a typical framework.

7.2.1 Age Editing Issues

The so-called age editing refers to changing the age attribute of a photo. Age transformation is widely used in movies, for example, young actors become old in the movie or old actors need to play young people. It is also used in the field of public security, such as finding children who have been lost for many years; in addition, there are many applications in life and entertainment as well as in customer analysis statistics.

Age editing contains two subproblems, as shown in Fig. 7.4. One is getting older (Age Progression) and the other is getting younger (Age Regression). The Aging problem is easier than the Younger problem, and most current models can only change to a certain age group without being able to simulate to a specific age value.

Fig. 7.4 Face age editing

7.2.2 Conditional Adversarial AutoEncoder Based on Latent Space

This section introduces a representative framework for face age editing, namely the Conditional Adversarial AutoEncoder (CAAE) model based on latent space [2].

The CAAE model first assumes that the face image is in a high-dimensional manifold (HDF), and that the age changes naturally as the image moves in a particular direction in this manifold, but manipulating the face image in a high-dimensional manifold is a very difficult task, and we cannot directly depict the trajectory. Therefore, as with most generative models, it is necessary to first map the image into a low-dimensional latent space to obtain a low-dimensional vector, and finally map the processed low-dimensional vector back into the high-dimensional manifold. These two mappings are implemented by Encoder E and Generator G, respectively, and the model structure is shown in Fig. 7.5.

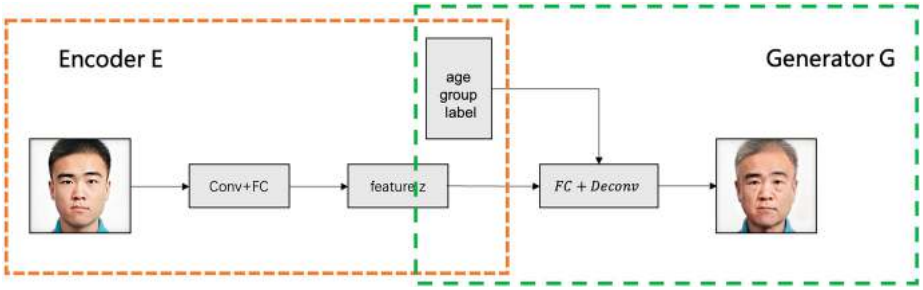


Fig. 7.5 Encoder E and generator G of CAAE model

The encoder E inputs the image and outputs the feature vector z, which is later stitched with the *n*-dimensional age label vector as the input to the generator G. Then G outputs the simulated face.

There are two discriminators, D_{img} and D_z. The age label vector is filled and input to the discriminator D_{img} with the face map for channel stitching, which discriminates the authenticity of the generated map, in fact, the classification of age groups, and it contains several convolutional layers and several fully connected layers. D_z is used to constrain z to be a uniform distribution, and it contains several fully connected layers.

7.3 Face Pose Editing

Large pose poses a challenge for algorithms such as face keypoint detection and face recognition. Pose editing algorithms can emulate faces in different poses, such as correcting faces in large poses to front faces, which helps to improve the performance of algorithms such as face recognition. In this section, we introduce the face pose editing problem with a typical framework.

7.3.1 Pose Editing Issues

Face pose editing means changing the face pose, as shown in Fig. 7.6, and switching the pose arbitrarily after the 3D reconstruction of the face.

Fig. 7.6 Face 3D reconstruction and pose editing

Face pose editing can be used to simulate different poses, which is useful for improving the key point localization of faces in large poses and the accuracy of recognition models.

7.3.2 Pose Editing Model Based on 3DMM Model

Since face pose reconstruction is usually done using face 3D reconstruction, the current GAN-based face pose editing model also often requires face 3D reconstruction tasks together for joint learning, so in this subsection we introduce the 3DMM model-based pose editing GAN framework.

Face Frontalization (FFGAN) [3] is an early pose editing GAN that uses a generator to generate front faces by inputting coefficients of face and 3DMM models, and a discriminator to determine true and false, and a face recognition model to supervise the maintenance of face identity attributes.

In the FFGAN model, the 3DMM coefficients provide global pose information as well as low-frequency details, while the input large pose image provides high-frequency details. The whole loss function consists of five components, in addition to the face reconstruction loss, the full-variance smoothing loss, the adversarial loss of GAN, and the face recognition identity preservation loss, a face symmetry constraint is added as a symmetry loss.

FaceID-GAN [4] has similar idea with the FFGAN, which also controls the generated pose by a 3DMM model, and constrains the identity information retention by a face recognition model C. The difference with FFGAN is that C does not just become supervised information as an independent classifier, but fights with the discriminator D together with the generator G and directly participates in the image generation process. This time the classifier C will not only distinguish different identity, but also distinguish the real pose features from the generated ones, and its structure is shown in Fig. 7.7.

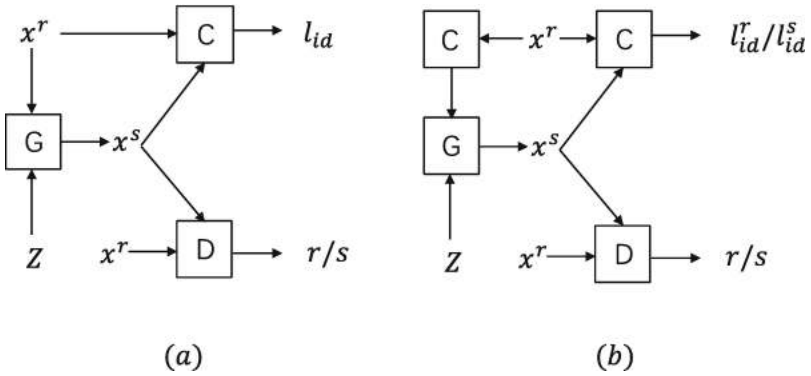


Fig. 7.7 Comparison of FaceID-GAN and general GAN model

The general GAN model with added identity constraints is represented in Figure 7.7a, where the generator inputs real images x^r and noise z, the output generates image x^r, and the classifier C outputs l_{id}. It can be seen that during the process from the real image to the generated image, the feature extraction uses the generator G, and then uses C to extract the features during classification process, which are not in the same feature space and will bring the problem of training difficulties.

In contrast, the input of the generator in FaceID-GAN in Fig. 7.7b is the real image x^r. The features and noise are extracted by C. Both the real image and the generated image are classified by the classifier C, thus using features from the same feature space, which will help reduce the training difficulty of the model.

The specific network structure of FaceID-GAN is shown in Fig. 7.8.

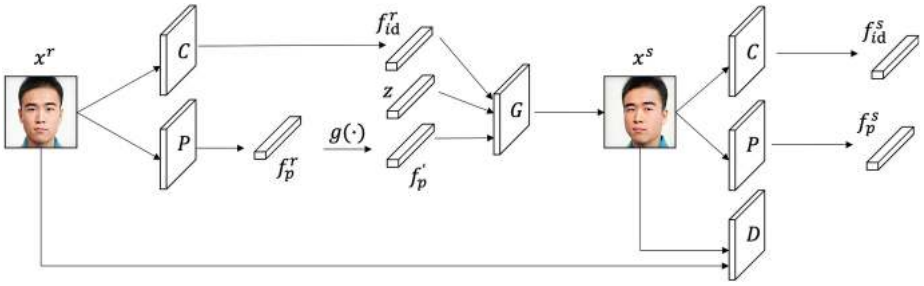


Fig. 7.8 FaceID-GAN structure

The input real image x^r, on the one hand, extracts the pose features by the 3DMM model P f^r_p, and it is transformed into the desired output pose features by the function g f'_p. On the other hand, the classification features are extracted by the classifier C f^r_{id}, which is combined with random noise z together to form the input of the generator G to generate the image x^s.

Then the generated images are again passed through classifier C, pose model P, and discriminator D, respectively, to extract the identity and pose features of the generated images and to discriminate whether the images are the same id and whether they are real or not.

The loss function of the discriminator is to minimize the reconstruction error of the real image and maximize the reconstruction error of the generated image, defined as follows. k_t is an empirical parameter.

$$\mathrm{R}(x) = \|x - \mathrm{D}(x)\|_1 \tag{7.4}$$

(7.5)

$$\min_{\theta_{\mathrm{D}}} L_{\mathrm{D}} = \mathrm{R}(x^r - k_t \mathrm{R}(x^s))$$

The loss function of classifier C consists of two components, the loss of the real image and corresponding labels and the loss of the generated image and corresponding labels, both of which are cross-entropy losses, and a loss weight is added to equalize the contribution of the generated image in this loss function.

$$\min_{\theta_{\mathrm{c}}} L_{\mathrm{c}} = \varphi(x^r, l^r_{\mathrm{id}}) + \lambda \varphi(x^s, l^s_{\mathrm{id}}) \Big) \tag{7.6}$$

Finally, the loss function of the generator G is shown in Eq. (7.7).

$$\min_{\theta_{\mathrm{G}}} L_{\mathrm{G}} = \lambda_1 \mathrm{R}(x^s) + \lambda_2 d^{\cos}(f^r_{\mathrm{id}}, f^s_{\mathrm{id}}) + \lambda_3 d^{l2}(f'_p, f^s_p) \tag{7.7}$$

The first term R is the reconstruction error of the generated image, the second term is the cosine distance of the identity features of the input image and the generated image, and the third term is the Euclidean distance of the pose features of the input image and the generated image.

Due to the presence of the pose model P, this method can generate arbitrary views. In addition, since the generator inputs the features of classifier C instead of the whole image, the features extracted by classifier C filter out a lot of irrelevant background information, thus focusing more on the features of the human face, making the generated images more realistic.

It is worth noting that we can edit not only the pose but also the expression based on the 3DMM model.

7.4 Face Style Editing

Face styling is different from expression, age, pose editing, etc. It will completely change the texture and color style of the face itself and is mostly used in the entertainment and social fields. In this section, we introduce the face styling editing problem and a typical framework.

7.4.1 Style Editing Issues

Common face stylization includes several types, as shown in Fig. 7.9.

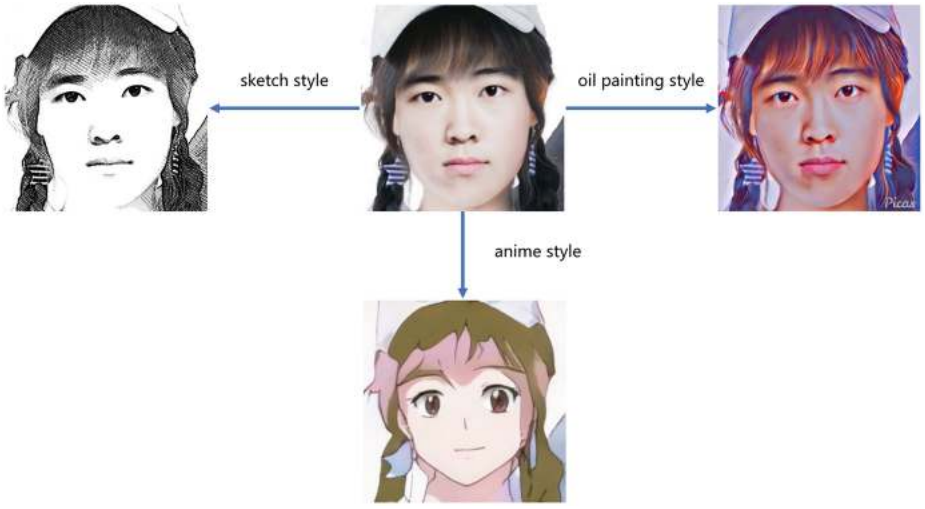


Fig. 7.9 Common face stylization

Face sketch: This belongs to a simpler class of images in stylization, which mainly generates black-and-white or colorful face edge contours. Traditional face sketch generation algorithms usually include filter-based and image block matching-based. The filter-based approach usually completes the positioning and generation of contours by edge detection and grayscale transformation, and the details are not smooth and complete enough. The image block-based method matches various parts of the face with the existing cartoon

parts in the dataset and then combines them, which is a more complicated step and difficult to perform accurate alignment.

Animated avatars: Unlike sketches, animated avatars have richer colors and textures, and the most common application is to generate avatars with similar styles to those in anime works, but retaining their own identity attributes, which are widely used in personalized social areas such as.

Artistic oil painting headshots: Artistic oil painting headshots usually have a more intense color style and brushstrokes, lacking image details, focusing on retaining the overall style, which can be achieved using the style migration network introduced earlier.

7.4.2 Stylized Model Based on Attention Mechanism

In this subsection, we introduce a representative framework for face stylized editing, namely a framework based on attention mechanisms and adaptive normalized style layers (Unsupervised Generative Attentional Networks with Adaptive Layer-Instance Normalization, or UGATIT [5]).

The first key technology of UGATIT is the attention mechanism, and the generator and discriminator structures are shown in Figs. 7.10 and 7.11.

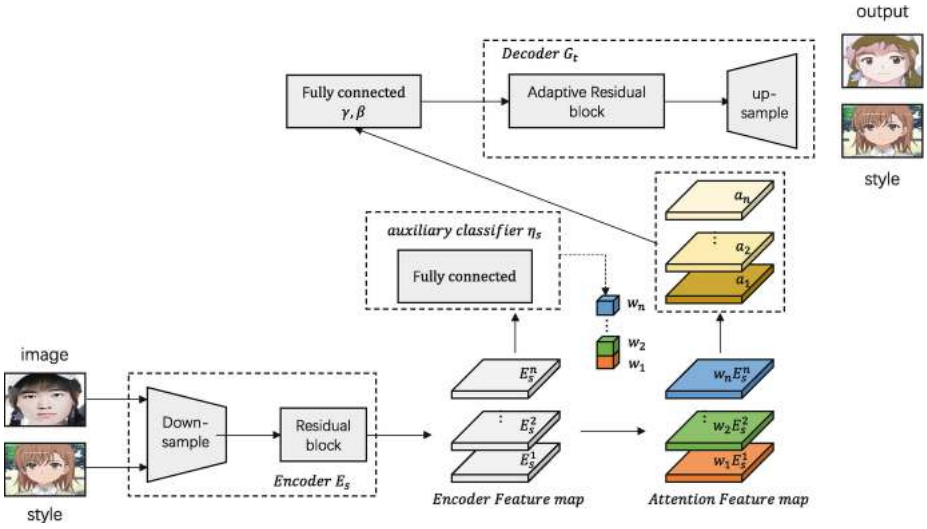


Fig. 7.10 Structure of UGATIT generator

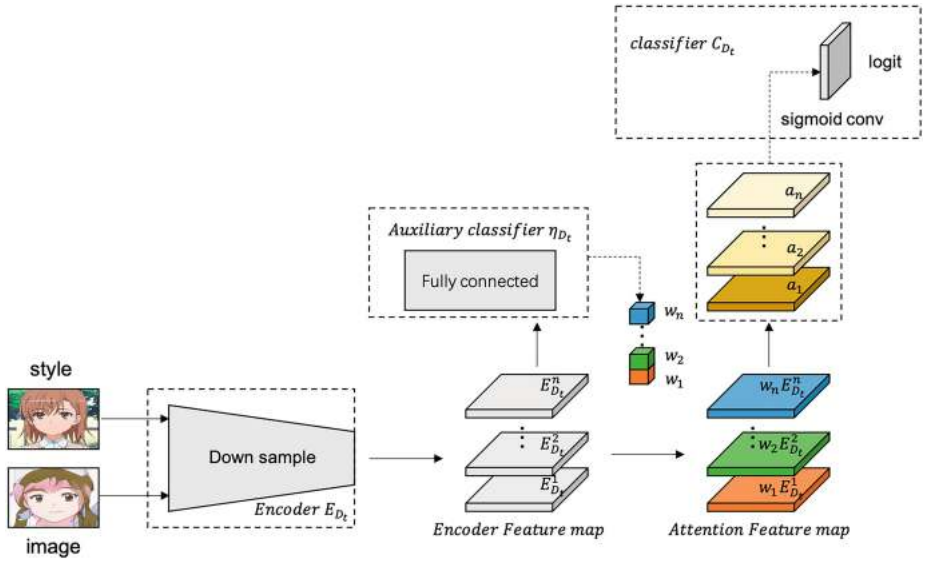


Fig. 7.11 Structure of UGATIT discriminator

Attention here is actually the use of the Class Activation Map (CAM feature map) under global and average pooling, where the input is the downsampled feature map and the output the weights of each channel.

In the generator, by the classifier $\eta s(x)$ learning weights w_s^k, as shown in Eq. (7.8). Where s denotes the source domain, k denotes the kth feature channel, E denotes the feature map, and i, j denotes the x, y coordinates.

$$\eta s(x) = \sigma\left(\sum_k w_s^k \sum_{ij} E_s^{k_{ij}}(x)\right) \tag{7.8}$$

The attention mechanism of the discriminator is based on the same principle, and the goal of this attention mechanism is to learn those important regions that can distinguish the difference between the source and target domains.

Additionally, UGATIT makes extensive use of the AdaLIN technique. In the field of image stylization, Instance Normalization (IN for short) and Layer Normalization (LN for short) are more commonly used techniques than Batch Normalization (BN for short).

IN retains more content structure because it normalizes each image feature map separately. Compared to IN, LN uses multiple channels for normalization, which enables better access to global features. adaLIN, on the other hand, combines the features of both with the following expression, where the parameters are learned through the attention module.

$$\widehat{a_I} = \frac{a - \mu_I}{\sqrt{\sigma_I^2 + \epsilon}} \tag{7.9}$$

$$\widehat{a_L} = \frac{a - \mu_L}{\sqrt{\sigma_L^2 + \epsilon}} \tag{7.10}$$

$$\mathrm{AdaLIN}(\alpha, \beta, \gamma) = \gamma \bullet \left(\rho \bullet \widehat{a_I} + (1 - \rho) \bullet \widehat{a_L}\right) + \beta \tag{7.11}$$

$$\rho \leftarrow \mathrm{cli}\,\mathrm{p}_{[0,1]}(\rho - \tau \Delta \rho) \tag{7.12}$$

where μ_I, μ_L are the mean values of each channel and each layer, respectively. σ_I^2 and σ_L^2 are the variances of each channel and each layer respectively, and γ and β are two learned parameters of fully connected layer, and τ is a learning rate that ρ ranges from 0 to 1, when IN is more effective, ρ is close to 1, when LN is more effective, ρ is close to 0. If γ and β take fixed values, then the module is LIN.

The discriminator contains a global discriminator and a local discriminator, the difference is that the global discriminator is deeper, reaching a stride size of 32, and the perceptual field of the global discriminator has exceeded 256 × 256. In addition, the attention module is added to the discriminator, which can enhance the discriminator's ability to determine the true and false images in the target domain.

The loss function consists of four components, including the loss of GAN, the loss of circular consistency, the loss of identity, and the loss of CAM. We will not go over the first two losses. The identity loss refers to the need to ensure that no conversion is desired between the same domains, i.e., the output should not change when using the conversion model from B to A for pictures in domain A, as defined below:

$$L_{\text{identity}}^{s \to t} = \mathrm{E}_{x \sim X_t}[|x - \mathrm{G}_{s \to t}(x)|_1] \tag{7.13}$$

CAM loss is used to learn which features can be improved for the current state to be used to distinguish between the two domains, so the generator and discriminator use their respective learned auxiliary classifiers to discriminate, and the loss is defined in Eq. (7.14) and Eq. (7.15), respectively.

$$L_{\text{cam}}^{s \to t} = -\mathrm{E}_{x \sim X_s}[\log\,(\eta s(x))] + \mathrm{E}_{x \sim X_t}[1 - \log\,(\eta s(x))] \tag{7.14}$$

$$L_{\text{cam}}^{D_t} = \mathrm{E}_{x \sim X_t}\left[\left(\eta_{D_t}(x)\right)^2\right] + \mathrm{E}_{x \sim X_s}\left[1 - \left(\eta_{D_t}(G_{s \to t}(x))\right)^2\right] \tag{7.15}$$

7.5 Face Makeup Editing

Face beauty algorithms have been developed for many years before deep learning techniques became popular, with filtering and deformation algorithms being the main ones, which can be applied to peeling, whitening, and reshaping of features. The more fashionable makeup migration algorithm, which can migrate the makeup of a portrait to any portrait photo, is a more complex technique in face beauty algorithms and can be implemented using GAN-based models. In this section, we introduce the face makeup creation editing problem with a typical framework.

7.5.1 Makeup Editing Issues

The so-called makeup creation editing, that is, given a reference map and a makeup-free map, the makeup of the reference map will be migrated to the makeup-free map, so as to achieve the effect of a thousand makeup migration, as shown in Fig. 7.12.

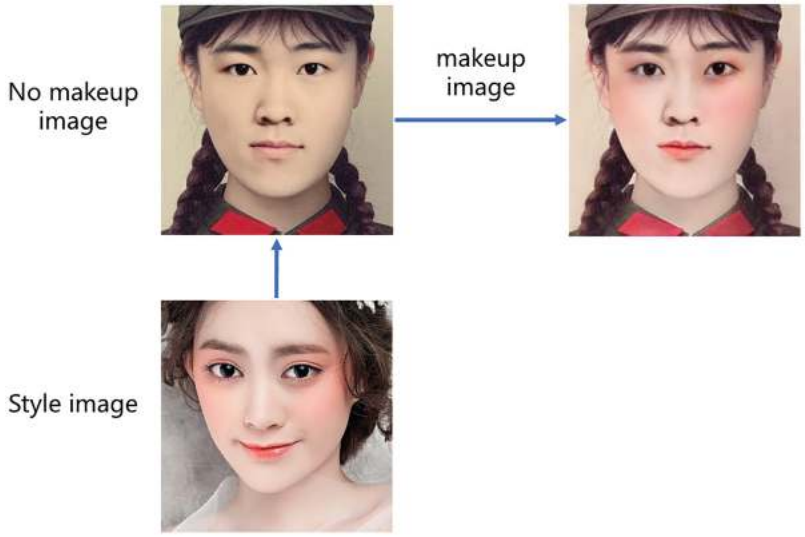


Fig. 7.12 Face makeup migration

7.5.2 GAN-Based Makeup Migration Algorithm

In this subsection, we introduce a representative framework for face makeup editing, namely Beauty GAN [6]. It inputs two face images, one without makeup and one with makeup, and the model outputs the result after the makeup change, i.e., one with makeup on and one with makeup off.

Beauty GAN adopts the classical image translation structure, and the generator G consists of two inputs, namely the no-makeup image I_{src}, the makeup image I_{ref}, and two outputs, namely the makeup-applied image I_{src}^{B}, the makeup-removed image I_{ref}^{A}, generated by the generator G composed of encoder, several residual blocks, and decoder. The structure schematic is shown in Fig. 7.13.

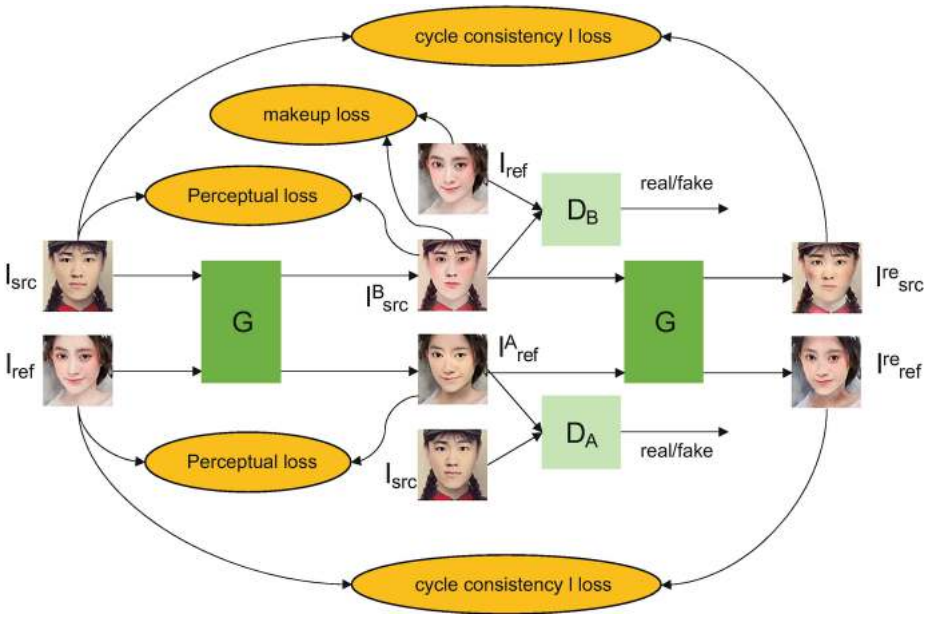


Fig. 7.13 BeautyGAN algorithm

BeautyGAN uses two discriminators D_A and D_B, where D_A is used to distinguish between true and false no-makeup maps and D_B is used to distinguish between true and false makeup maps.

In addition to the basic GAN loss, BeautyGAN contains three important losses, which are cycle consistency loss, perceptual loss, and makeup loss, the first two are global loss and the last one is local loss.

In order to eliminate the defects of migration details, the makeup-applied image I_{src}^{B} and the makeup-removed image I_{ref}^{A} are input to G again, and the makeup-removal and makeup-application are performed once again to obtain two reconstructed images $I_{\text{src}}^{\text{re}}$ and the makeup-removal image $I_{\text{ref}}^{\text{re}}$, at which time a image is constrained by the cycle loss to be the same as the corresponding original image after two G transformations. Because the input of the generator contains a pair of images, the difference with CycleGAN is that the same generator G is used here, and this loss is used to maintain the background information of the image, and the specific loss definition is the same as CycleGAN, which will not be repeated.

The application and removal of makeup cannot change the original character identity information, which can be constrained by perceptual loss based on the VGG model, defined as follows:

$$L_{\text{per}} = \frac{1}{C_l \times H_l \times W_l} \sum_{ijk} E_l \tag{7.16}$$

which C_l, H_l, W_l are the number of channels in the first l number of channels in the layer, the feature map height and width, and E_l is the Euclidean distance of the features, containing two components as follows:

$$E_l = \left[F_l(I_{\text{src}}) - F_l\left(I_{\text{src}}^{B}\right)\right]_{ijk}^{2} + \left[F_l(I_{\text{ref}}) - F_l\left(I_{\text{ref}}^{A}\right)\right]_{ijk}^{2} \tag{7.17}$$

In order to more accurately control the makeup effect in local regions, BeautyGAN trains a semantic segmentation network to extract the mask of different regions of the face, so that the makeup loss needs to be satisfied in the three regions of the face, eyes, and mouth for both the no-makeup and makeup maps, and the makeup loss is achieved by histogram matching, where the loss in one region is defined as follows:

$$L_{\text{item}} = I_{\text{src}}^{B} - HM\left(I_{\text{src}}^{B} \circ M_{\text{item}}^{1}, I_{\text{ref}} \circ M_{\text{item}}^{2}\right)_2^2 \tag{7.18}$$

M_{item}^{1} and M_{item}^{2} denote, respectively, the two corresponding area masks of I_{src}^{B} and I_{ref} and $\circ$ denotes the pixel-by-pixel multiplication, the item can represent the three regions of face, eye, and mouth, respectively,

and HM is a histogram matching operation.

The overall makeup loss is defined as follows:

$$L_{\text{makeup}} = \lambda_l L_{\text{lips}} + \lambda_s L_{\text{shadow}} + \lambda_f L_{\text{face}} \tag{7.19}$$

where L_{lips}, L_{shadow}, and L_{face} denote the lips, the eyes and the face, respectively. λ_l, and λ_s, and λ_f are the corresponding weights.

The complete BeautyGAN generator loss is as follows:

$$L = \alpha L_{\text{adv}} + \beta L_{\text{cyc}} + \gamma L_{\text{per}} + L_{\text{makeup}} \tag{7.20}$$

7.6 Face Swapping Algorithm

Face swapping algorithms, i.e., editing the identity of a face to make it become another person, have wider applications in film and TV drama creation, entertainment, and social fields, and it also poses a challenge to current face recognition models. In this section, we introduce the face swapping problem and a typical framework.

7.6.1 Identity Editing Issues

Early face swapping algorithms included two main categories, one is based on the geometric deformation of 2D face shapes, i.e., based on the detected key points, then calculate the deformation matrix between two face shapes for transformation, and then add post-processing techniques such as image fusion, etc. The current face swapping algorithms in applications such as Daily P are based on this. Figure 7.14 shows a typical face swapping case.

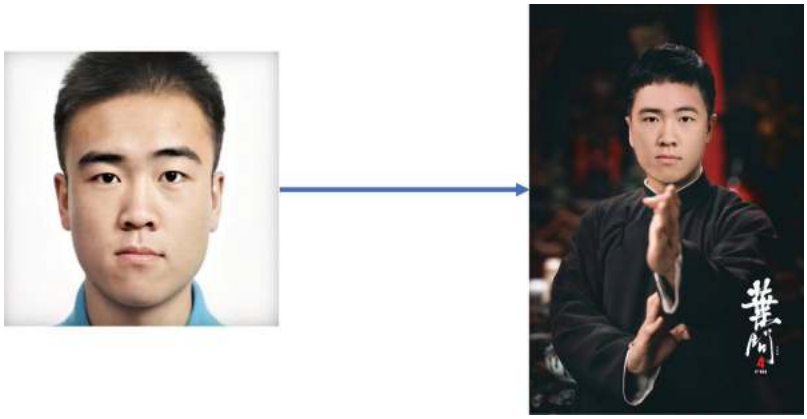


Fig. 7.14 Schematic diagram of face swapping algorithm

The other category is the 3D-based approach, which includes multi-stage processing techniques such as face reconstruction, tracking, and alignment, each of which requires complex operations, multiple images or videos, and cannot be applied in real time.

7.6.2 Deepfakes Face Swapping Algorithm

Most of the current deep learning-based face swapping algorithms are based on GAN, and the current series of mainstream face swapping algorithms are popularly originated from Deepfakes [7], so we take Deepfakes as an example to introduce the core technology of face swapping algorithm.

Deepfakes is an open source project and also a generic term for a class of algorithms whose training process and testing process are shown in Figs. 7.15 and 7.16, respectively.

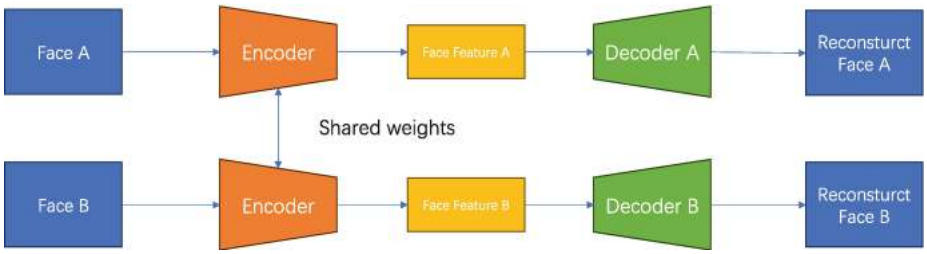


Fig. 7.15 Deepfakes training flow

Fig. 7.16 Deepfakes test flow

The training of Deepfakes requires two sets of images in two domains, called A and B. Deepfakes trains a decoder on set A and set B, respectively, under the constraint that the same encoder is used.

In the testing process, the face in image A can be replaced by the face in collection B by selecting the image from collection A, extracting the features by the same encoder, and then inputting the decoder that has been trained on collection B. Those who are interested can refer to the open source code to try.

In addition, face swapping algorithm can also be regarded as a face-to-face image translation problem, so Pix2Pix, CycleGAN, and other models can be directly applied to obtain very realistic face swapping results under the supervision of adding face masks, pose, lighting, and other information.

7.7 Generic Face Attribute Editing

StyleGAN is an image generation framework of great performance that can also be used for face attribute editing. In this section, we introduce the key techniques of face attribute editing based on StyleGAN.

7.7.1 Key Issues in StyleGAN Face Editing

For more information about the principle of StyleGAN, readers can go back to Chap. 5 to read about it. Now we want to use StyleGAN for real face editing, we need to solve two key problems (Fig. 7.17).

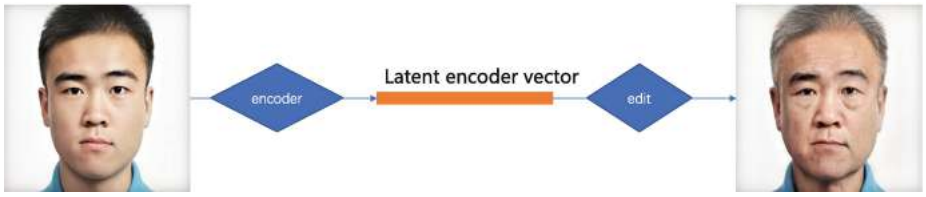


Fig. 7.17 Flow and key issues of face attribute editing based on StyleGAN

1. How to obtain the latent encoding vector of a real face, which corresponds to the input Z or output W of the mapping network in StyleGAN.
2. How to control the high-level semantic properties of the generated face images by modifying the Z or W vectors.

We next focus on the solution of the latent encoding vector and experiment with attribute editing based on the latent encoding vector in the next section.

7.7.2 Solving for Latent Coding Vectors

The current solution to the real face encoding vector is basically based on two ideas: one is to learn an encoder to implement the mapping, and the other is to solve the vector directly by optimization.

7.7.2.1 Encoder-Based Solution

The encoder-based solution framework is shown in Fig. 7.18 and consists of two modules.

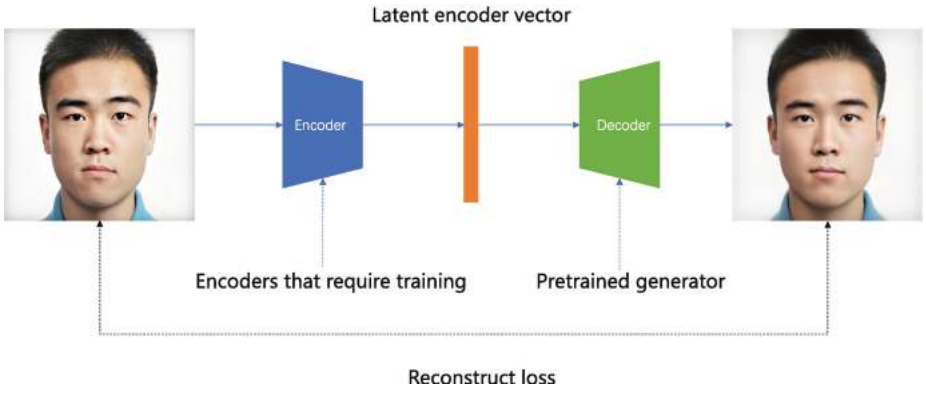


Fig. 7.18 Face encoder training framework

Encoder represents the encoder to be trained, and Decoder represents the generator model that has been trained, such as the generator part of StyleGAN. The real image is input to the encoder to get Z or W, and then input to the generator to get the generated face to complete the reconstruction of the face image.

By learning an encoder directly, it is possible to extract latent coding vectors for any image in one training without optimizing each image, but it is also prone to overfitting on the training dataset.

7.7.2.2 Optimization-Based Solution Method

Another approach is based on optimization solving, directly for each image, optimize the corresponding W. This scheme is used in frameworks such as StyleGAN v2, Image2StyleGAN [8], and the 512-dimensional W is expanded into W^+, W^+ is a 18 × 512-dimensional matrices, so that each adaptive instance normalization (AdaIN) style module can be used for each different W, achieving freer attribute editing.

The method based on the optimal solution consists of the following steps:

1. Given an image I, and a pre-trained generator G.
2. Initialize a latent encoding vector, such as W, whose initial value can use the calculated statistical average.
3. Iterative iterations are performed according to the optimization objective until a preset termination condition is reached.

Common forms of optimization objectives are:

$$w^* = \underset{w}{\operatorname{argmin}} \mathscr{L}_{\text{percept}}(G(w), I) + \frac{\lambda_{\text{mse}}}{N} ||G(w) - I||_2^2 \tag{7.21}$$

where $\mathscr{L}_{\text{percept}}$ is the perceptual loss distance in feature space, which we have introduced many times in the previous sections.

Therefore, the specific form is not repeated. λ_{mse} is employed for balancing the weight ratio between perceptual loss and MSE loss. w can be solved by using the gradient descent algorithm.

The advantage of the optimization-based solution method is that it is more accurate, but the optimization is slow and must be iterated for each image.

After solving the latent encoding vector, we can edit the high-level semantic properties of the face by editing the vector [9], for the StyleGAN architecture, the latent encoding vector can be either Z or W. Generally, encoding based on W will give better results.

7.8 Hands-on Face Attribute Editing Based on StyleGAN Model

In Chap. 5, we have explained the face generation based on the StyleGAN model, in this chapter we practice face attribute editing based on the StyleGAN model, in this section we also use the open source project in

Chap. 5, Sect. 5.9. Depending on the number of input images, face attribute editing can be divided into editing of a single face and editing of multiple faces [8, 9].

7.8.1 Face Reconstruction

To use StyleGAN for face editing, we first need to project the face into the latent coding vector space. The approach we use in this section is based on an optimization approach, i.e., for each face image, the latent coding vector is solved optimally for each face image individually, and the basic idea has been introduced in Sect. 7.7.2.

7.8.1.1 Optimization Objectives

The optimization objective is based on Eq. (7.21), and the noise regularization loss is added, and the code related to the calculation of the loss is briefly explained below.

Firstly, the calculation of perceptual loss, which requires the use of pre-trained models like AlexNet, VGG, etc. The VGG model is defined as follows:

```
import torch
from torchvision import models as tv
## VGG model definition
class vgg16(torch.nn.Module).
def __init__(self, requires_grad=False, pretrained=True).
super(vgg16, self). __init__()
vgg_pretrained_features = tv.vgg16(pretrained=pretrained).features
self.slice1 = torch.nn.Sequential()
self.slice2 = torch.nn.Sequential()
self.slice3 = torch.nn.Sequential()
self.slice4 = torch.nn.Sequential()
self.slice5 = torch.nn.Sequential()
self.N_slices = 5
for x in range(4).
self.slice1.add_module(str(x), vgg_pretrained_features[x])
for x in range(4, 9).
self.slice2.add_module(str(x), vgg_pretrained_features[x])
for x in range(9, 16).
self.slice3.add_module(str(x), vgg_pretrained_features[x])
for x in range(16, 23).
self.slice4.add_module(str(x), vgg_pretrained_features[x])
for x in range(23, 30).
self.slice5.add_module(str(x), vgg_pretrained_features[x])
if not requires_grad.
for param in self.parameters().
param.requires_grad = False
def forward(self, X).
h = self.slice1(X)
h_relu1_2 = h
h = self.slice2(h)
h_relu2_2 = h
h = self.slice3(h)
h_relu3_3 = h
h = self.slice4(h)
h_relu4_3 = h
h = self.slice5(h)
h_relu5_3 = h
vgg_outputs = namedtuple("VggOutputs", ['relu1_2', 'relu2_2', 'relu3_3',
'relu4_3', 'relu5_3'])
out = vgg_outputs(h_relu1_2, h_relu2_2, h_relu3_3, h_relu4_3, h_relu5_3)
```

```
return out
```

The above code block defines the VGG model and outputs the features of different steps in the form of arrays to facilitate feature selection.

In Chap. 5, we introduced the evaluation criterion of StyleGAN, i.e., the perceptual path length, based on which the perceptual loss can be defined, which is also the scheme used for the perceptual loss in this experiment. The difference with the direct calculation in VGG space is that an additional 1 × 1 convolutional layer is needed for dimensional transformation after the features are calculated.

Next we take a look at the definition of a perception network:

```
## Perception Network
class PNetLin(nn.Module).
def __init__(self, pnet_type='vgg', pnet_rand=False, pnet_tune=False,
use_dropout=True, spatial=False, version='0.1', lpips=True).
super(PNetLin, self). __init__()

self.pnet_type = pnet_type ## selected base network
self.pnet_tune = pnet_tune ## whether to do fine tuning
self.pnet_rand = pnet_rand ## whether to use random parameters, otherwise
use pre-trained model
self.spatial = spatial ## whether to calculate metrics separately for
different spatial locations
self.lpips = lpips ## whether to use the lpips criterion, otherwise use the
VGG feature space directly to calculate the loss
self.version = version ## version
self.scaling_layer = ScalingLayer() ## Scale scaling

if(self.pnet_type in ['vgg','vgg16']).
net_type = pn.vgg16
self.chns = [64,128,256,512,512]
self.L = len(self.chns) ## Number of features

## Initialize the model, and the parameters to be trained
self.net = net_type(pretrained=not self.pnet_rand,
requires_grad=self.pnet_tune)

if(lpips).
self.lin0 = NetLinLayer(self.chns[0], use_dropout=use_dropout)
self.lin1 = NetLinLayer(self.chns[1], use_dropout=use_dropout)
self.lin2 = NetLinLayer(self.chns[2], use_dropout=use_dropout)
self.lin3 = NetLinLayer(self.chns[3], use_dropout=use_dropout)
self.lin4 = NetLinLayer(self.chns[4], use_dropout=use_dropout)
self.lins = [self.lin0,self.lin1,self.lin2,self.lin3,self.lin4]

def forward(self, in0, in1, retPerLayer=False).
## v0.0 has no scaling, v0.1 has input scaling
in0_input, in1_input = (self.scaling_layer(in0), self.scaling_layer(in1)) if
self.version=='0.1' else (in0, in1)
## Calculate the features of two images
outs0, outs1 = self.net.forward(in0_input), self.net.forward(in1_input)
feats0, feats1, diffs = {}, {}, {}

## Iterate over the feature layer and calculate the L2 normalized feature
difference
for kk in range(self.L).
feats0[kk], feats1[kk] = util.normalize_tensor(outs0[kk]),
util.normalize_tensor(outs1[kk])
```

```
diffs[kk] = (feats0[kk] - feats1[kk])**2

## Calculate metrics, when for lpips, there are 1*1 layers that need to be
learned
if(self.lpips).
res = [spatial_average(self.lins[kk].model(diffs[kk]), keepdim=True) for kk
in range(self.L)]
else.
res = [spatial_average(diffs[kk].sum(dim=1,keepdim=True), keepdim=True) for
kk in range(self.L)]

val = res[0]
for l in range(1,self.L).
val += res[l]

if(retPerLayer).
return (val, res)
else.
return val
```

where the scaling layer is defined as follows:

```
class ScalingLayer(nn.Module).
def __init__(self).
super(ScalingLayer, self). __init__()
self.register_buffer('shift', torch.Tensor([-.030,-.088,-.188])
[None,:,None,None])
self.register_buffer('scale', torch.Tensor([.458,.448,.450])
[None,:,None,None])

def forward(self, inp).
return (inp - self.shift) / self.scale
```

The 1 × 1 convolution layer is defined as follows:

```
## 1×1 convolutional layer
class NetLinLayer(nn.Module).
def __init__(self, chn_in, chn_out=1, use_dropout=False).
super(NetLinLayer, self). __init__()

layers = [nn.Dropout(),] if(use_dropout) else []
layers += [nn.Conv2d(chn_in, chn_out, 1, stride=1, padding=0, bias=False),]
self.model = nn.Sequential(*layers)
```

After the perceptual loss calculation is defined, it can be called in the high-level perceptual loss class. For detailed code, please refer the reader to Chap. 5 StyleGAN face image generation code.

Next we turn to the definition of noise regularization loss:

```
## Noise regularization loss
def noise_regularize(noises).
loss = 0
for noise in noises.
size = noise.shape[2]
while True.
loss = (
loss
+ (noise * torch.roll(noise, shifts=1, dims=3)).mean().pow(2)
```

```
+ (noise * torch.roll(noise, shifts=1, dims=2)).mean().pow(2)
)

if size <= 8.
break

noise = noise.reshape([-1, 1, size // 2, 2, size // 2, 2])
noise = noise.mean([3, 5])
size //= 2
return loss
```

This one loss is actually the common total variation (TV) loss, which is used in all kinds of image generation tasks to smooth out noise, help obtain smoother image reconstruction results, and increase the generalization ability of the model, and the above code also uses a multi-scale implementation when it is implemented.

7.8.1.2 Face Reconstruction

Next we take a look at the solution for face reconstruction, with the following core code:

```
if __name__ == "__main__".
## Pre-trained model weights
parser.add_argument(
"--ckpt", type=str, required=True, help="path to the model checkpoint"
)

## Output image size
parser.add_argument(
"--size", type=int, default=256, help="output image sizes of the generator"
)

## Learning Rate Parameters
parser.add_argument(
"--lr_rampup".
type=float.
default=0.05.
help="duration of the learning rate warmup".
)
parser.add_argument(
"--lr_rampdown".
type=float.
default=0.25.
help="duration of the learning rate decay".
)
parser.add_argument("--lr", type=float, default=0.1, help="learning rate")

## Noise related parameters, noise level, noise attenuation range, noise
regularization
parser.add_argument(
"--noise", type=float, default=0.05, help="strength of the noise level"
)
parser.add_argument(
"--noise_ramp".
type=float.
default=1.0.
help="duration of the noise level decay".
)
parser.add_argument(
```

```
"--noise_regularize".
type=float.
default=10000.
help="weight of the noise regularization".
)

## MSE loss
parser.add_argument("--mse", type=float, default=0.5, help="weight of the
mse loss")

## Number of iterations
parser.add_argument("--step", type=int, default=1000, help="optimize
iterations")

## Reconstructed image
parser.add_argument(
"--files", type=str, help="path to image files to be projected"
)

## Reconstruction results
parser.add_argument(
"--results", type=str, help="path to results files to be stored"
)

## Calculate learning rate
def get_lr(t, initial_lr, rampdown=0.25, rampup=0.05).
lr_ramp = min(1, (1 - t) / rampdown)
lr_ramp = 0.5 - 0.5 * math.cos(lr_ramp * math.pi)
lr_ramp = lr_ramp * min(1, t / rampup)

return initial_lr * lr_ramp

## Merge latent vectors and noise
def latent_noise(latent, strength).
noise = torch.randn_like(latent) * strength
return latent + noise

## Generate images
def make_image(tensor).
return (
tensor.detach()
.clamp_(min=-1, max=1)
.add(1)
.div_(2)
.mul(255)
.type(torch.uint8)
.permute(0, 2, 3, 1)
.to("cpu")
.numpy()
)

## Generate noise equal to the size of the image
def make_noise(device,size).
noises = []
step = int(math.log(size, 2)) - 2
for i in range(step + 1).
size = 4 * 2 ** i
```

```
noises.append(torch.randn(1, 1, size, size, device=device))
return noises

## Noise normalization
def noise_normalize_(noises).
for noise in noises.
mean = noise.mean()
std = noise.std()
noise.data.add_(-mean).div_(std)

args = parser.parse_args()

device = "cpu"
## Calculate the average number of latent vectors
n_mean_latent = 10000

## Get the minimum image size used to calculate the loss
resize = min(args.size, 256)

## Preprocessing functions
transform = transforms.Compose(
[
transforms.Resize(resize).
CenterCrop(resize).
ToTensor().
Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]).
]
)

## projection of the face image, the image will be processed into a batch
imgs = []
imgfiles = os.listdir(args.files)
for imgfile in imgfiles.
img = transform(Image.open(os.path.join(args.files,imgfile)).convert("RGB"))
imgs.append(img)

imgs = torch.stack(imgs, 0).to(device)

## Load Model
netG = StyledGenerator(512,8)
netG.load_state_dict(torch.load(args.ckpt,map_location=device)["g_running"],
strict=False)
netG.eval()
netG = netG.to(device)
step = int(math.log(args.size, 2)) - 2
with torch.no_grad().
noise_sample = torch.randn(n_mean_latent, 512, device=device)
latent_out = netG.style(noise_sample) ## input noise vector Z, output latent
vector W = latent_out
latent_mean = latent_out.mean(0)
latent_std = ((latent_out - latent_mean).pow(2).sum() / n_mean_latent) **
0.5

## Perceived loss calculation
percept = lpips.PerceptualLoss(
model="net-lin", net="vgg", use_gpu=device.startswith("cuda")
)
```

```
## Build noise input
noises_single = make_noise(device,args.size)

noises = []
for noise in noises_single.
noises.append(noise.repeat(imgs.shape[0], 1, 1, 1).normal_())

## Initialize the Z vector
latent_in = latent_mean.detach().clone().unsqueeze(0).repeat(imgs.shape[0],
1)
latent_in.requires_grad = True

for noise in noises.
noise.requires_grad = True

optimizer = optim.Adam([latent_in] + noises, lr=args.lr)

pbar = tqdm(range(args.step))

## Optimal learning of Z-vectors
for i in pbar.
t = i / args.step ## The range of t is (0,1)
lr = get_lr(t, args.lr)
optimizer.param_groups[0]["lr"] = lr

## Noise attenuation
noise_strength = latent_std * args.noise * max(0, 1 - t / args.noise_ramp)
** 2
latent_n = latent_noise(latent_in, noise_strength.item())
latent_n.to(device)
img_gen = netG([latent_n], noise=noises, step=step) ## Generated image
batch, channel, height, width = img_gen.shape

## Calculate loss at resolutions up to 256
if height > 256.
factor = height // 256

img_gen = img_gen.reshape(
batch, channel, height // factor, factor, width // factor, factor
)
img_gen = img_gen.mean([3, 5])

p_loss = percept(img_gen, imgs).sum() ## perceptual loss
n_loss = noise_regularize(noises) ## Noise loss
mse_loss = F.mse_loss(img_gen, imgs) ## MSE loss

loss = p_loss + args.noise_regularize * n_loss + args.mse * mse_loss

optimizer.zero_grad()
loss.backward()
optimizer.step()

noise_normalize_(noises)

pbar.set_description(
(
f "perceptual: {p_loss.item():.4f}; noise regularize: {n_loss.item():.4f};"
f" mse: {mse_loss.item():.4f}; loss: {loss.item():.4f}; lr: {lr:.4f}"
```

```
)
)

## Regenerate high resolution images
img_gen = netG([latent_in], noise=noises,step=step)
img_ar = make_image(img_gen)

result_file = {}
for i, input_name in enumerate(imgfiles).
noise_single = []
for noise in noises.
noise_single.append(noise[i : i + 1])

print("i="+str(i)+"; len of imgs: "+str(len(img_gen)))
result_file[input_name] = {
"img": img_gen[i].
"latent": latent_in[i].
"noise": noise_single.
}

img_name = os.path.join(args.results,input_name)
pil_img = Image.fromarray(img_ar[i])
pil_img.save(img_name) ## Store the image
np.save(os.path.join(args.results,input_name.split('.')
[0]+'.npy'),latent_in[i].detach().numpy())## store the latent vector
```

In the above code, latent_in is the latent code to be optimized for learning. When using the call of netG([latent_n], noise=noises, step=step), latent_n is the vector Z of the mapping network, which consists of latent_in and the noise vector that decays with iteration, and there is no input average style vector at this time.

According to the interpretation of the structure of the generator model in Sect. 5.9, only latent_n affects the generated style at this point. Therefore, it is a vector Z-based reconstruction method.

Of course, we can also set latent_n as the average style vector and set the blend weight style_weight to 0, so that only latent_n will affect the generated style and it will be used as the input of the synthetis network, i.e., vector W. This is the reconstruction method based on vector W.

In the following, we will compare the differences between these two reconstruction methods.

The learning rate is increased and then decreased by a warmup strategy, with the maximum not exceeding 0.1.

The weights of perceptual loss, MSE loss, and noise regularization loss are 1.0, 1.0,10,000, and the amplitude and attenuation range factors of noise correlation are 0.05 and 1, respectively.

Figure 7.19 shows some results of the face reconstruction.

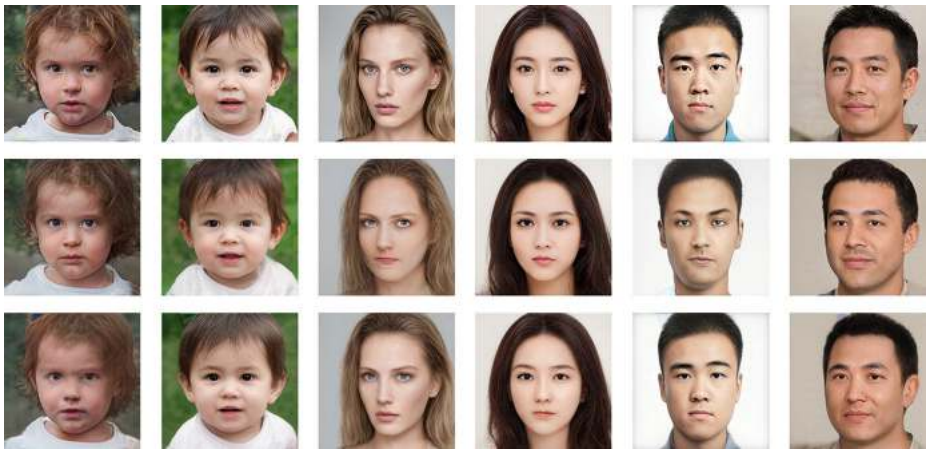


Fig. 7.19 Face reconstruction

The first row in Fig. 7.19 shows the original image, the second row shows the reconstructed image based on vector Z, and the third row shows the reconstructed image based on vector W.

Figure 7.20 shows a 3-part training loss profile of an image based on Z and based on W vectors, where the solid line corresponds to W and the dashed line corresponds to Z. It should be noted that the noise loss is not multiplied by the corresponding weights, and if multiplied by the corresponding weights, the magnitude of the three parts of the loss is equivalent at the final convergence.

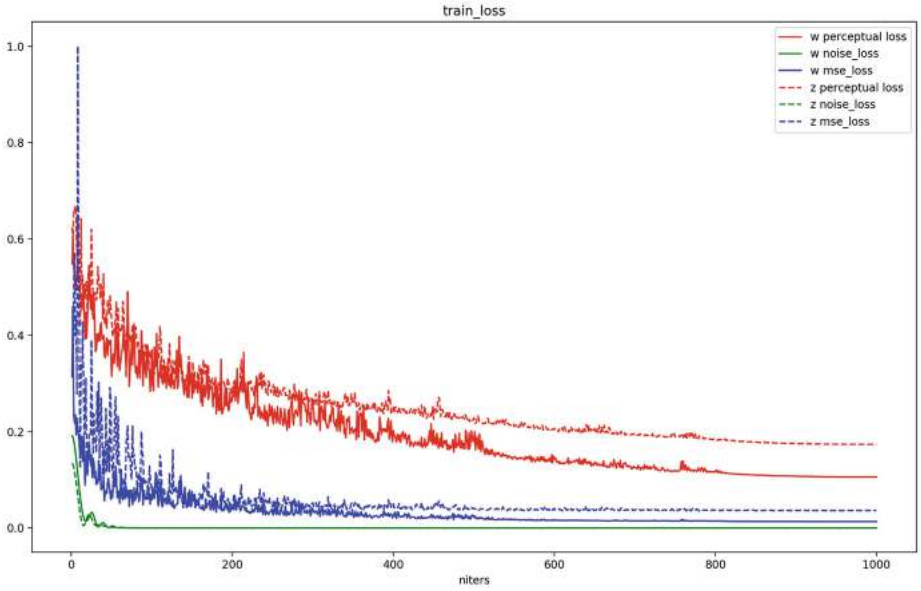


Fig. 7.20 Face reconstruction loss curve

From the results, the overall pose, skin color, hair style, face shape, and background reconstruction of the face images are good in both methods, and the face is clearer in the vector Z-based reconstruction method, but the identity is not maintained. This is mainly because the feature vector Z needs to go through a nonlinear mapping network to get W, which is more difficult than learning W directly, and many studies have shown that using W vectors can get better reconstruction results.

From the loss curves, it can be seen that the training loss of W vector-based reconstruction can obtain lower values, but the perceptual loss converges more slowly and the actual loss of MSE is lower, which can be seen that a certain perceptual quality is sacrificed on the basis of obtaining a more accurate identity reconstruction, making it sensitive when editing the W vector.

Next we use the vector Z-based reconstruction results for face attribute editing because this allows us to compare the difference between editing attributes based on Z and W. The corresponding Z vectors are not available in the vector W-based reconstruction results because the mapping network is unable to obtain inputs from the outputs.

7.8.2 Face Attribute Blending and Interpolation

Next we perform blending and interpolation of face attributes, which is a blending operation of attributes between multiple images.

7.8.2.1 Face Attribute Style Mixing

We first experience face attribute style blending, and the style blending operation can be realized by the vector operation in the following formula:

$$W = \lambda\{W_1[0:m], (1-\lambda)W_2[m:n]\} \tag{7.22}$$

where $W_1[0:m]$ denotes the first m dimensions of the vector, and $W_2[m:n]$ denotes the mth to nth dimension of the vector, and the two are spliced to obtain the new vector W.

It is worth noting that here the W is not the output vector of the mapping network W, but rather the AdaIN scaling and offset coefficients for the different stylized layers, i.e., the style coefficients. As we introduced in Chap. 5, the style modules of different resolutions correspond to different layers of face features, and here we use the style coefficients corresponding to two face images for blending to experience the style blending of different layers of features.

Column 1 in Fig. 7.21 indicates the source map, column 5 indicates the target map, and columns 2, 3, and 4 indicate that the style of the source map is used in the style layer of the corresponding resolution, and the style of the target map is used in the style layer of the other resolutions.

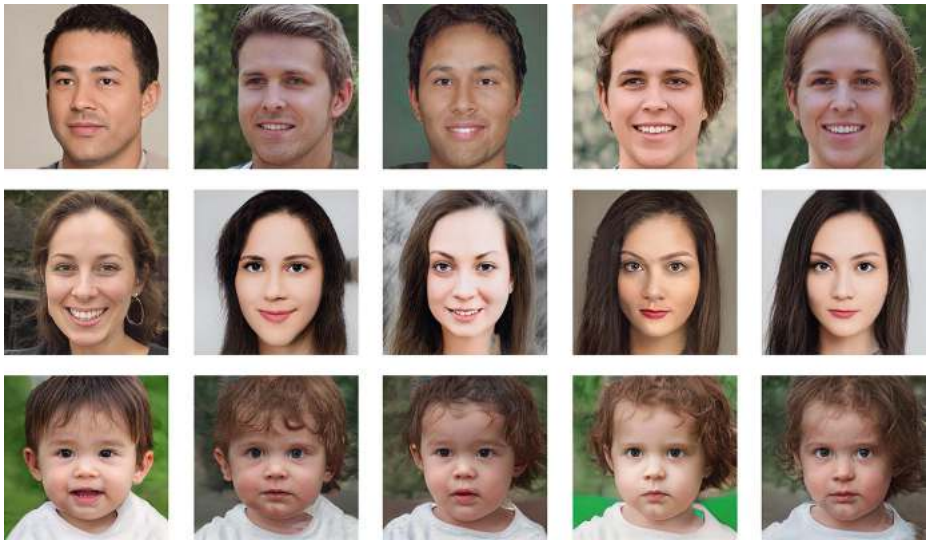


Fig. 7.21 Face style blending

Column 2 indicates that the style vectors of the stylized module with resolution 4 × 4 and 8 × 8 are from the source map, and the style vectors of the other resolution modules are from the target map. It can be seen that the result map has coarse-grained features from the source map, such as face pose and hairstyle features, and fine-grained features from the target map, such as hair and eye color.

Column 3 indicates that the stylized modules with resolutions of 16 × 16 and 32 × 32 have style vectors from the source map, and the style vectors of the other resolution modules are from the target map. It can be seen that the medium granularity features of the source map, such as eye morphology, lip color, and other features, are retained.

Column 4 indicates that the style vectors of the stylized module with resolutions between 64 × 64 and 1024 × 1024 are from the source graph, and the style vectors of the other resolution modules are from the target graph. It can be seen that the result map has the fine-grained features of the source map, such as hair and skin color and texture, and the coarse-grained features of the target map, such as pose and hair style.

7.8.2.2 Face Style Interpolation

Next, we experience face style interpolation, which can be achieved by the following W vector operation.

The operation of style interpolation is shown in Eq. 7.23.

$$W = \lambda W_1 + (1 - \lambda) W_2 \tag{7.23}$$

Figures 7.22 and 7.23 show the results of face style interpolation based on vector Z and vector W, respectively.

Fig. 7.22 Face style interpolation based on Z vector

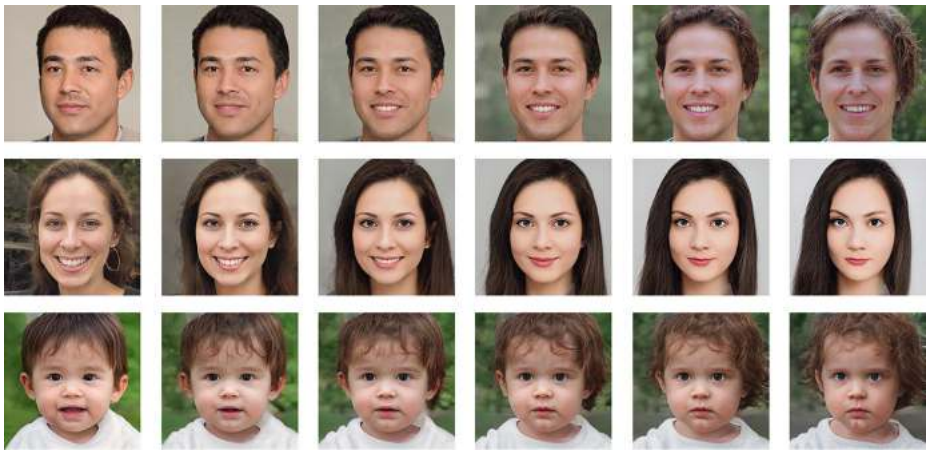


Fig. 7.23 Face style interpolation based on W vector

The first column represents the A-domain image, corresponding to the n-dimensional column vector W_1. Column 6 represents the B-domain image, corresponding to the n-dimensional column vector W_2, and columns 2, 3, 4, and 5 denote the weighting of λ under different weights for W_1 and W_2. The images generated after weighting the λ = 0.8, 0.6, 0.4, 0.2. It can be seen that the results implemented smooth transition of styles, but the results based on the W vector are significantly smoother than those based on the Z vector.

7.8.3 Face Attribute Editing

The face style blending introduced in the previous section can directly achieve the blending of face attributes by the latent vector operation of two images, while if you want to edit the attributes of a single face precisely, you need to first find the editing direction of the latent vector, called the direction vector. In this section, we introduce the solution of the direction vector and the attribute editing based on the direction vector.

7.8.3.1 Basic Principle

The next attribute editing is based on the following assumptions: if the latent encoding vector is changed linearly in the direction vector, the generated image, and the semantic content also change continuously, so that attribute changes can be made using a linear model:

$$W = W_0 + \alpha n \tag{7.24}$$

W denotes the result code, and W_0 denotes the face feature code, and α denotes the offset coefficient, and n denotes the direction vector.

The next problem to be solved is the solution of the direction vector n. The specific steps are as follows:

1. Randomly sample the latent encoding vectors to generate face images, save the face images and the corresponding latent encoding vectors.
2. For the generated face images, the CNN classification model of the face attributes that we want to edit is trained. Any binary semantics has a hyperplane that can be used as a classification boundary for the semantic category, and changing the latent encoding vector on one side of the hyperplane does not change the corresponding semantic category, and this hyperplane can be represented by a unit normal vector.
3. Based on the labels obtained from the CNN classification model, a logistic regression model is trained on the latent coding vectors to obtain the direction vectors, as shown in Fig. 7.23.

Figure 7.24 shows the two types of samples, 0 and 1, which are the latent coding vectors W. The weights of the solved logistic regression model $\overrightarrow{w}$, it is actually the direction vector $\hat{n}$. Once the direction vector is obtained, the face-related attributes can be edited using Eq. (7.24).

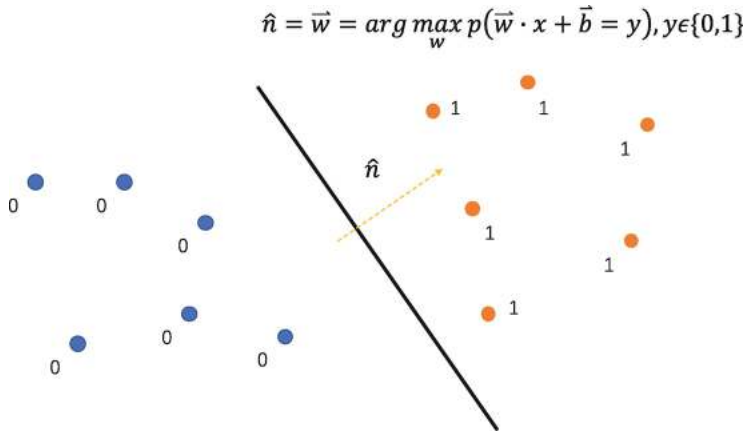


Fig. 7.24 Classification boundarie and direction vector

7.8.3.2 Face Expression Editing

Next we performed face attribute editing, first we generated 50,000 face images randomly by using StyleGAN, next we classify the images into two categories: with and without expressions, and it needs to use a pre-trained 2-category expression recognition model. To ensure that the model has high accuracy, the method used for training is to first extract the lip region of the face and then train the lip region.

Since this task is relatively simple, we decided to design a simple network called simpleconv3. The network contains three layers of convolution, three fully connected layers, each convolutional layer has a kernel size of 3 × 3, a step size of 2, a padding of 1, and the input image size is set to 48 × 48.

The configuration of the convolutional layers is shown in Table 7.1.

Table 7.1 simpleconv3 network convolutional layer and fully connected layer configuration

Network layer	Input feature map size	Output feature map size	Convolution kernel size	Stride size	Filling size
conv1	3 × 48 × 48	12 × 24 × 24	3 × 3	2	1
conv2	12 × 24 × 24	24 × 12 × 12	3 × 3	2	1
conv3	24 × 12 × 12	48 × 6 × 6	3 × 3	2	1

The number of neurons in the three fully connected layers is 512, 128, and 4, respectively.

The structure of the training network after visualizing the network in Caffe format using the Netron tool is shown in Fig. 7.25.

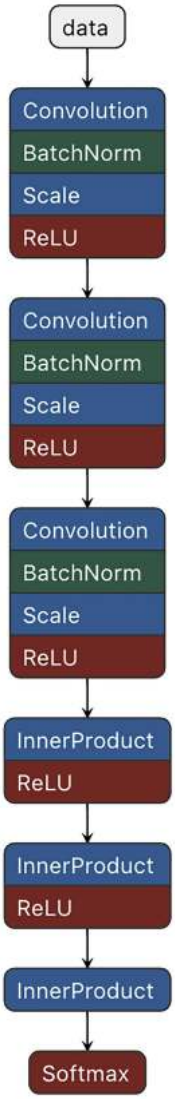


Fig. 7.25 Simple expression recognition model

It can be seen that the structure consists of three groups of convolution blocks, each containing a convolution layer Convolution, a normalization layer BatchNorm, a scale layer Scale, an activation layer ReLU. And three fully connected layers are included, the first two fully connected layers are followed by the ReLU activation layer, the last fully connected layer is used as the classification output layer, and finally a Softmax layer is added for normalizing the probabilities.

Figure 7.26 shows some samples of images with and without smiles classified based on the smile expression model.

Fig. 7.26 Emotion picture case

After training the model, we get the labels of the images, and then learn a linear classifier for their corresponding latent vectors to get the direction vectors, and the core code is as follows:

```
## Import logistic regression model functions
from sklearn.linear_model import LogisticRegression

## Constructing datasets
import glob
posdir = sys.argv[1]
negdir = sys.argv[2]
possamples = glob.glob(os.path.join(posdir,'*.npy'))
negsamples = glob.glob(os.path.join(negdir,'*.npy'))
x_features = []
y_label = []
for sample in possamples.
feature = np.squeeze(np.load(sample))
x_features.append(list(feature))
y_label.append(1)
for sample in negsamples.
feature = np.squeeze(np.load(sample))
x_features.append(list(feature))
y_label.append(0)
x_features = np.array(x_features)
y_label = np.array(y_label)

print("x_features="+str(x_features.shape))
print("y_label="+str(y_label.shape))

#x_features = np.array([[-1, -2], [-2, -1], [-3, -2], [1, 3], [2, 1], [3,
2]])
#y_label = np.array([0, 0, 0, 1, 1, 1])

## Calling logistic regression models
lr_clf = LogisticRegression()

## Fitting a constructed data set with a logistic regression model
lr_clf = lr_clf.fit(x_features, y_label) # Its fitting equation is
y=w0+w1*x1+w2*x2

## View the w of its corresponding model
print('the weight of Logistic Regression:',lr_clf.coef_)
np.save('w.npy',lr_clf.coef_)

## View the w0 of its corresponding model
```

```
print('the intercept(w0) of Logistic Regression:',lr_clf.intercept_)
```

Because the latent vector dimension is 1 × 512, the obtained direction vector is also 1 × 512 dimension, and then we can edit the attributes based on the direction vector. The core code for Z-vector based editing is as follows:

```
from model import StyledGenerator

if __name__ == "__main__".
device = "cpu"
parser.add_argument(
"--ckpt", type=str, required=True, help="path to the model checkpoint"
)
parser.add_argument(
"--size", type=int, default=1024, help="output image sizes of the generator"
)
parser.add_argument(
"--files", type=str, help="path to image files to be projected"
)
parser.add_argument(
"--direction", type=str, help="direction file to be read"
)
parser.add_argument(
"--directionscale", type=float, help="direction scale"
)
args = parser.parse_args()

## Load Model
netG = StyledGenerator(512)
netG.load_state_dict(torch.load(args.ckpt,map_location=device)["g_running"],
strict=False)
netG.eval()
netG = netG.to(device)
step = int(math.log(args.size, 2)) - 2

## Load direction vectors
direction = np.load(args.direction)
directiontype = args.direction.split('/')[-1].split('.') [0]
editscale = args.directionscale
npys = glob.glob(args.files+"*.npy")

for npyfile in npys.
latent = torch.from_numpy(np.load(npyfile))
if len(latent.shape) == 1.
latent = latent.unsqueeze(0)
latent = latent + torch.from_numpy((editscale*direction[0]).astype(np.float32)
latent.to(device)
img_gen = netG([latent], step=step) ## generated images
img_name =
os.path.join(npyfile.replace('.npy','_'+directiontype+'_'+str(editscale)+'.jpg
utils.save_image(img_gen, img_name, normalize=True)
np.save(img_name.replace('.jpg','.npy'),latent)
```

Figures 7.27 and 7.28 show the editing results based on Z-vectors and W-vectors, respectively.

Fig. 7.27 Z-vector-based editing of face smile attributes, $\alpha = 0.5$

Fig. 7.28 W vector-based editing of face smile attributes, $\alpha = 0.05$

The first row shows the original image, the second row shows the decrease of smile expression amplitude, and the third row shows the increase of smile expression amplitude.

It can be seen that it is able to achieve the editing of smile expressions based on Z vector and W vector. However, the model based on Z vector obviously changes other attributes, such as hairstyle and face identity. This indicates that the editing based on vector Z cannot achieve the decoupling of expression attributes from other attributes well, and there is large room for improvement, while the model based on W vector can protect other attribute information better.

7.8.3.3 Adding and Removing Face Attributes

Since different faces have different attributes, we can also add and remove attributes based on this, and the W vector operation is shown in Eq. (7.25), where W_3 and W_2 come from the same person and are subtracted to get a vector of a certain attribute, which is then added to W_1, which means the corresponding attribute of W_3 is added to the face of W_1.

$$W = W_1 + \lambda(W_3 - W_2) \tag{7.25}$$

Figures 7.29 and 7.30 show the results of adding and removing the face smile attribute based on the Z vector and W vector, respectively.

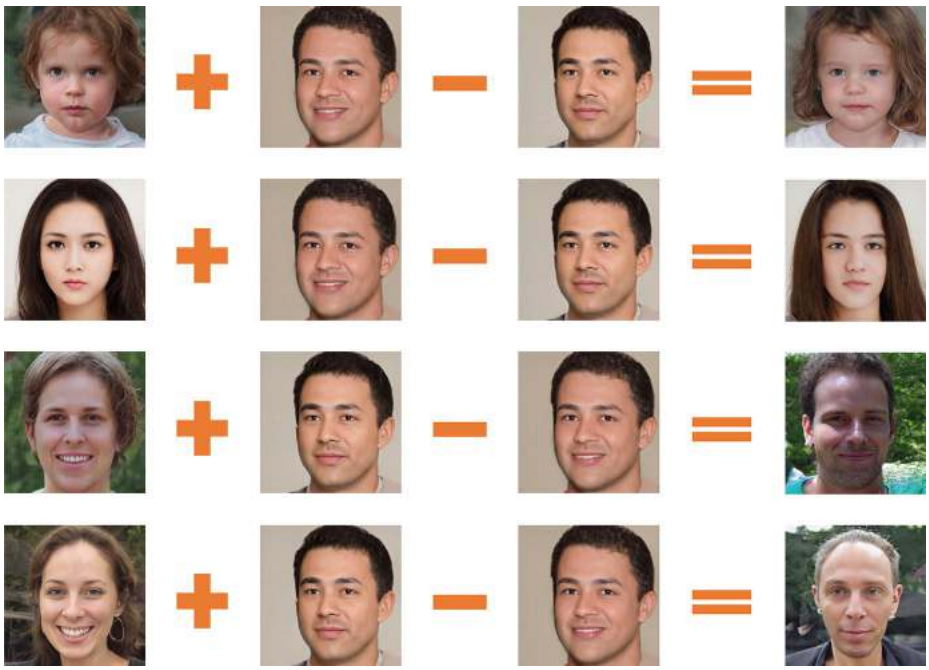


Fig. 7.29 Adding and removing face attributes based on Z vectors, $\lambda = 1.5$

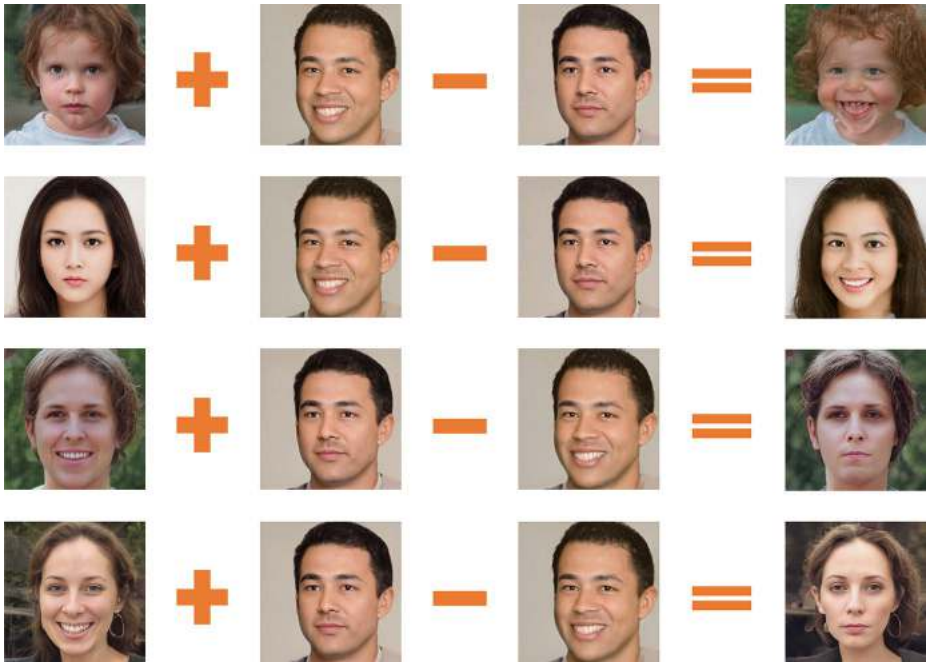


Fig. 7.30 Adding and removing face attributes based on W vectors, $\lambda = 1$

The first column represents the source image, corresponding to the n-dimensional vector W_1. Columns 2 and 3 denote the target image, respectively, corresponding to the n-dimensional vectors W_3 and W_2. Column 4 represents the generated image, and it can be seen that although the Z vector-based can achieve the addition and removal of the smile attribute to some extent; it will modify the identity of the person. The W vector can be used to add and remove the smile attribute well.

7.8.4 Summary

In this section, we practiced face attribute editing using the StyleGAN model and achieved the expected experimental results, but there are also some areas that can be improved, mainly including the following points.

For one, it is difficult to maintain the identity information of a face in face reconstruction tasks. However, for face images, identity is very important. If one wants to improve this, one can add identity-related loss to train the face reconstruction task based on the face recognition model.

Second, in face attribute editing, irrelevant attributes also change, especially the Z-vector-based editing, which does not have better attribute decoupling properties than the W-vector after mapping network transformation. When using W vectors for reconstruction, we can refer to the work of frameworks such as StyleGAN v2 and expand it from 1 × 512 dimensions to 18 × 512 dimensions, which can achieve more flexible attribute editing results.

Third, since the StyleGAN model is used in this experiment, there are relatively obvious flaws in the generation of faces, readers can use the updated StyleGAN v2, StyleGAN v3 to get higher quality of image generation. In addition, the model used in this experiment is not the official open source model, and it will have better results if the official NIVDIA open source model is used.

Fourth, unsupervised face attribute editing schemes are used, such as the GANSpace framework based on PCA principal component analysis [10] and the SeFa framework based on eigenspace matrix decomposition [11], which can search several optimal direction vectors on their own and thus editing some attributes that are difficult to train classifiers to obtain direction vectors, such as attributes like hairstyle, which are left to the reader to conduct experiments.

References

1. Song L, Lu Z, He R, et al. Geometry guided adversarial facial expression synthesis[C]//Proceedings of the 26th ACM international conference on Multimedia. 2018: 627–635.
2. Zhang Z, Song Y, Qi H. Age progression/regression by conditional adversarial autoencoder[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2017: 5810–5818.
3. Shen Y, Luo P, Yan J, et al. Faceid-gan: learning a symmetry three-player gan for identity-preserving face synthesis[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018: 821–830.
4. Yin X, Yu X, Sohn K, et al. Towards large-pose face frontalization in the wild [C]//Proceedings of the IEEE International Conference on Computer Vision. 2017: 3990–3999.
5. Kim J, Kim M, Kang H, et al. U-gat-it: Unsupervised generative attentional networks with adaptive layer-instance normalization for image-to-image translation[J]. arXiv preprint arXiv:1907.10830, 2019.
6. Li T, Qian R, Dong C, et al. Beautygan: Instance-level facial makeup transfer with deep generative adversarial network[C]//Proceedings of the 26th ACM international conference on Multimedia. 2018: 645–653.
7. https://github.com/deepfakes/faceswap
8. Abdal R, Qin Y, Wonka P. Image2stylegan: How to embed images into the stylegan latent space?[C]//Proceedings of the IEEE/CVF international conference on computer vision. 2019: 4432–4441.
9. Shen Y, Gu J, Tang X, et al. Interpreting the latent space of gans for semantic face editing[C]//Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2020: 9243–9252.
10. Härkönen E, Hertzmann A, Lehtinen J, et al. Ganspace: Discovering interpretable gan controls[J]. Advances in neural information processing systems, 2020, 33: 9841–9850.
11. Shen Y, Zhou B. Closed-form factorization of latent semantics in gans[C]//Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2021: 1532–1540.

P. Long, X. Guo, *Generative Adversarial Network*
https://doi.org/10.1007/978-981-96-9404-4_8

8. Image Quality Enhancement

Peng Long[1] ✉ and Xiaozhou Guo[2]
(1) Beijing YouSan Educational Technology, Beijing, China
(2) • China Electronics Technology Group Corporation No. 54 Research Institute, Shijiazhuang, China

Abstract
This chapter discusses GAN applications in image quality enhancement, including denoising, deblurring, tone mapping, super-resolution, and restoration. Denoising frameworks like GCBD use GANs to synthesize realistic noise for training. Deblurring models (e.g., DeblurGAN and DBGAN) combine perceptual and adversarial losses with multi-scale discriminators. Tone mapping leverages paired (e.g., MIT-Adobe FiveK) or unpaired datasets (e.g., CycleGAN-based unsupervised enhancement). Super-resolution (e.g., SRGAN) employs VGG-based perceptual loss. Restoration models address inpainting and artifact removal. The chapter contrasts simulated vs. real-world datasets (e.g., RENOIR, GoPro) and highlights challenges in generalization and real-time processing. Finally, Practical implementations of SRGAN are demonstrated, including code interpretation and training details.

Keywords Image enhancement – Denoising – Deblurring – Super-resolution – Image restoration – SRGAN – Unsupervised learning

Previously, we introduced the basic framework of GAN in image generation and translation tasks. GAN has also been widely applied in many low-level image processing tasks. In this chapter, we introduce some typical technical frameworks of GAN in image quality enhancement, and the reader can extend the learning based on the related contents.

8.1 Image Noise Reduction

Images are disturbed by noise during both generation and transmission, so image noise reduction is a fundamental problem, and the generative model GAN has a natural advantage in capturing the distribution of noise. In this section, we introduce a typical framework for GAN-based image noise reduction.

8.1.1 Image Noise Reduction Problem

Image noise is the presence of unnecessary or redundant interference information in image data, more broadly defined as "unpredictable random error," which can be described by using random processes and characterized by probability distribution functions and probability density distribution functions. The presence of noise seriously affects the image quality, such as contaminating the edges of the image and affecting the distribution of grayscale, thus hindering the understanding of the image by people and computers.

The first image from the left in Fig. 8.1 is a medical image with noise, the second image is a noisy night image, and the third image is a raindrop occlusion image. Image noise reduction is to recover the original noise-free image.

Fig. 8.1 Common images with noise

The current denoised dataset is mainly built in two ways.

The first way: obtain high-quality images from existing image database, then perform image processing (such as brightness adjustment) and add artificially synthesized noise according to the noise model to generate simulated noisy images. This type of method is relatively simple and time-saving, and high-quality images can be obtained directly from the Internet, However, since the noise is artificially synthesized, it differs from the real noisy image, which makes the denoising process of the network which is trained on this dataset have limited effect on the real noisy image.

Most of the early studies used simulation datasets, and the representative dataset is Tampere Image Database (TID2013) [1]. This dataset contains 25 reference images by using 24 different contamination methods, including additive and multiplicative Gaussian noise, high-frequency noise, and coding errors, each containing five degree levels, and the final processing yields 3000 images.

The second way: for the same scene, take a low ISO image as the true value and a high ISO image as the noise image, and adjust the camera parameters such as exposure time to make the brightness of the two images the same, so that a real scene dataset can be obtained. A typical representative of this is the RENOIR dataset [2], which was built by first taking 120 images of dark-light scenes, which contains both indoor and outdoor scenes. Four images were taken for each scene, which contains two noisy images and two low-noise images. The acquisition equipment and the related parameter statistics are shown in Table 8.1.

Table 8.1 RENOIR data set acquisition configuration

Equipment	Light-sensitive element size (mm)	Quantity	Low-noise image ISO	Low-noise image sensing time	Noise images ISO	Noise images sensing time	Image size
Xiaomi Mi3	4.69 × 3.52	40	100	Auto	1.6k,3.2k	Auto	4208 × 3120
Canon S90	7.4 × 5.6	40	100	3.2	640,1k	Auto	3684 × 2760
Canon T3i	22.3 × 14.9	40	100	Auto	3.2k,6.4k	Auto	5202 × 3465

It can be seen that low-noise images, i.e., images that are taken as true, are acquired using low ISO and also have a long exposure time, while high-noise images are acquired using two stops of higher ISO equipment. For low-noise images, the same configuration is acquired twice, one at the very beginning and the other is acquired after the high-noise image, and the image is discarded if the PSNR is below 34.

Once a high-quality input-output image pair (noisy/noise-free images pair) is obtained and a training dataset is built, a supervised learning-based approach can be used to learn the noise reduction model.

The quality of the dataset has a great impact on the denoising results, and how to obtain data with as many scenes as possible and high-quality reference images (Ground Truth) is a hot topic of current research.

8.1.2 GAN-Based Image Denoising Framework

A major challenge faced by deep learning-based image noise reduction is the difficulty in obtaining large amounts of paired real noise and noise-free data. In the two noise acquisition schemes in Sect. 8.1.1, the simulation dataset requires mathematical modeling of complex noise, which is difficult to simulate real scenes. The acquisition of real datasets, on the other hand, is more demanding in terms of both equipment and environment. While they each have drawbacks, generative adversarial networks can be used to generate real datasets, thus reducing both the cost of data acquisition and obtaining higher quality data. Next, we present a typical solution to the image noise reduction problem based on GAN.

GCBD (GAN-CNN-Based Blind Denoiser) [3] method uses GAN to capture noise from real noisy images to obtain real pairwise maps for noise reduction model training, and the whole algorithm flow is schematically shown in Fig. 8.2.

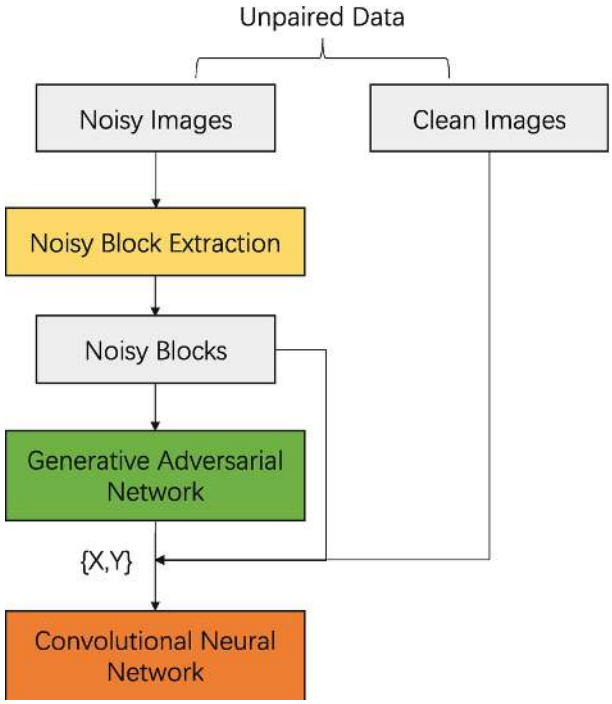


Fig. 8.2 GAN-based unsupervised denoising model

As can be seen from Fig. 8.2, a set of "unpaired" Noisy Image and Clean Image are given first, and then it uses a noise block extraction network to extract approximate noise blocks from noisy images to train a generative adversarial network for noise modeling and sampling. A large number of noise blocks are then sampled from the trained GAN model, and these noise blocks are then combined with clear images to obtain paired training data for denoising into a convolutional neural network.

The noise block extraction network selects sub-blocks from noisy images by selecting the smoother sub-block of the image and then subtracting the mean gray value of that sub-block, which can be seen to adopt the assumption of Gaussian additive noise model.

The acquisition of real noise and noise-free images is the key to applying deep learning to the denoising problem, and unsupervised models based on approaches such as GAN are worth focusing on.

8.2 Image Deblurring

Deblurring is also a common basic image problem. Classical deep learning models need to estimate unknown blur kernels, and it is difficult to remove blurs of large magnitude. GAN is now also used to solve the deblurring problems and gains some achievements.

In this section, we introduce a typical framework for GAN-based deblurring.

8.2.1 Image Deblurring Problem

Due to the shaking of the device during the shooting process, inaccurate focus or too fast movement of the target, sometimes we capture images with significant blurring, Fig. 8.3 shows two typical types of blurred image samples.

Fig. 8.3 Out-of-focus and motion blur

Figure 8.3a shows a blurry picture taken by a fixed-focus lens, where the animal's eyes are out of focus due to the camera being too close to the target.

Figure 8.3b shows the blur caused by the rapid movement of the cat. In addition, camera shake can cause similar motion blur, which often occurs when shooting long exposure images without a tripod or other fixed device.

The earliest researchers generated images directly from arbitrary images using different types of blurring kernels, but the trained models did not generalize well due to the large blurring differences from real images. A common type of blur is motion blur, so researchers often use GoPro sports cameras to capture high-speed moving targets and thus build deblurred datasets.

The GoPro dataset [4] is a widely used deblurred dataset currently and the researcher used a GoPro 4 Hero Black camera for the dataset acquisition. The fps used for acquisition is 240, and the image size is 1280x720. Then, continuous 7–13 frames of images are averaged to obtain images with different degrees of blur and clear images in sequence.

For example, taking 15 frames of images for averaging, then the equivalent exposure time of each averaged image is 1/16 s. Taking the middle (i.e., the 8th) image as the clear image and the averaged image as the blurred image, the training image pairs can be obtained. The data set finally contains 3214 blurred and clear image pairs, of which 2103 pairs are used as the training set and the rest as the test set.

Compared with constructing datasets from different kinds of fuzzy kernels, the acquisition scheme of GoPro datasets can construct more realistic datasets and is widely adopted in supervised deblurring algorithms.

8.2.2 GAN-Based Image Deblurring Framework

Here we first take a look at the basic deblurring framework DeblurGAN [5], which has the basic flow shown in Fig. 8.4.

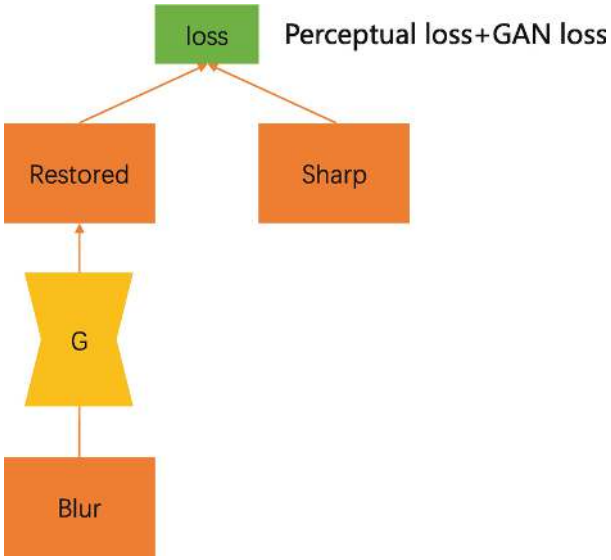


Fig. 8.4 Schematic diagram of DeblurGAN model framework

In Fig. 8.4, Blur is the blurred input image, which is passed through the generator G to generate the deblurred image (Restored), and then compared with the real clear image Sharp to calculate the loss function. The loss includes two parts, which are the perceptual loss and the adversarial loss in the VGG feature space.

Subsequent authors have improved the DeblurGAN framework by proposing DeblurGAN v2 [6]. DeblurGAN v2 uses FPN as the core module of the generator to improve the performance of the generative model. The discriminator, on the other hand, uses least-squares loss, which is measured in terms of both global and local scales. The authors believe that for highly non-uniform blurred images containing complex target motion, the global scale helps the discriminator to integrate contextual information of the whole image, thus enabling larger and more complex real blurred images to be processed compared to DeblurGAN.

Most current deblurring frameworks use simulated blurred and unblurred image pairs for training, where the simulated blur is often weighted with multiple consecutive frames, but this is not the same as the real blur generation mechanism and does not take into account the camera response function, which is not a time-series smooth function, so the model does not generalize well to real scenes. Real blur generation often contains many factors, such as the aforementioned out-of-focus, camera shake, and rapid target motion.

Similar to the image noise reduction problem, DBGAN is a deblurring framework for learning blur types [7], which uses two GANs, one for learning to generate blur and one for learning to deblur, and the whole framework is shown in Fig. 8.5.

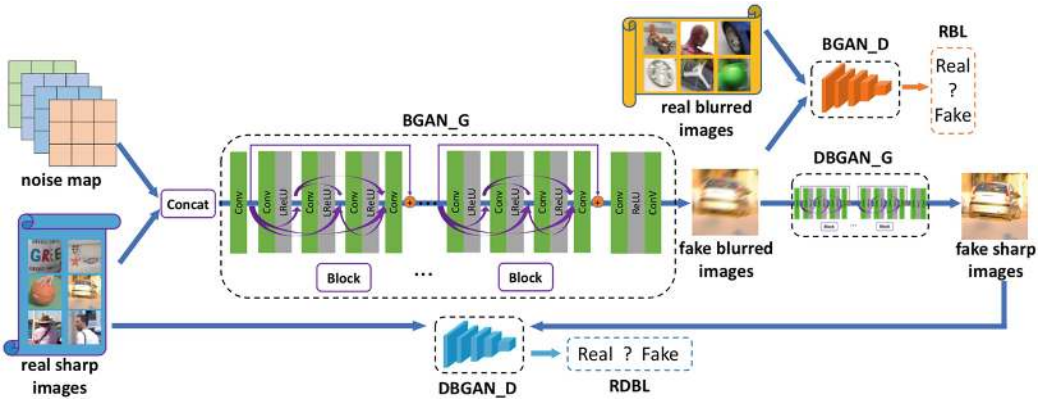


Fig. 8.5 Schematic diagram of DBGAN model framework

Figure 8.5 contains two modules, a generation module BGAN (learning-to-Blur GAN (BGAN) and a deblur module DBGAN (learning-to-DeBlur GAN).

BGAN first performs simulation learning for blurring, where the input is a stitching of real blur-free images and random noise maps, and the network structure itself is a module based on residual block that does not change the spatial resolution size of the input maps. BGAN provides blurred images and unblurred image pairs for the next deblurred DBGAN, which solves the previous problem of using simulation data with large differences from the real blurred data.

Most of the current deblurring models are not yet able to achieve relatively ideal results for real images and are not practical, and further research is needed.

8.3 Image Tone Mapping

Tone Mapping focuses on the global and local adjustment of image color, including brightness, hue and so on. Current researchers have proposed many tone mapping deep learning models, but none of them can enhance all kinds of images perfectly yet. GAN can learn rich adjustment patterns because it can capture real data distribution. Therefore some good results have been achieved.

In this section, we introduce a typical framework for GAN-based image tone mapping.

8.3.1 Image Tone Mapping Problem

After a professional photographer has finished shooting his work, a series of image processing operations are performed in post-processing, which often include brightness, sharpness, saturation, contrast, hue, and even content adjustment operations, which all belong to the modification of the global and local pixel values of the image.

Several examples of adjustments are shown in Fig. 8.6.

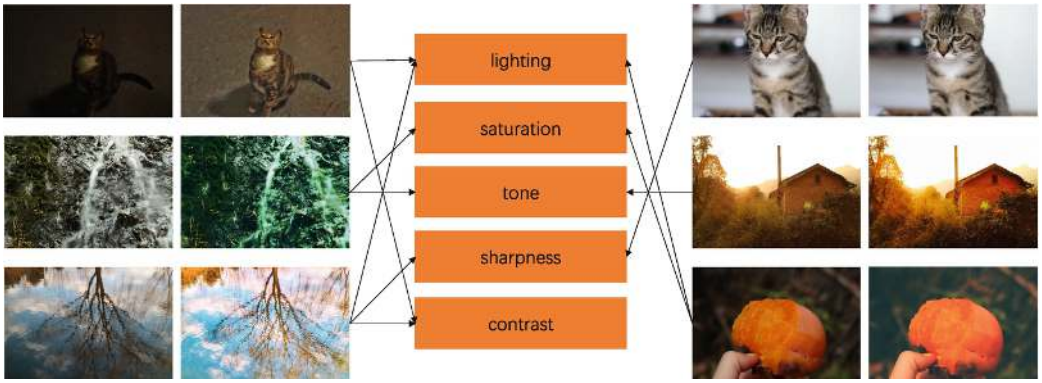


Fig. 8.6 Image enhancement operation example

Four sets of comparison images are shown in Fig. 8.6, where the left side of each set are the original images and the right side are the enhanced images.

There are many software such as Snapseed and Splash Retouch that provide automatic tone mapping of photos. However, since automatic enhancement involves many operations and an understanding of image aesthetics, it has not reached the level of manual post production yet.

Tone mapping is usually a continuous nonlinear mapping operation, and it can be divided into two main categories in general.

The first category is contrast enhancement: the purpose is to enhance the content of interest in the image and suppress the content of disinterest, thus improving the recognition of the image. Generally, due to the surrounding environment and the settings of the hardware of the device itself, the effect of the pictures taken by the camera head is not as good as the direct observation result of the human eye, especially in low light and other backgrounds, the contrast of the images taken by the camera is often low and the visual effect is poor, so the enhancement of this type of image is a common operation.

The top left image in Fig. 8.6 is a typical photo taken in low light. The overall brightness of the image is low and the contrast is also low, so it is necessary to adjustments them. Employing Snapseed software to increase brightness and contrast and then reduce shadows to get the image on the right.

The top right image in Fig. 8.6 shows the effect of using Snapseed software to make high-dynamic range image adjustments, often referred to as the High-Dynamic Range (HDR) effect.

The general display can only represent 8 bits, i.e., $2^8 = 256$ brightness levels, while the human eye can see the range is about 10^5, corresponding to the binary about 2^{16}, that is, 16 bits. HDR technology is to use 8 bits to simulate the information that can be expressed in 16 bits, with higher contrast.

The second category is saturation and tone enhancement: it often refers to adjusting the tonal style of the entire image to create works that further highlight the subject.

In most cases, directly captured images often give people a feeling of being too plain due to low saturation, while photos with high color saturation will display better aesthetic effects, which are shown in the lower left figure of Fig. 8.6.

The overall brightness of the original image was low and the tones were dark and not clean enough, lacking a sense of art. Using Snapseed software to increase the brightness and saturation, the visual effect was greatly enhanced after the adjustment, and the picture had a bright color.

Although most cameras have an automatic white balance function, there are times when we need to adjust the white balance to enhance the visual sense or even to achieve a special expression**.** In Snapseed software, the white balance menu includes two options, color

temperature and coloring. The ends of the color temperature menu are blue and yellow, while the ends of the coloring menu are red (warm colors) and green (cool colors). The bottom right figure in Fig. 8.6 is the addition of warm colors.

8.3.2 Image Tone Mapping Dataset

In order to study the automatic image enhancement problem, relevant datasets need to be created. Some datasets are suitable for static scenes by using different parameter configurations for shooting in the same scene. Some use different devices to shoot at the same time and need to match the viewpoint. Next, we will introduce the two datasets that are commonly used, which are the MIT-Adobe FiveK dataset, and the DPED dataset.

The MIT-Adobe FiveK dataset [8] is the most widely used tone mapping dataset, which was released in 2011 and contains 5000 RAW photos taken by DSLR cameras, each of which has been post-adjusted by five experienced photographers using Adobe Lightroom tools, with adjustments mainly for tone. Since the dataset contains paired data from the original and five post images, and has multiple post retouch images from the same photographer, it can be used for learning a particular post style. In addition, each image is labeled with semantic information, such as indoor and outdoor, day and night, people, nature, and man-made targets, which can be used for training models under different scenes.

The DPED dataset [9] captured by three different mobile phones and a digital camera, and then match and crop the images. The three phones were iPhone 3GS, BlackBerry Passport and Sony Xperia Z, and the camera was a Canon 70D DSLR. 5727 images were taken by the iPhone, 4549 by the Sony and 6015 by the BlackBerry. The dataset covered a variety of common daytime lighting and weather conditions and was collected over a 3-week period, and images in the dataset is obtained all by using automatic shooting mode.

Since four devices are simultaneously capturing images, it is impossible to align the captured images completely in the early stage, so post-processing alignment is required. The authors used the SIFT algorithm to align the images, and the final pairs of images were guaranteed to have no more than five pixels of deviation from each other.

8.3.3 GAN-Based Image Tone Mapping Framework

Among the deep learning tone mapping frameworks, there are models based on various types of filtering parameters learning, and there are models based on pixel regression, and since GAN is good at capturing data distribution, we mainly introduce the models based on pixel regression.

A typical framework is shown in Fig. 8.7.

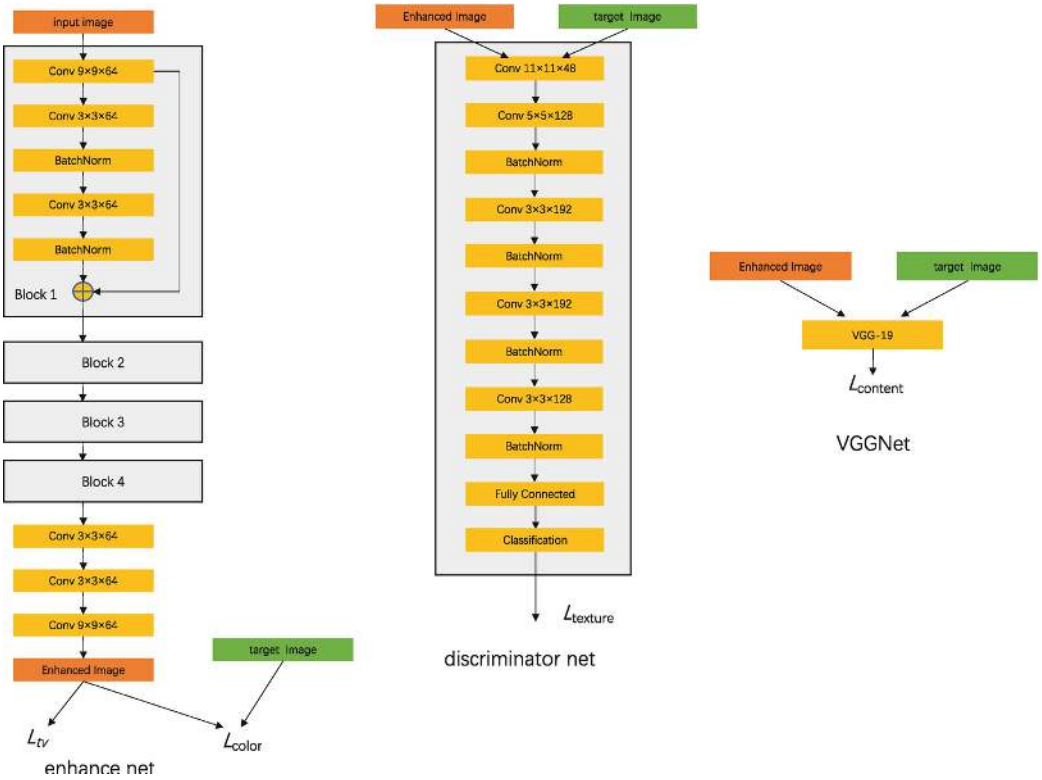


Fig. 8.7 GAN-based supervised image enhancement model

Three networks are included in Fig. 8.7, an Image Enhancement Network (IEN), a Discriminator Network (DN), and a feature retention network VGG-19. The DPED dataset, which contains paired images with low and high quality, is used for training, so the input and target images are one-to-one correspondence.

The image enhancement network can be regarded as the generative network of GAN, where the input is a three-channel image, which first passes through four residual blocks (block1,b2,b3,b4 in the figure), and each residual block has two convolutional layers inside. Then it goes through three convolutional layers, and the last convolutional layer outputs a three-channel image, which is the enhanced image.

The model includes two loss functions, color loss l_{color} and smoothing loss l_{tv}. Of these l_{color} requires the truth image to be computed together with the enhanced image, which is a reconstruction loss that can be computed using the standard Euclidean distance.

In the calculation of color loss l_{color}. Firstly, Gaussian blur was applied to both the target image and the enhanced image. The Gaussian blur can remove part of the edge details and texture, and retain the contrast and color of the overall image, which makes the color smoother locally and also has a certain local translation invariance, which is more conducive to the stable learning of the model than using the target image and the enhanced image directly.

l_{tv} is the standard smoothing loss, which comes from the field of image denoising, and it can achieve a small overall smoothing of the image and effectively remove noise such as pretzels.

The input to the discriminative network D is generated by fusing the target image together with the enhanced image. There are various ways of fusion, the authors use a pixel-by-pixel weighted summation approach and also use methods such as channel stitching. The discriminative network has five convolutional layers, a fully connected layer with a dimension of 1024, and a two-dimensional probability vector output. The target image is used as a

conditional input, so the discriminator has the same principle as CGAN, and the loss function is cross-entropy loss, which is also called texture loss (textures loss).

Specifically, when calculating the texture loss, both the target image and the enhanced image are transformed into grayscale maps, the reason being that the texture information of an image is mainly related to the grayscale spatial distribution, which reduces the learning difficulty of the model and the risk of overfitting.

The pre-trained VGG network is used as a feature preservation network to extract high-level features from both the target image and the enhanced image. and then content loss is computed by using the standard Euclidean distance. The meaning behind content loss, also known as perceptual loss, is that if the target image is very close to the enhanced image, then the features extracted through the VGG network should also be close, which is used to place constraints on the high-level semantic information and is widely used in tasks such as image super-resolution and stylization.

Since the framework in Fig. 8.7 requires paired data for training, the authors later used the CycleGAN model, which was extended to an unsupervised scheme [10], so that it does not have to rely on paired data, and the basic framework is shown in Fig. 8.8.

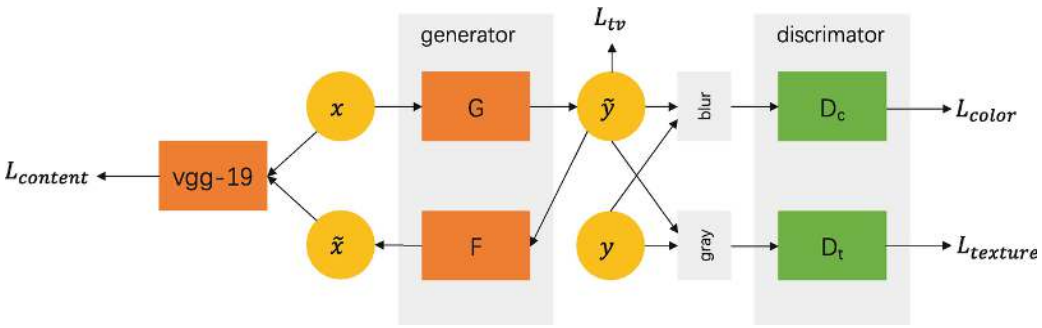


Fig. 8.8 GAN-based unsupervised image enhancement model

Where x is the input low quality image and the boosted image $\tilde{y}$ is obtained after inputting x into a generator G and then the result is fed into a reverse generator to generate the image $\tilde{x}$. The perceptual loss L_{content} is calculated between x and $\tilde{x}$ by using the vgg19 network. y is the input high-quality image, which is not a one-to-one correspondence to x. After performing Gaussian blurring on y and$\tilde{y}$, they are sent to discriminator D_c to get the color loss L_{color} and after grayscale fusion, they are sent to the discriminator D_t to obtain the texture loss L_{texture} and the smoothing loss L_{tv} is calculated by $\tilde{y}$.

The computational details of the losses, the details of the network structure of the generator and the discriminator are the same as the model in Fig. 8.8.

Using unpaired datasets for training can greatly reduce the dependence on data, thus enabling the training of models with more high-quality datasets and improving the generalization ability of the models.

Both of the above schemes rely on limited image datasets, whose cost is still high. In order to use more complex and diverse data, subsequent researchers have proposed the Seeing In the Dark GAN (SIDGAN) framework [11], which uses simulated images to synthesize real scene videos and use them for low-light video enhancement.

SIDGAN obtains a large-scale dataset by converting videos on the Internet into low-light videos, while direct conversions face the problem of domain mismatch, so a long-exposure image is used as an intermediary to synthesize a large dataset of simulation pairs. The overall block diagram of the paper is shown in Fig. 8.9.

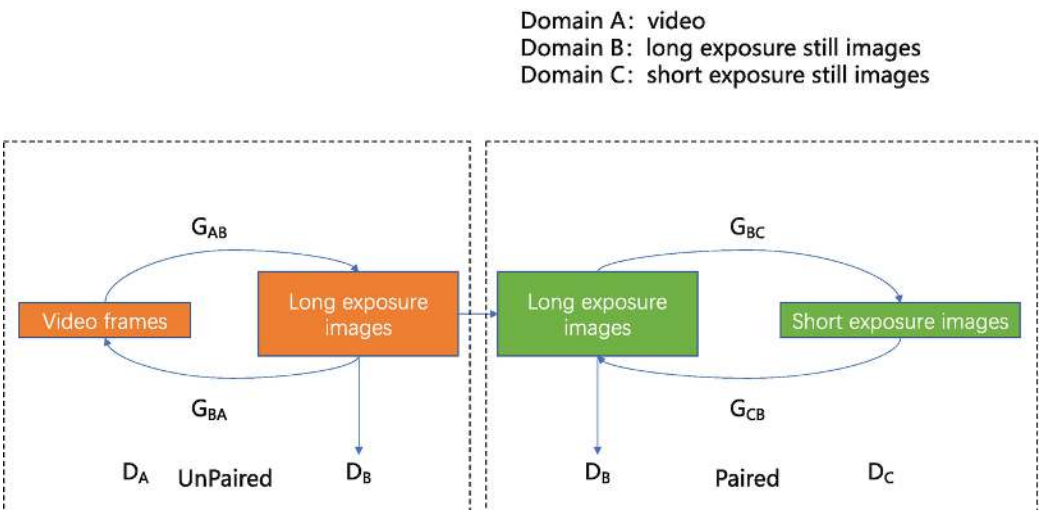


Fig. 8.9 GAN-based unsupervised image enhancement model

It can be seen that two CycleGAN-like models are used for training, the conversion from the normal Video A set to the long-exposure image B set, and the conversion from the long-exposure image B set to the short-exposure image C set, respectively. Where A to B is unpaired, while B to C conversion is paired. The A-to-B transformation includes two adversarial losses, cyclic consistency loss and identity preservation loss, both of which are generic losses in CycleGAN.

All the generators use the UNet structure and all the discriminators use the PatchGAN structure. The training is divided into three steps, which are analyzed as follows:

1. First train on a real static video (i.e., no moving target) dataset to learn the correct distribution of colors and illumination.
2. Then fine-tune on the simulated dynamic video data and learn timing consistency.
3. Finally, fine-tune on the stationary data.

Although current researchers have proposed a large number of tone mapping models, none of them has yet been able to enhance all kinds of images more perfectly, and although many mainstream APPs are equipped with automatic enhancement functions, they are still far from the level of post-retouching by photographers. The method based on deep learning models shows good algorithmic potential, and readers who are interested in it can continue to follow up.

8.4 Image Super-Resolution

People's pursuit of resolution is never-ending, the higher the resolution can get clearer imaging effect and have better aesthetic feeling. At the same time, restoration of many old photos and videos with low resolution has great human and social value, and GAN has been quite successful in the field of super resolution.

In this section, we introduce the typical framework of GAN-based image super-resolution.

8.4.1 Image Super-Resolution Problem

We often refer to the image resolution as the product of the number of pixels on the long side of the image and the number of pixels on the short side of the image, for example, the Canon EOS M3 has a maximum resolution of 6000 × 4000, with 6000 pixels in a row and 24 million pixels for the entire image.

The higher the resolution, the clearer the image obtained, at the same time, the higher resolution also means more storage space, for the very limited space of mobile devices, it needs to consider the balance of resolution and storage space.

Image super-resolution, which proposed to recover from a low-resolution image to a high-resolution image, is widely used in daily image and video storage and viewing.

The 320 × 240 resolution images were mainstream in mobile phones 10 years ago, and their visual beauty was incomparable compared to the 4K resolution that is readily available today. We can use image super-resolution techniques to recover low-resolution images taken back then, a typical example of which is shown in Fig. 8.10.

Fig. 8.10 Old photos with super-resolution, the left image is the original image, the right image is the adjusted image

Today, image super-resolution is used in the restoration of many valuable historical photographs and videos, which have great humanistic and commemorative value.

The simulation of image super-resolution datasets is relatively simple, and researchers can often create relevant datasets by downsampling high-resolution images. Since there is no strict requirement for the type of images, smaller datasets such as BSD68, BSD100, larger datasets such as ImageNet, and domain-specific datasets such as the face attribute dataset CelebA have been used by researchers.

8.4.2 GAN-Based Image Super-Resolution Framework

Next, we introduce several common super-resolution frameworks for images, including GAN-based frameworks, as well as typical unsupervised GAN frameworks.

8.4.2.1 Basic Framework

With the development of generative adversarial networks GAN, the adversarial learning mechanism of generators and discriminators has shown a powerful learning capability in image generation tasks. Researchers at Twitter have proposed the SRGAN [12] model using ResNet as the generator structure and VGG as the discriminator structure, and the model structure is schematically shown in Fig. 8.11.

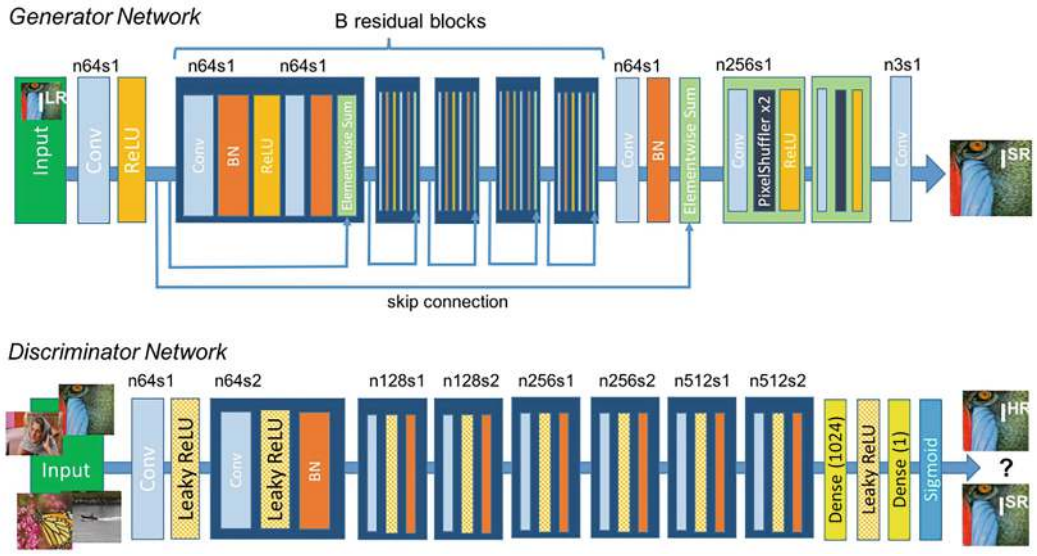


Fig. 8.11 SRGAN model structure

The generator structure in Fig. 8.11 contains several residual modules that do not change the feature resolution and several post-upsampling modules based on sub-pixel convolution. The discriminator structure, on the other hand, contains several convolutional layers with increasing number of channels, and each time the number of feature channels is doubled, the feature resolution is reduced to half of the original one.

The SRGAN model constructs a content loss function based on VGG network features instead of the previous MSE loss function and achieves better reconstruction results in visual perception by using adversarial learning of generators and discriminators.

Subsequent researchers proposed an enhanced version of SRGAN, namely ESRGAN [13], which is based on SRGAN by optimizing the structure of the generator and the loss function, and achieved better image super-resolution results compared to SRGAN.

8.4.2.2 Unsupervised Model

Although researchers have proposed dozens of image super-resolution models, most of the current supervised image super-resolution models are often not effective for real image super-resolution, mainly because most of the models use simulated data. Researchers need to use image algorithms to sample high-resolution maps to obtain low-resolution maps for imitating the real image degradation process, but the real image degradation is not only the resolution reduction, but also the introduction of various types of image noise and defects in the process, which is very difficult to simulate by basic image processing algorithms, so the models trained based on sampling are prone to overfitting and have poor generalization ability.

Therefore, some authors have proposed to let GAN first learn the image degradation process to obtain low-resolution images, and then train the model based on the obtained dataset consisting of paired high-resolution and low-resolution images. It is an unsupervised learning process in which a series of frameworks are represented. We select one for introduction [14], which is shown in Fig. 8.12 Loss.

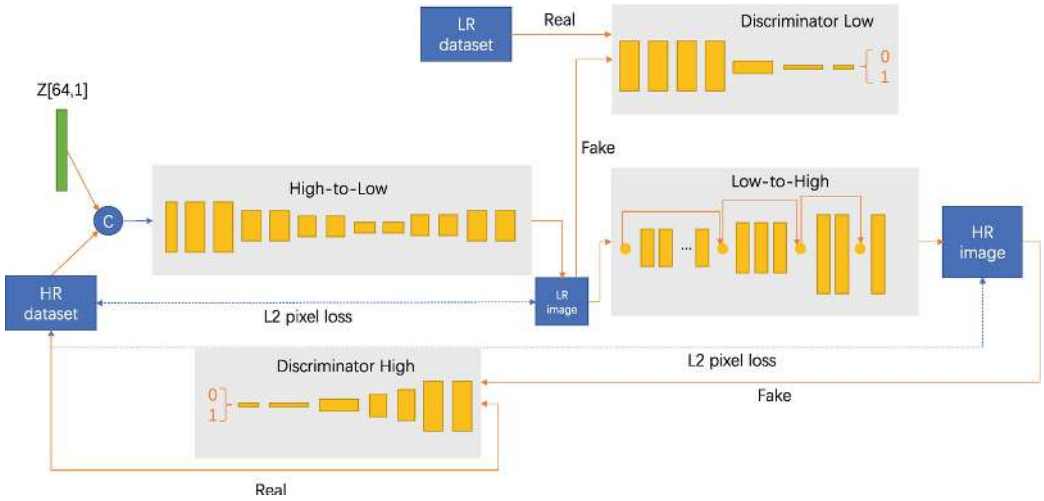


Fig. 8.12 High-to-Low GAN and Low-to-High GAN models

The whole process consists of a High-to-Low GAN and a Low-to-High GAN, and we describe the two models in detail as follows:

High-to-low GAN model: This model serves to generate low-resolution images from high-resolution datasets. The high-resolution image datasets can be Celeb-A, AFLW, LS3D-W, and VGGFace2 with high face quality, etc., and the low-resolution image datasets can be Wider face with low face quality, etc., which form the unpaired high-resolution-low-resolution dataset. The downsampling network (High-to-low) in High-to-low GAN is a encoding and decoding structure, whose input is stitched from random noise z and high-resolution map. Then the input is used to generate low-resolution map.

Low-to-High GAN model: From the output results of High-to-low GAN, paired low-resolution and high-resolution training data can be obtained, thus enabling the training of a normal super-resolution network, i.e., the Low-to-High GAN model, which is a structure based on skip-connection.

The two frameworks introduced above are typical supervised and unsupervised frameworks, which use network pairs to reconstruct images from low-resolution maps and cannot completely avoid the blurring problem of reconstructed high-resolution images, which is difficult to solve by adjusting the network structure, making the reconstruction results unrealistic and lacking in details.

PULSE (Photo Upsampling via Latent Space Exploration) [15] is another self-supervised super-resolution framework based on generative models, which opens up a new way of image super-resolution. Instead of upsampling the low-resolution image step by step to add details, it samples in the StyleGAN latent space to obtain multiple high-resolution images and then downsamples them, which can achieve 64-fold super-resolution, and the schematic diagram of the framework is shown in Fig. 8.13.

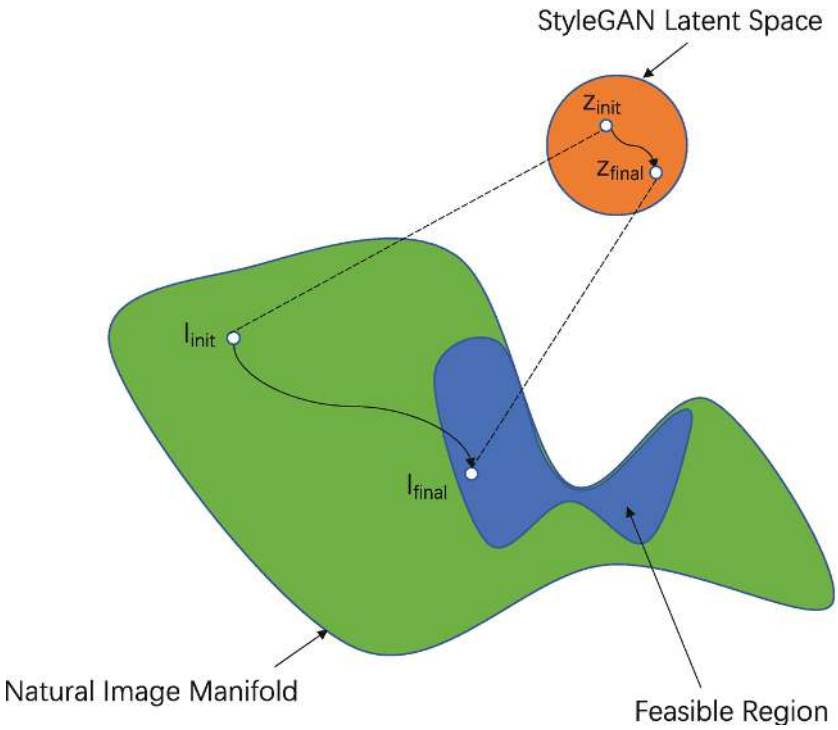


Fig. 8.13 Schematic diagram of the PULSE framework

In Fig. 8.13, I_{init} represents the initial high-resolution image, and I_{final} represents the final high-resolution result map. Z_{init}, Z_{final} represent the initial latent vector and the final latent vector, respectively.

PULSE obtains both realistic and high-definition reconstructions by traversing the generated high-resolution images and comparing the low-resolution images corresponding to these high-resolution images with the original image of the input map, where the closest is the solution, with the goal of making the solution region fall in the space of the natural image.

8.5 Image Restoration

Image restoration, i.e., the removal of unwanted defective areas in images, is a very critical function, and it is currently undergoing rapid development. GAN has become one of the indispensable key technologies for solving this problem due to its powerful generation ability.

In this section, we introduce a typical framework for GAN-based image restoration.

8.5.1 Image Restoration Basics

In the early stage of photography, it often happens that we cannot control the shooting scene, for example, the busy crowd in scenic spots makes it difficult to get photos with clean backgrounds. On the other hand, images may also be contaminated after being spread through the medium many times, resulting in damaged areas, and Fig. 8.14 shows some images that need to be repaired.

Fig. 8.14 Picture to be repaired

The Repair Brush tool in Photoshop software is a tool that allows for localized image repair. The technical principle behind it is PatchMatch, which is a method based on image block filling that allows for gradual repair by using an interactive strategy.

Figure 8.15 shows the results of using the Photoshop Repair Brush tool to repair the image in Fig. 8.14.

Fig. 8.15 Photoshop's repair results for the image in Fig. 8.14

Traditional image restoration methods are based on the principle of image self-similarity by finding matching blocks with similar textures in the current image and then using Poisson fusion and other methods to complement, this class of methods are represented by structure propagation, which has been able to better complement smaller areas. However, the problem of this type of method is that it only takes into account the similarity of the image, without considering the semantic information, and it can remove the defects better for the simple texture and the position far from the image subject. However, for complex textures, which are similar to the background and adhering to the image subject, the completed image is often very unrealistic and cannot be completed for larger missing areas, such as the "cat tail" in the second image and the "street light pole" in the third image.

Among the traditional image restoration methods, there is another class of models based on image decomposition and sparse representation, which learn the dictionaries of the optimal sparse representation from the structure (Cartoon) and texture (Texture) of the image separately and then perform image restoration, but it is still difficult to achieve better results for real images.

8.5.2 GAN-Based Image Restoration Framework

The traditional restoration method usually uses similarity algorithms to select image blocks from other regions of the image for completion, while GAN itself has powerful image generation capabilities. It uses a context encoder [16] to infer information about the occluded part from the unoccluded part of the occluded image, and the specific network structure is shown in Fig. 8.16.

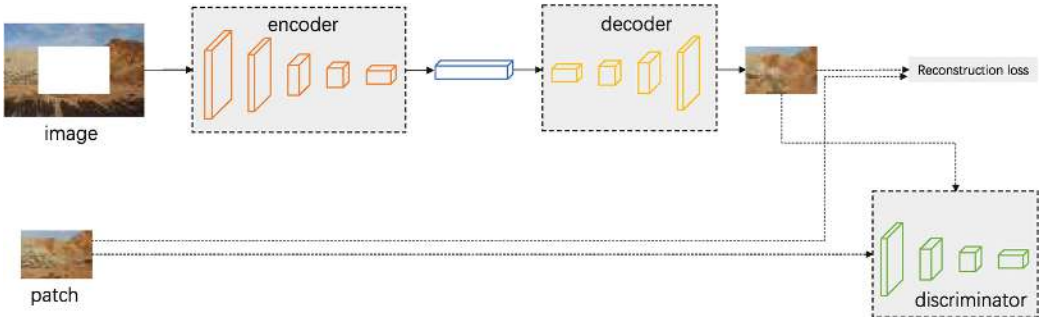


Fig. 8.16 Context encoder

Context Encoders contain an encoder, a fully connected layer, and a decoder for learning image features and generating prediction maps corresponding to the regions of the image to be repaired, with the input being the original image including the occluded regions and the output being the prediction results for the occluded regions.

The main structure of the encoder is AlexNet network, if the input size is 227 × 227, the size of feature map is 6 × 6 × 256.

The encoder is followed by a channel-by-channel fully connected layer. In order to obtain a large sensory field with a small computational size, the authors designed a channel-by-channel fully connected structure, which has an input size of 6 × 6 × 256 and an output size that does not change.

Of course, it is not necessary to use a channel-by-channel fully connected layer structure here, but only to control the features with a large receptive field. When the perceptual field is small, the effective information from outside the region will not be available to the interior points of the complemented region, and the effect of complementation will be greatly affected.

The dashed line in Fig. 8.17 indicates the region to be repaired, and the blue box indicates the receptive field. The receptive field of point p_1 contains both the region to be repaired and the region that does not need to be repaired and has some deterministic information. In contrast, the receptive field of point p_2 has only the region to be repaired, which is difficult to obtain useful information for learning.

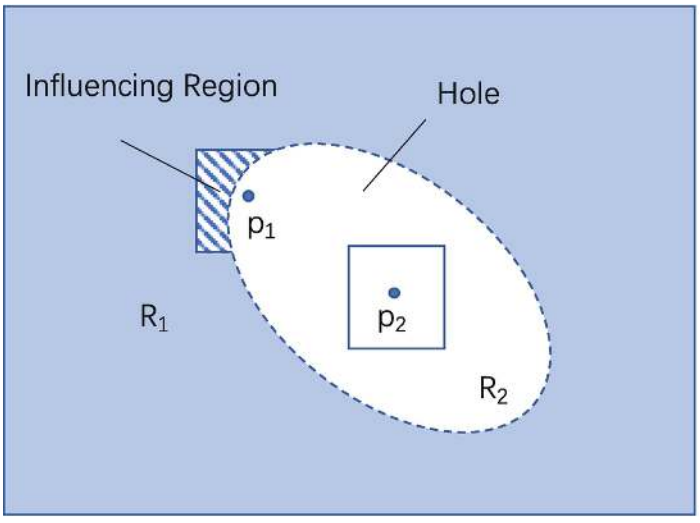


Fig. 8.17 Schematic diagram of the repair area

The decoder contains a number of upsampling convolutions that output the part to be repaired. The specific upsampling ratio is related to the size of the repaired part which is relative to the original image.

The loss function during the training of the network is composed of two parts. The first part is the image reconstruction loss of the encoder-decoder part, which uses the L2 distance between the predicted part and the original image. Only the part that needs to be repaired is calculated, so it needs to be under the control of the mask. The second part is the adversarial loss of the GAN. The network model parameters are considered to be optimal when the discriminator of the GAN is unable to determine whether the prediction image is from the training set or not.

Context Encoder is the first GAN-based image complementation network, which can achieve the filling of larger holes. However, its generator and discriminator structures are relatively simple, and although the complementation results are relatively realistic, they have very unsmooth boundaries and do not satisfy local consistency.

In allusion to this character, the researchers jointly used global and local discriminators to improve the Context Encoder model and proposed a locally and globally consistent framework (i.e., Globally and Locally Consistent Image Completion, or GLCIC) [17], which is shown in Fig. 8.18. It contains three modules, one is the image complementation model, one is the global discriminator, and one is the local discriminator. The global discriminator can be used to judge the consistency of the whole image reconstruction, the local discriminator can be used to judge whether the filled image blocks have good local details, and the specific discriminative loss is to concatenate the global discriminator and the local discriminator output feature vectors, and then to discriminate the authenticity after sigmoid mapping.

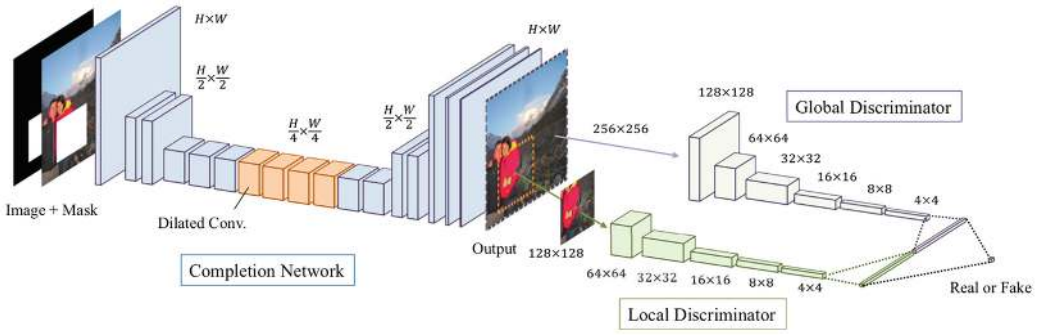


Fig. 8.18 Schematic diagram of GLCIC framework

Context Encoder and GLCIC framework, both use simulated data for model training, i.e., by filling the image with regular blank image blocks to imitate defects, but real images include many degradation types, such as complex noise and irregular scratches, which are difficult to simulate by simple degradation strategies, and different types of degradation require different processing methods. For example, unstructured defects, such as grain, fading, require statistical features of neighboring pixels. While for structured defects, such as scratches, require global contextual information. How to model the real type of degradation by simulation is the key to repair these images.

As the same with the previously introduced problems of image noise reduction, enhancement, deblurring, and super-resolution, unsupervised solutions are needed to better deal with complex and realistic image restoration problems, for which a researcher has proposed a technical framework (i.e., Bringing Old Photos Back to Life [18]) that treats the image restoration problem as a transformation problem between three domains.

A schematic diagram of the unsupervised image restoration framework is shown in Fig. 8.19.

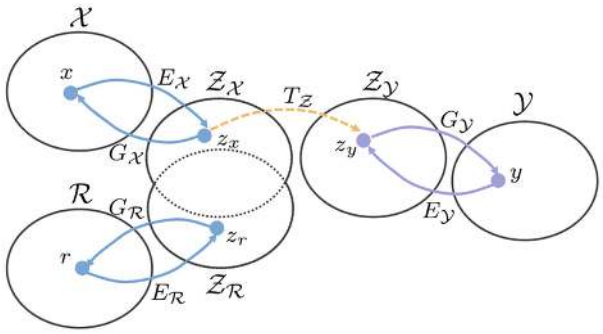


Fig. 8.19 Schematic diagram of the unsupervised image restoration framework

Where *x* and *y* are paired images, *x* is the simulated degradation image, *y* is the original image of high quality, and *r* is the real degradation image. The simulated image and the real image are first mapped to the same latent space *z* using a VAE, and the real image is mapped to that space using another VAE. The goal of learning is to align *x* and *r* in the same *Z* domain so that $z_r = z_x$ can be achieved, and then a transformation T_z is performed to complete the transformation from z_x to z_y to achieve the learning from the real degenerate image *r* to the high-quality original image *y*. The process is expressed in mathematical equation in Eq. (8.1):

$$r_{R\to y} = G_y \circ T_Z \circ E_R(r) \tag{8.1}$$

In this way, learning can be done using paired simulation data to complete the conversion from low-quality to high-quality images in the latent space, allowing for better generalization over real images. The core problem of the whole framework is how to make the *X* and *R* domains can be mapped to the same space *Z*.

For *r* and *x*, which share a VAE_1, and *Z* using a Gaussian prior distribution constraint, the optimization objective for *r* is as follows:

$$L_{\text{VAE}_1}(r) = KL\left(E_{R,x}(z_r|r)\middle\|N(0,I)\right) + \alpha E_{z_r\sim E_{R,x}((z_r|r))}\left[\left\|G_{R,x}\left(r_{R\to R|z_r} - r\right\|_1 + E_{\text{VAE}_1,\text{GAN}}(r)\right.\right. \tag{8.2}$$

The first KL scatter is to constrain the distribution of *Z*. The second is the reconstruction loss, and the third is the LSGAN loss, which is used to alleviate the oversmoothing problem in GAN. The optimization objective of *x* $L_{\text{VAE}_1}(x)$ is similar and it will not be reiterated here.

Meanwhile, a discriminator is used to train Z_r and Z_x against each other, which is defined as follows:

$$L_{\text{VAE}_1,\text{GAN}}(r,x) = E_{x\sim\chi}\left[D_{R,\chi}(E_{R,\chi}(x))^2\right] + E_{r\sim R}\left[\left(1 - D_{R,\chi}(E_{R,\chi}(r))^2\right]\right. \tag{8.3}$$

The optimization objectives of the entire VAE are as follows:

$$\min_{E_{R,\chi},G_{R,\chi}} \max_{D_{R,\chi}} L_{\text{VAE}_1}(r) + L_{\text{VAE}_1}(x) + L_{\text{VAE}_1,\text{GAN}}(r,x) \tag{8.4}$$

After training the VAE, it is necessary to learn the mapping from *x* to *y*. At this time, fix the two VAEs and only train the transformation network *T* with the following optimization objectives:

$$L(x,y) = \lambda_1 L_{T,l_1} + L_{T,GAN} + \lambda_2 L_{FM} \tag{8.5}$$

where L_{T,l_1} is the distance in Z-space, and $L_{T,\text{GAN}}$ is the loss form of LSGAN, and L_{FM} is the feature matching loss, which is used to ensure the stability of the model training, and is actually the feature space distance of the VGG network, which is not reiterated here because it has been introduced many times before.

8.6 Face Super-Resolution Reconstruction Based on SRGAN

In the previous subsections, we introduced the typical technical frameworks of GAN in image noise reduction, deblurring, tone mapping, image super-resolution, image restoration, and other problems, and they have obtained commercialization achievements. In this section, we implement the complete process of training and testing of super-resolution models based on Pytorch.

8.6.1 Project Interpretation

Let's first introduce the dataset, benchmark model and interpret the entire project code, including the dataset interface, the network structure. and the definition of the optimization goals.

8.6.1.1 Dataset and Benchmark Model

Most datasets for super-resolution reconstruction tasks are obtained by sampling from high-resolution images, and we adopt a same scheme here as well. The dataset can be either a large dataset containing millions of images like ImageNet or a small dataset with a rich enough pattern, and after a trade-off we choose a high-definition face dataset, CelebA-HQ [19]. The CelebA-HQ dataset is released in 2019 and contains 30,000 high-definition face images including different attributes, where the image sizes are all 1024 × 1024.

We choose SRGAN, the first model that uses GAN for the super-resolution reconstruction task, as the benchmark model.

8.6.1.2 Dataset Interface

As we said earlier in the introduction of image super-resolution datasets, image super-resolution datasets are often sampled from high-resolution maps to obtain low-resolution maps, which are then composed into image pairs for training purposes, and the following is the core code for processing the data in the training and validation sets:

```
from os import listdir
from os.path import join
import numpy as np
from PIL import Image
from torch.utils.data.dataset import Dataset
from torchvision.transforms import Compose, RandomCrop, ToTensor,
ToPILImage, CenterCrop, Resize
import imgaug.augmenters as iaa
aug = iaa.JpegCompression(compression=(0, 50))

## Adjust the crop size to an integer multiple of upscale_factor
based on the upsampling factor
def calculate_valid_crop_size(crop_size, upscale_factor).
return crop_size - (crop_size % upscale_factor)
```

```
## Training set high resolution image preprocessing function
def train_hr_transform(crop_size).
return Compose([
RandomCrop(crop_size).
ToTensor().
])

## Preprocessing function for low-resolution images of training set
def train_lr_transform(crop_size, upscale_factor).
return Compose([
ToPILImage().
Resize(crop_size // upscale_factor, interpolation=Image.BICUBIC).
ToTensor()
])
## Training dataset class
class TrainDatasetFromFolder(Dataset).
def __init__(self, dataset_dir, crop_size, upscale_factor).
super(TrainDatasetFromFolder, self). __init__()
self.image_filenames = [join(dataset_dir, x) for x in
listdir(dataset_dir) if is_image_file(x)] ##Get all images
crop_size = calculate_valid_crop_size(crop_size, upscale_factor)##
Get the crop size
self.hr_transform = train_hr_transform(crop_size) ## high resolution
image preprocessing function
self.lr_transform = train_lr_transform(crop_size, upscale_factor) ##
low resolution image preprocessing function
## dataset iteration pointer
def __getitem__(self, index).
hr_image =
self.hr_transform(Image.open(self.image_filenames[index]))
##Randomly crop to get a high resolution image
lr_image = self.lr_transform(hr_image) ##Get the low-resolution
image
return lr_image, hr_image

def __len__(self).
return len(self.image_filenames)

## Validate dataset classes
class ValDatasetFromFolder(Dataset).
def __init__(self, dataset_dir, upscale_factor).
super(ValDatasetFromFolder, self). __init__()
self.upscale_factor = upscale_factor
self.image_filenames = [join(dataset_dir, x) for x in
listdir(dataset_dir) if is_image_file(x)]

def __getitem__(self, index).
hr_image = Image.open(self.image_filenames[index])

## get crop size
```

```
w, h = hr_image.size
crop_size = calculate_valid_crop_size(min(w, h),
self.upscale_factor)
lr_scale = Resize(crop_size // self.upscale_factor,
interpolation=Image.BICUBIC)
hr_scale = Resize(crop_size, interpolation=Image.)
hr_image = CenterCrop(crop_size)(hr_image) ## center crop to get
high resolution image
lr_image = lr_scale(hr_image) ##Get the low-resolution image
return ToTensor()(lr_image), ToTensor()(hr_image)

def __len__(self).
return len(self.image_filenames)
```

From the above code, we can see that two preprocessor interfaces are included, train_hr_transform and train_lr_transform. train_hr_transform contains operations mainly for random cropping, while train_lr_transform contains operations mainly for scaling.

There is also a function named calculate_valid_crop_size, which is used to adjust the crop_size for the training set when the configured image size crop_size does not divide the upscale_factor by an integer. We should avoid when using it, i.e., configure crop_size so that it is equal to an integer multiple of upscale_factor. For the validation set, the narrow edge of the image min(w, h) is used for the initialization of crop_size, so this function serves to adjust crop_size when the narrow edge of the image does not divide the upscale_factor.

The training set class TrainDatasetFromFolder contains several operations. It uses train_hr_transform to randomly crop an image of square size from the original image to the crop size, and uses train_lr_transform to obtain the corresponding low-resolution map. The validation set class ValDatasetFromFolder, on the other hand, crops the image centrally according to the adjusted crop_size and then uses train_lr_transform to obtain the corresponding low-resolution map.

Here we use random cropping and JPEG noise compression as data augmentation operations during training. JPEG noise is added using the imgaug library, whose project address is https://github.com/aleju/imgaug, Fig. 8.20 shows the images with different magnitudes of JPEG noise added to some samples.

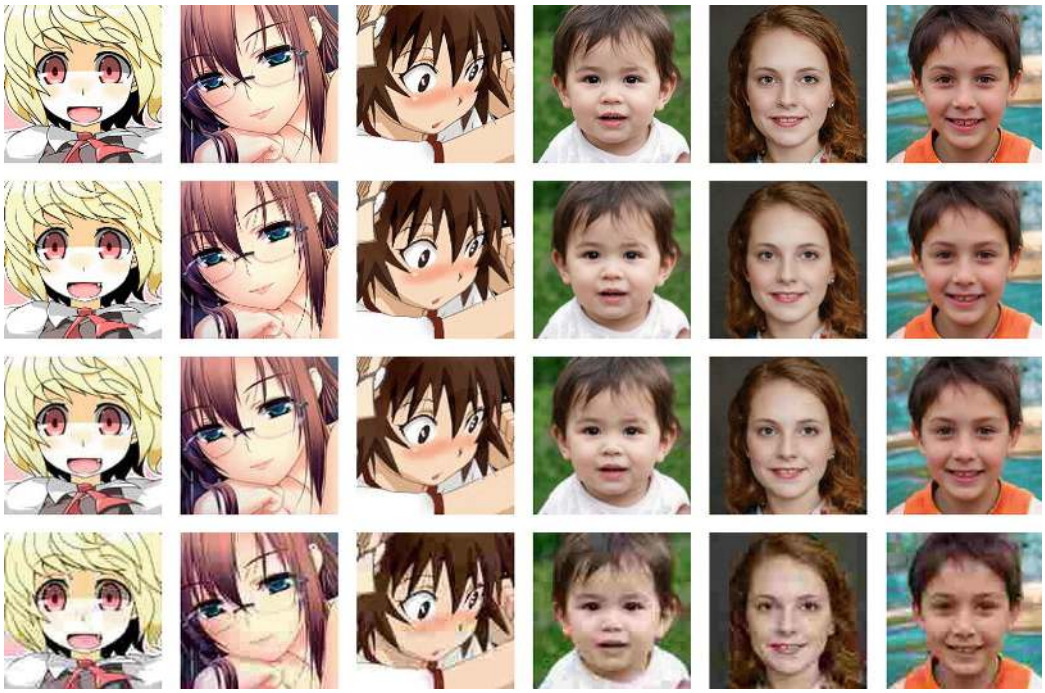


Fig. 8.20 JPEG noise sample image

The first row in Fig. 8.20 shows the original image with a resolution of 512 × 512 size, the second row shows the image scaled to 128 × 128 size without adding JPEG compression noise, and the third and fourth rows show the images scaled to 128 × 128 size and with the imgaug library adding JPEG compression noise of 30% and 90% amplitude, respectively. It can be seen that JPEG noise has a significant impact on the image quality, especially when the noise amplitude is large, and the patchy effect is very obvious. We will compare the results of the model with adding JPEG noise of different amplitudes compared with those without adding JPEG noise later to verify that for real image super-resolution tasks, data augmentation operations that are closer to the real degradation process are necessary.

8.6.1.3 Generator

The generator is an upsampling model based on the residual module, which is defined to include the residual module, the upsampling module, and the backbone model as follows:

```
## Residuals Module
class ResidualBlock(nn.Module).
def __init__(self, channels).
super(ResidualBlock, self). __init__()
## Two convolutional layers, convolutional kernel size is 3×3, the
number of channels remains the same
self.conv1 = nn.Conv2d(channels, channels, kernel_size=3, padding=1)
self.bn1 = nn.BatchNorm2d(channels)
self.prelu = nn.PReLU()
self.conv2 = nn.Conv2d(channels, channels, kernel_size=3, padding=1)
self.bn2 = nn.BatchNorm2d(channels)

def forward(self, x).
residual = self.conv1(x)
```

```
residual = self.bn1(residual)
residual = self.prelu(residual)
residual = self.conv2(residual)
residual = self.bn2(residual)
return x + residual

## upsampling module with a recovery resolution of 2 per
class UpsampleBLock(nn.Module).
def __init__(self, in_channels, up_scale).
super(UpsampleBLock, self). __init__()
## Convolution layer, input channels are in_channels, output
channels are in_channels * up_scale ** 2
self.conv = nn.Conv2d(in_channels, in_channels * up_scale ** 2,
kernel_size=3, padding=1)
## PixelShuffle upsampling layer from the post upsampling structure
self.pixel_shuffle = nn.PixelShuffle(up_scale)
self.prelu = nn.PReLU()

def forward(self, x).
x = self.conv(x)
x = self.pixel_shuffle(x)
x = self.prelu(x)
return x

## Generate Model
class Generator(nn.Module).
def __init__(self, scale_factor).
upsample_block_num = int(math.log(scale_factor, 2))

super(Generator, self). __init__()
## First convolutional layer with 9×9 convolutional kernel size, 3
input channels and 64 output channels
self.block1 = nn.Sequential(
nn.Conv2d(3, 64, kernel_size=9, padding=4).
nn.PReLU()
)
## 6 residual modules
self.block2 = ResidualBlock(64)
self.block3 = ResidualBlock(64)
self.block4 = ResidualBlock(64)
self.block5 = ResidualBlock(64)
self.block6 = ResidualBlock(64)
self.block7 = nn.Sequential(
nn.Conv2d(64, 64, kernel_size=3, padding=1).
nn.BatchNorm2d(64)
)
## upsample_block_num upsampling modules, each upsampling module
restores 2 times the upsampling multiplier
block8 = [UpsampleBLock(64, 2) for _ in range(upsample_block_num)]
```

```
## The last convolutional layer, with a convolutional kernel size of
9×9, 64 input channels and 3 output channels
block8.append(nn.Conv2d(64, 3, kernel_size=9, padding=4))
self.block8 = nn.Sequential(*block8)

def forward(self, x).
block1 = self.block1(x)
block2 = self.block2(block1)
block3 = self.block3(block2)
block4 = self.block4(block3)
block5 = self.block5(block4)
block6 = self.block6(block5)
block7 = self.block7(block6)
block8 = self.block8(block1 + block7)
return (torch.tanh(block8) + 1) / 2
```

In the above generator definition, the nn.PixelShuffle module is called to implement upsampling, which is based on the specific principle of sub-pixel convolutional post upsampling ESPCN model [20], and the flow schematic is shown in Fig. 8.21.

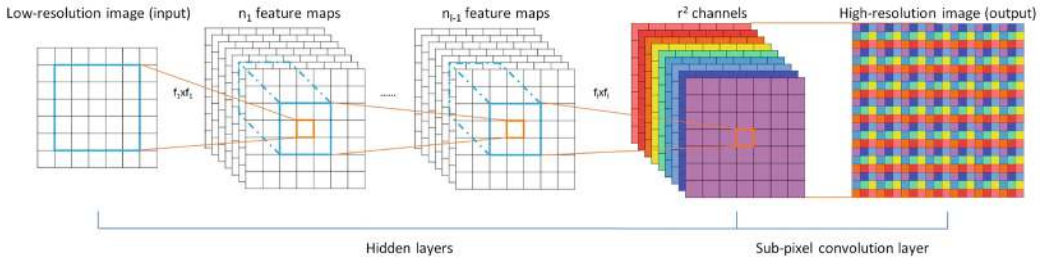


Fig. 8.21 Post upsampling ESPCN model based on sub-pixel convolution

For an image with dimension $H \times W \times C$, the standard deconvolution operation outputs a feature map with dimension $rH \times rW \times C$, where r is the number of times to be enlarged, while as can be seen from Fig. 8.21, the output feature map dimension of the sub-pixel convolution layer is $H \times W \times C \times r^2$, i.e., the feature map keeps the same dimension as the input image, but the number of channels is expanded to the original r^2 times, and then rearranged to obtain the high-resolution results.

The whole process can use a smaller convolution kernel to obtain a larger perceptual field due to the input of smaller image size, which allows the information of neighboring pixel points in the input image to be used effectively and also avoids the increase in computational complexity and is an idea to convert the spatial upsampling problem into a channel upsampling problem, which is adopted as an upsampling module by most of the mainstream image super-resolution models.

8.6.1.4 Discriminator

The discriminator is a general VGG-like CNN model, whose complete definition is as follows:

```
## Residuals Module
class Discriminator(nn.Module).
def __init__(self).
super(Discriminator, self). __init__()
```

```
self.net = nn.Sequential(
## 1st convolutional layer with 3×3 convolutional kernel size, 3
input channels and 64 output channels
nn.Conv2d(3, 64, kernel_size=3, padding=1).
nn.LeakyReLU(0.2).
## 2nd convolutional layer with 3×3 convolutional kernel size, 64
input channels and 64 output channels
nn.Conv2d(64, 64, kernel_size=3, stride=2, padding=1).
nn.BatchNorm2d(64).
nn.LeakyReLU(0.2).
## 3rd convolutional layer with 3×3 convolutional kernel size, 64
input channels and 128 output channels
nn.Conv2d(64, 128, kernel_size=3, padding=1).
nn.BatchNorm2d(128).
nn.LeakyReLU(0.2).
## 4th convolutional layer with 3×3 convolutional kernel size, 128
input channels and 128 output channels
nn.Conv2d(128, 128, kernel_size=3, stride=2, padding=1).
nn.BatchNorm2d(128).
nn.LeakyReLU(0.2).
## 5th convolutional layer with 3×3 convolutional kernel size, 128
input channels and 256 output channels
nn.Conv2d(128, 256, kernel_size=3, padding=1).
nn.BatchNorm2d(256).
nn.LeakyReLU(0.2).
## 6th convolutional layer with 3×3 convolutional kernel size, 256
input channels and 256 output channels
nn.Conv2d(256, 256, kernel_size=3, stride=2, padding=1).
nn.BatchNorm2d(256).
nn.LeakyReLU(0.2).
## 7th convolutional layer with 3×3 convolutional kernel size, 256
input channels and 512 output channels
nn.Conv2d(256, 512, kernel_size=3, padding=1).
nn.BatchNorm2d(512).
nn.LeakyReLU(0.2).
## 8th convolutional layer with 3×3 convolutional kernel size, 512
input channels and 512 output channels
nn.Conv2d(512, 512, kernel_size=3, stride=2, padding=1).
nn.BatchNorm2d(512).
nn.LeakyReLU(0.2).
## Global pooling layer
nn.AdaptiveAvgPool2d(1).
## Two fully connected layers, implemented using convolution
nn.Conv2d(512, 1024, kernel_size=1).
nn.LeakyReLU(0.2).
nn.Conv2d(1024, 1, kernel_size=1)
)

def forward(self, x).
batch_size = x.size(0)
```

```
return torch.sigmoid(self.net(x).view(batch_size))
```

8.6.1.5 Definition of Loss

Next we take a look at the definition of loss, mainly generator loss, which contains a total of four components, namely adversarial network loss, pixel-by-pixel image MSE loss, perceptual loss based on the VGG model, and TV smoothing loss for constrained image smoothing.

```
## Generator loss definition
class GeneratorLoss(nn.Module).
def __init__(self).
super(GeneratorLoss, self). __init__()
vgg = vgg16(pretrained=True)
loss_network = nn.Sequential(*list(vgg.features)[:31]).eval()
for param in loss_network.parameters().
param.requires_grad = False
self.loss_network = loss_network
self.mse_loss = nn.MSELoss() ##MSE loss
self.tv_loss = TVLoss() ##TV smoothing loss

def forward(self, out_labels, out_images, target_images).
# Against loss
adversarial_loss = torch.mean(1 - out_labels)
# Perceived loss
perception_loss = self.mse_loss(self.loss_network(out_images),
self.loss_network(target_images))
# Image MSE loss
image_loss = self.mse_loss(out_images, target_images)
# TV smoothing loss
tv_loss = self.tv_loss(out_images)
return image_loss + 0.001 * adversarial_loss + 0.006 *
perception_loss + 2e-8 * tv_loss

## TV smoothing loss
class TVLoss(nn.Module).
def __init__(self, tv_loss_weight=1).
super(TVLoss, self). __init__()
self.tv_loss_weight = tv_loss_weight

def forward(self, x).
batch_size = x.size()[0]
h_x = x.size()[2]
w_x = x.size()[3]
count_h = self.tensor_size(x[:, :, 1:, :])
count_w = self.tensor_size(x[:, :, :, 1:])
h_tv = torch.pow((x[:, :, 1:, :] - x[:, :, :h_x - 1, :]), 2).sum()
w_tv = torch.pow((x[:, :, :, :, 1:] - x[:, :, :, :w_x - 1]),
2).sum()
return self.tv_loss_weight * 2 * (h_tv / count_h + w_tv / count_w) /
batch_size
```

```
@staticmethod
def tensor_size(t).
return t.size()[1] * t.size()[2] * t.size()[3]
```

8.6.2 Model Training

Next, we interpret the core training code of the model and check the results of the model training.

8.6.2.1 Model Training

In addition to the model and loss definition, the training code also needs to complete the optimizer definition, training and validation indicator variables storage, the core code is as follows.

```
## Parameter Interpreter
parser = argparse.ArgumentParser(description='Train Super Resolution
Models')
## crop size, i.e. training scale
parser.add_argument('--crop_size', default=240, type=int,
help='training images crop size')
## upscale factor
parser.add_argument('--upscale_factor', default=4, type=int,
choices=[2, 4, 8].
help='super resolution upscale factor')
## Number of iteration rounds
parser.add_argument('--num_epochs', default=100, type=int,
help='train epoch number')

## Training Master Code
if __name__ == '__main__'.
opt = parser.parse_args()
CROP_SIZE = opt.crop_size
UPSCALE_FACTOR = opt.upscale_factor
NUM_EPOCHS = opt.num_epochs

## Get training set/validation set
train_set = TrainDatasetFromFolder('data/train',
crop_size=CROP_SIZE, upscale_factor=UPSCALE_FACTOR)
val_set = ValDatasetFromFolder('data/val',
upscale_factor=UPSCALE_FACTOR)
train_loader = DataLoader(dataset=train_set, num_workers=4,
batch_size=64, shuffle=True)
val_loader = DataLoader(dataset=val_set, num_workers=4,
batch_size=1, shuffle=False)

netG = Generator(UPSCALE_FACTOR) ##Generator definition
netD = Discriminator() ## Discriminator definition
generator_criterion = GeneratorLoss() ##Generator optimization
target

## Whether to use GPU
```

```
if torch.cuda.is_available().
netG.cuda()
netD.cuda()
generator_criterion.cuda()

## Generator and Discriminator Optimizer
optimizerG = optim.Adam(netG.parameters())
optimizerD = optim.Adam(netD.parameters())

results = {'d_loss': [], 'g_loss': [], 'd_score': [], 'g_score': [],
'psnr': [], 'ssim': []}
## epoch iterations
for epoch in range(1, NUM_EPOCHS + 1).
train_bar = tqdm(train_loader)
running_results = {'batch_sizes': 0, 'd_loss': 0, 'g_loss': 0,
'd_score': 0, 'g_score': 0} ##Results variables

netG.train() ##Generator training
netD.train() ## Discriminator training

## Data iteration per epoch
for data, target in train_bar.
g_update_first = True
batch_size = data.size(0)
running_results['batch_sizes'] += batch_size

## Optimize the discriminator to maximize D(x)-1-D(G(z))
real_img = Variable(target)
if torch.cuda.is_available().
real_img = real_img.cuda()
z = Variable(data)
if torch.cuda.is_available().
z = z.cuda()
fake_img = netG(z) ##Get the generated result
netD.zero_grad()
real_out = netD(real_img).mean()
fake_out = netD(fake_img).mean()
d_loss = 1 - real_out + fake_out
d_loss.backward(retain_graph=True)
optimizerD.step() ##Optimize the discriminator

## Optimization Generator Minimize 1-D(G(z)) + Perception Loss +
Image Loss + TV Loss
netG.zero_grad()
g_loss = generator_criterion(fake_out, fake_img, real_img)
g_loss.backward()

fake_img = netG(z)
fake_out = netD(fake_img).mean()
optimizerG.step()
```

```
# Record current losses
running_results['g_loss'] += g_loss.item() * batch_size
running_results['d_loss'] += d_loss.item() * batch_size
running_results['d_score'] += real_out.item() * batch_size
running_results['g_score'] += fake_out.item() * batch_size

## Validate against the validation set
netG.eval() ## Set the validation mode
out_path = 'training_results/SRF_' + str(UPSCALE_FACTOR) + '/'
if not os.path.exists(out_path).
os.makedirs(out_path)

## Calculate validation set related metrics
with torch.no_grad().
val_bar = tqdm(val_loader)
valing_results = {'mse': 0, 'ssims': 0, 'psnr': 0, 'ssim': 0,
'batch_sizes': 0}
val_images = []
for val_lr, val_hr in val_bar.
batch_size = val_lr.size(0)
valing_results['batch_sizes'] += batch_size
lr = val_lr ## low resolution truth map
hr = val_hr ## high resolution truth map
if torch.cuda.is_available().
lr = lr.cuda()
hr = hr.cuda()
sr = netG(lr) ## image super-resolution reconstruction results

batch_mse = ((sr - hr) ** 2).data.mean() ## Calculate MSE metrics
valing_results['mse'] += batch_mse * batch_size
valing_results['psnr'] = 10 * log10(1 / (valing_results['mse'] /
valing_results['batch_sizes'])) ## Calculate PSNR metrics
batch_ssim = pytorch_ssim.ssim(sr, hr).item() ## Calculate SSIM
metrics
valing_results['ssims'] += batch_ssim * batch_size
valing_results['ssim'] = valing_results['ssims'] /
valing_results['batch_sizes']
## Store model parameters
torch.save(netG.state_dict(), 'epochs/netG_epoch_%d_%d.pth' %
(UPSCALE_FACTOR, epoch))
torch.save(netD.state_dict(), 'epochs/netD_epoch_%d_%d.pth' %
(UPSCALE_FACTOR, epoch))
## Record the loss of the training set and the psnr,ssim and other
metrics of the validation set \scores\psnr\ssim
results['d_loss'].append(running_results['d_loss'] /
running_results['batch_sizes'])
results['g_loss'].append(running_results['g_loss'] /
running_results['batch_sizes'])
results['d_score'].append(running_results['d_score'] /
running_results['batch_sizes'])
```

```
results['g_score'].append(running_results['g_score'] /
running_results['batch_sizes'])
results['psnr'].append(valing_results['psnr'])
results['ssim'].append(valing_results['ssim'])

## Store results to local file
if epoch % 10 == 0 and epoch ! = 0.
out_path = 'statistics/'
data_frame = pd.DataFrame(
data={'Loss_D': results['d_loss'], 'Loss_G': results['g_loss'],
'Score_D': results['d_score'].
'Score_G': results['g_score'], 'PSNR': results['psnr'], 'SSIM':
results['ssim']}.
index=range(1, epoch + 1))
data_frame.to_csv(out_path + 'srf_' + str(UPSCALE_FACTOR) +
'_train_results.csv', index_label='Epoch')
```

From the above code, we can see that the crop_size used for training is 240 × 240, we scale all the image to 320 × 320 for training, the image size is 320 × 320 for validation, the batch size is 64, and the optimizer used is Adam, which uses the default optimization parameters. We trained the model with an upsampling multiplier of 4.

8.6.2.2 Training Results

Next, we train the model without adding JPEG compression noise, adding 0–50% amplitude compression noise, and adding 70–99% amplitude compression noise, respectively.

The result curves of PSNR and SSIM after training 100 epochs are shown in Fig. 8.22.

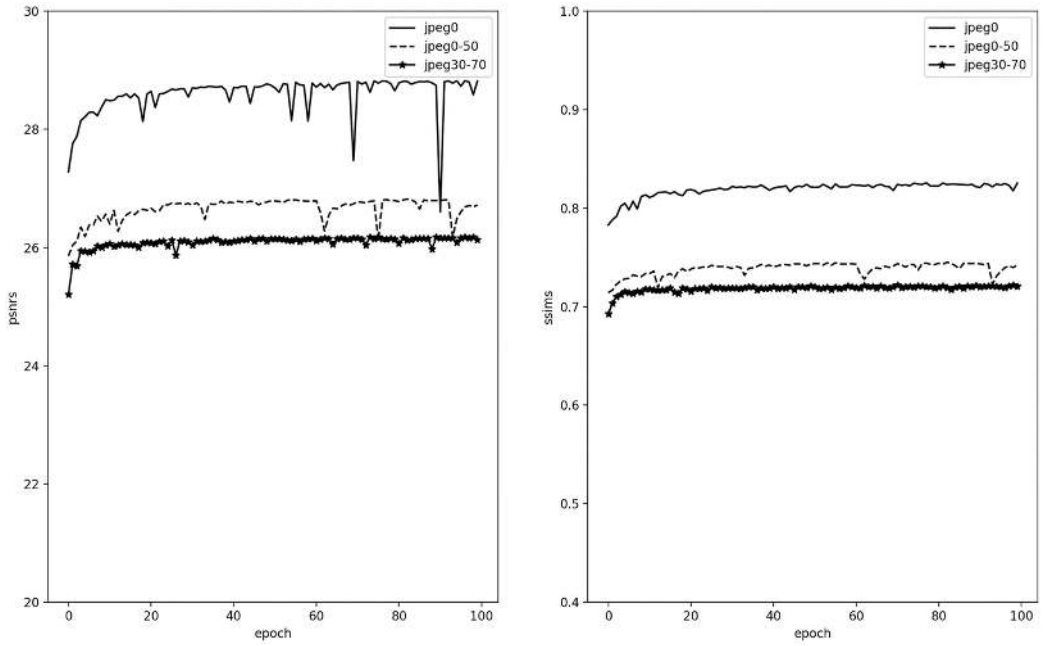


Fig. 8.22 PSNR and SSIM curves for 4× upsampling

The curves corresponding to jpeg0 in Fig. 8.22 represent the training results without adding JPEG compression noise, and the curves corresponding to jpeg0–50 and jpeg30–70 represent

the training results with adding 0–50% amplitude of compression noise and adding 30–70% amplitude of compression noise, respectively. It can be seen that the model has basically converged, and the larger the added noise amplitude is, the lower the final PSNR index and SSIM index will be.

8.6.3 Model Testing

Next we use our own data to test the model.

8.6.3.1 Test Code

First we interpret the test code, which needs to complete the loading of the model, image preprocessing and storage of the results, the complete code is as follows:

```
import torch
from PIL import Image
from torch.autograd import Variable
from torchvision.transforms import ToTensor, ToPILImage
from model import Generator

UPSCALE_FACTOR = 4 ## upsampling multiplier
TEST_MODE = True ## Use GPU for testing

IMAGE_NAME = sys.argv[1] ## Image path
RESULT_NAME = sys.argv[2] ## Result image path

MODEL_NAME = 'netG.pth' ## Model path
model = Generator(UPSCALE_FACTOR).eval() ## Set as verification mode
if TEST_MODE.
model.cuda()
model.load_state_dict(torch.load(MODEL_NAME))
else.
model.load_state_dict(torch.load(MODEL_NAME, map_location=lambda
storage, loc: storage))

image = Image.open(IMAGE_NAME) ## Read the image
image = Variable(ToTensor()(image), volatile=True).unsqueeze(0) ##
Image preprocessing
if TEST_MODE.
image = image.cuda()

out = model(image)
out_img = ToPILImage()(out[0].data.cpu())
out_img.save(RESULT_NAME)
```

8.6.3.2 Reconstruction Results

Figure 8.23 shows the super-resolution result of a live image, where the input image is scaled from 512 × 512 to 128 × 128 size, and then stored in JPEG and PNG formats, respectively, by using opencv's imwrite function, with the former using the default JPEG compression ratio of the opencv library.

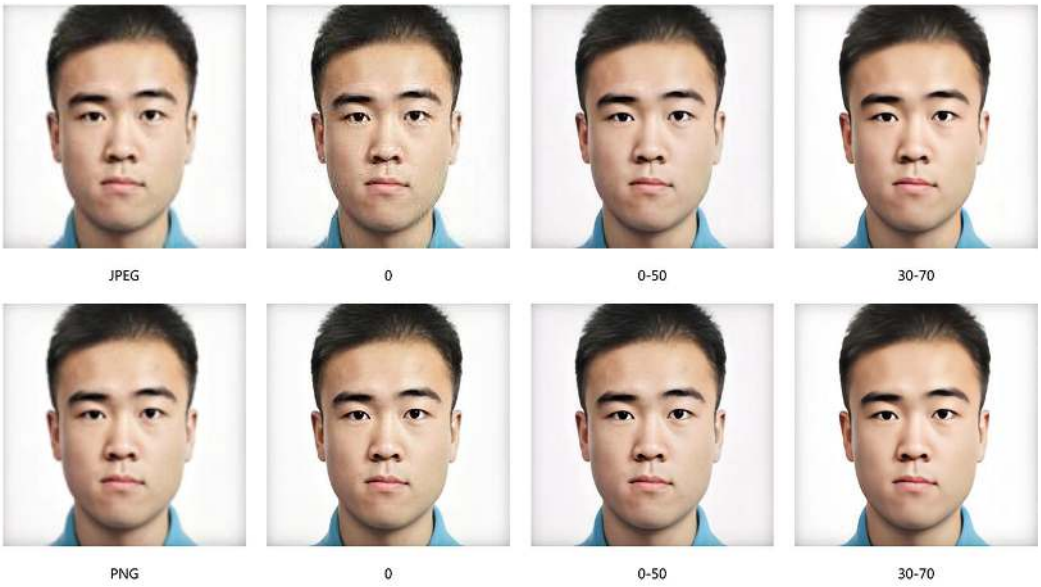


Fig. 8.23 SRGAN results for real human images

In Fig. 8.23, column 1 is the original image, where two rows are in JPEG and PNG formats, respectively, which are displayed by using bilinear interpolation for upsampling. Column 2 shows the 4x super-resolution results after training without adding JPEG noise data augmentation, column 3 shows the 4x super-resolution results after training with JPEG noise data augmentation of 0–50% random amplitude, and column 4 shows the 4x super-resolution results after training with JPEG noise data augmentation of 30–70% random amplitude.

Comparing row 1 and row 2, it can be seen that for JPEG compressed images, if no noise data augmentation is added, the result map will amplify the noise in the original image, which can be easily seen from the result of column 2. However, the amplitude of noise should not be too large; otherwise, the reconstruction result will be distorted. Comparing the result of column 4 with that of column 3, although column 4 has stronger noise suppression ability, the face image has started to show distortion, such as the skin is too smooth and the eyes are obviously distorted. Therefore, we cannot simply increase the noise amplitude.

Although our training dataset is composed of real human images, the model can also generalize to face images in other domains, and Fig. 8.24 shows the super-resolution results of an anime image.

Fig. 8.24 SRGAN results for anime images

We can draw the same conclusion as Fig. 8.23, where the best super-resolution performance is obtained after adding a moderate augmentation of the noisy data.

8.6.4 Summary

In this section, we have practiced the SRGAN model. We used the high-definition face dataset for training, and performed image super-resolution reconstruction of low-resolution face images to verify the effectiveness of the SRGAN model, although the model still has a large room for improvement, it needs to use pairs of datasets for training, and the pattern generation of low-resolution images during training is too simple to complete the reconstruction of complex degradation types.

When super-resolution reconstruction is to be performed on images with more complex degradation types, the model training should also adopt data augmentation methods corresponding to real scenes, including but not limited to contrast enhancement, various types of noise pollution, JPEG compression, and other operations, which are left to the reader for experimentation.

References

1. Ponomarenko N, Jin L, Ieremeiev O, et al. Image database TID2013: Peculiarities, results and perspectives [J]. Signal Processing: Image Communication, 2015, 30: 57–77.

2. Anaya J, Barbu A. RENOIR-A dataset for real low-light image noise reduction [J]. Journal of Visual Communication and Image Representation, 2018, 51: 144–154.
[Crossref]

3. Chen J, Chen J, Chao H, et al. Image blind denoising with generative adversarial network based noise modeling [C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018: 3155–3164.

4. Nah S, Hyun Kim T, Mu Lee K. Deep multi-scale convolutional neural network for dynamic scene deblurring [C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2017: 3883–3891.

5. Kupyn O, Budzan V, Mykhailych M, et al. Deblurgan: Blind motion deblurring using conditional adversarial networks [C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2018: 8183–8192.

6. Kupyn O, Martyniuk T, Wu J, et al. Deblurgan-v2: Deblurring (orders-of-magnitude) faster and better [C]//Proceedings of the

IEEE International Conference on Computer Vision. 2019: 8878–8887.

7. Zhang K, Luo W, Zhong Y, et al. Deblurring by Realistic Blurring [C]//2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2020.

8. Bychkovsky V, Paris S, Chan E, et al. Learning photographic global tonal adjustment with a database of input/output image pairs [C]//CVPR 2011. IEEE. 2011: 97–104.

9. Ignatov A, Kobyshev N, Timofte R, et al. DSLR-quality photos on mobiledevices with deep convolutional networks [C]//Proceedings of the IEEEInternational Conference on DSLR-quality photos on mobile Computer Vision. 2017: 3277–3285.

10. Ignatov A, Kobyshev N, Timofte R, et al. WESPE: weakly supervised photo enhancer for digital cameras [C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops. 2018: 691–700.

11. Triantafyllidou D, Moran S, McDonagh S, et al. Low Light Video Enhancement using Synthetic Data Produced with an Intermediate Domain Mapping [J]. arXiv preprint arXiv:2007.09187, 2020.

12. Ledig C, Theis L, Huszár F, et al. Photo-realistic single image super-resolution using a generative adversarial network [C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2017: 4681–4690.

13. Wang X, Yu K, Wu S, et al. Esrgan: Enhanced super-resolution generative adversarial networks [C]//Proceedings of the European Conference on Computer Vision (ECCV). 2018: 0–0.

14. Bulat A, Yang J, Tzimiropoulos G. To learn image super-resolution, use a gan to learn how to do image degradation first [C]//Proceedings of the European conference on computer vision (ECCV). 2018: 185–200.

15. Menon S, Damian A, Hu S, et al. PULSE: Self-Supervised Photo Upsampling via Latent Space Exploration of Generative Models [J]. arXiv: Computer Vision and Pattern Recognition, 2020.

16. Pathak D, Krahenbuhl P, Donahue J, et al. Context encoders: feature learning by inpainting [C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 2536–2544.

17. Iizuka S, Simo-Serra E, Ishikawa H. Globally and locally consistent image completion [J]. ACM Transactions on Graphics (ToG), 2017, 36(4): 1–14.
[Crossref]

18. Wan Z, Zhang B, Chen D, et al. Bringing old photos back to life[C]//proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2020: 2747–2757.

19. Karras T, Aila T, Laine S, et al. Progressive growing of gans for improved quality, stability, and variation [J]. arXiv preprint arXiv:1710.10196, 2017.

20. Shi W, Caballero J, Huszár F, et al. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 1874–1883.

P. Long, X. Guo, *Generative Adversarial Network*
https://doi.org/10.1007/978-981-96-9404-4_9

9. 3D Image and Video Generation

Peng Long[1] ✉ and Xiaozhou Guo[2]
(1) Beijing YouSan Educational Technology, Beijing, China
(2) • China Electronics Technology Group Corporation No. 54 Research Institute, Shijiazhuang, China

Abstract
This chapter introduces 3D image and video generation using GANs. For 3D image generation, frameworks like Visual Object Networks decompose tasks into shape, projection, and texture modules. PrGAN infers 3D shapes from 2D views via adversarial training. Video generation models include Video-GAN (separating static/dynamic components) and MoCoGAN (decoupling content/motion spaces). MoCoGAN-HD extends StyleGAN for high-resolution video synthesis by manipulating latent vectors. MD-GAN uses a two-stage pipeline (content generation + motion refinement) with Gram matrix constraints. The chapter addresses challenges in temporal coherence, resolution, and dataset scarcity (e.g., ShapeNet for 3D shapes).

Keywords 3D generation – Video synthesis – MoCoGAN – Temporal consistency

In Chap. 5, we introduced the core techniques of image generation in detail. Nowadays, researchers are also gradually starting to study more complex GAN-based video generation and 3D image generation problems, and in this chapter we briefly introduce some of the technical frameworks.

9.1 3D Image and Video Generation Applications

In this section, we first introduce the applications related to 3D image generation and video generation. Compared with 2D image generation, the current 3D image and video generation has not yet reached the effect of indistinguishing truth from falsehood.

9.1.1 3D Image Generation Applications

The real world we live in is three-dimensional, and with the gradual saturation of two-dimensional image algorithm research, academia and industry are now beginning to focus more on three-dimensional image research, of which three-dimensional image generation is a subdirection. Figure 9.1 shows some 3D images generated by the model.

Fig. 9.1 3D image generated from the model

Since 3D image datasets are more difficult to acquire compared to 2D images, 3D image generation for expanding training datasets is a promising application.

9.1.2 Video Generation and Prediction Applications

Video generation and prediction are two related but different problems, each with a common technical framework but also with a different focus.

9.1.2.1 Video Generation Application

Video generation can be seen as an extension of the image generation task, which requires the generation of temporally stable image sequences. High-quality video generation can be used in application areas such as artwork creation and dataset expansion.

Figure 9.2 shows some of the generated video frames, but the current video generation is not yet able to reach the high-quality level of image generation.

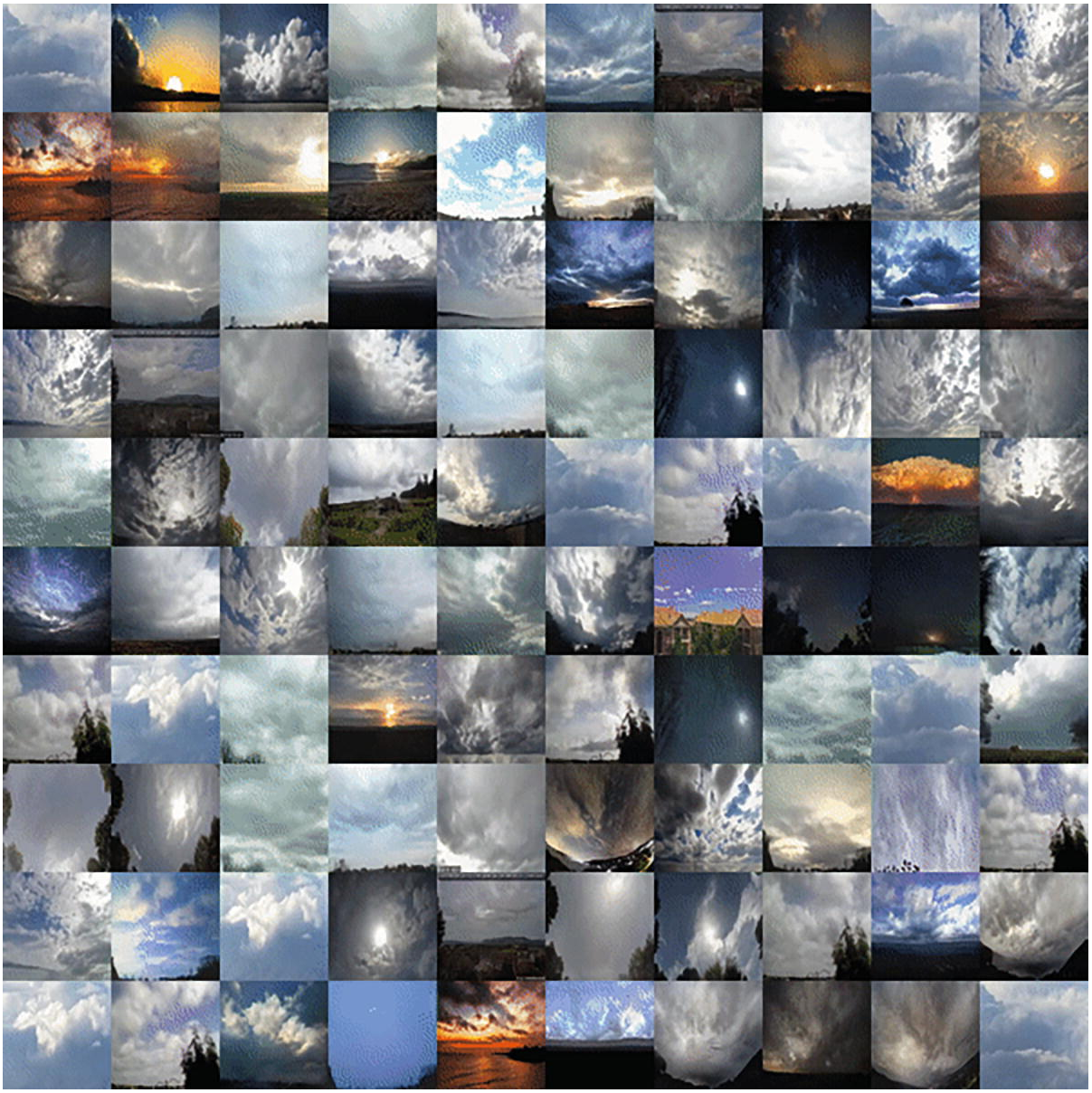


Fig. 9.2 Generated video frames

9.1.2.2 Video Prediction Applications

Video prediction differs from video generation in that it predicts the output of the next image frame from the previous frame.

Figure 9.3 shows a case of video frame sequence prediction, where the original image is a real shot still image, and the image generation framework can simulate the hot air effect.

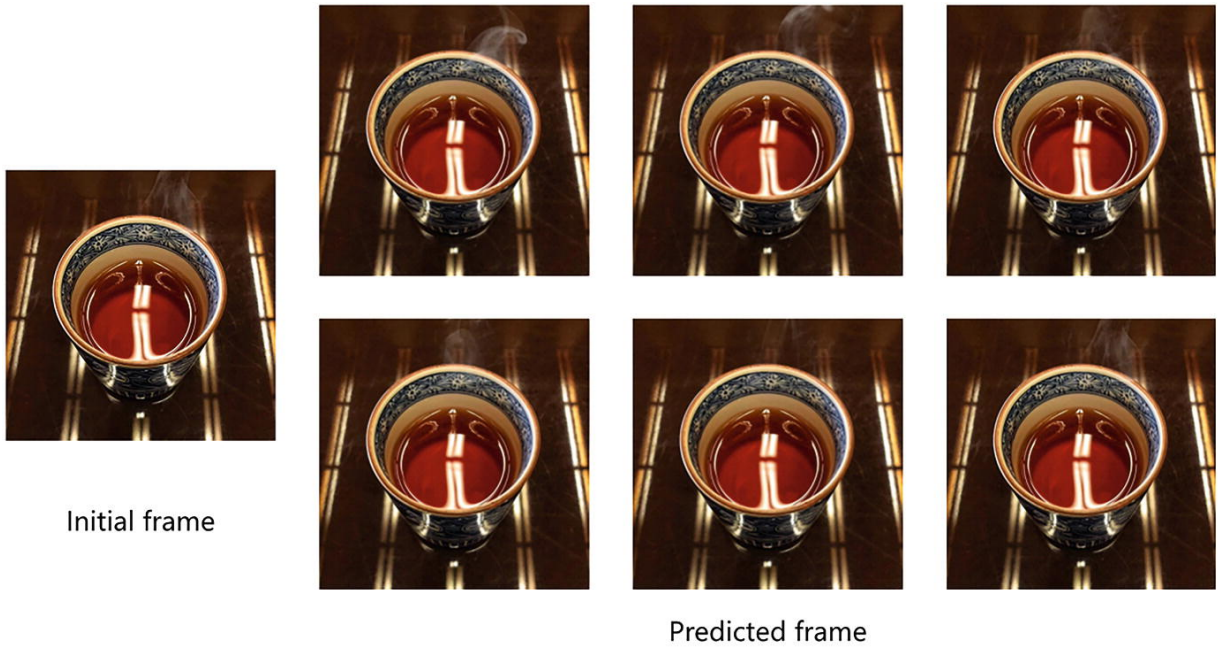


Fig. 9.3 Video prediction

9.2 3D Image Generation Framework

Real-life images are three-dimensional, and in recent years there has been an increasing amount of research on three-dimensional images. By extending convolution from 2D to 3D, GAN can also be used for 3D image generation [1].

9.2.1 General 3D Image Generation Framework

3D images need to generate not only 3D shapes but also realistic textures. Google researchers have pioneered a complete framework for 3D image generation (Visual Object Network [2]), which generates 3D shapes, 2.5D contours and depths, and 2D textures in three steps in sequence.

The framework consists of three networks, the shape network, the contour and depth projection network, and the texture network (Fig. 9.4).

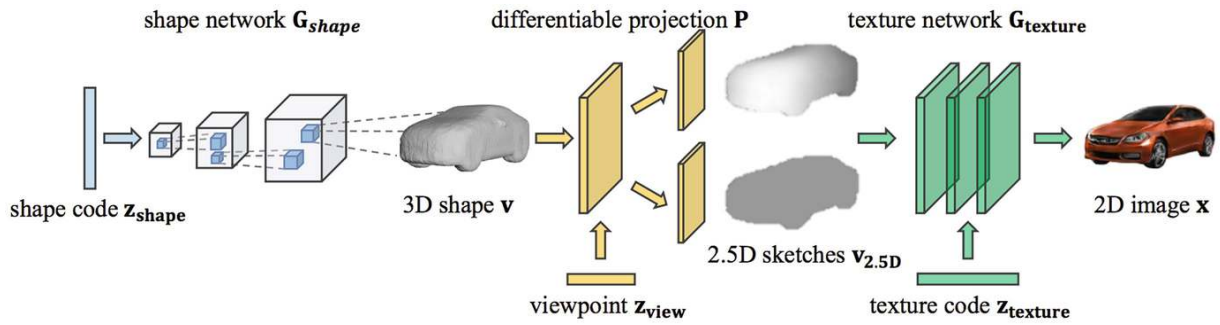


Fig. 9.4 Visual object networks framework

1. Shape generation network with a one-dimensional shape encoding z as input and a 3D shape v as output. It is a normal generative adversarial network, except that the generated image is changed from 2D to 3D and the corresponding convolution kernel is changed from 2D to 3D.
2. Contour and depth projection network with input 3D shape v and output 2.5D depth and contour. The purpose of this step is to implement projections of 3D models and 2D images based on 2.5D, where the projection matrix can sample the pose from the empirical distribution.
3. Texture generation network with input 2.5D contours and output 2D textures, i.e., 2D images. It uses CycleGAN architecture to implement one-to-many mapping.

Because the three modules are conditionally independent, the models do not require paired data between 2D and 3D shapes, and thus each can be trained on large-scale 2D image and 3D shape datasets. The classic 3D shape dataset, ShapeNet [3], contains thousands of CAD models for 55 object classes.

9.2.2 2D to 3D Prediction Framework

The 3D image generation framework introduced in the previous subsection is a framework for generating 3D objects from scratch, which is similar to the DCGAN model, where the input is a noise vector, the generation result cannot be controlled and has a high training difficulty. We have introduced the conditional GAN model in Chap. 5,

which controls the generated results by inputting some conditions, and for the 3D image generation task, we can also provide some additional information to improve the quality of the generated results.

PrGAN (Projective Generative Adversarial Network) [4] is a 3D shape prediction GAN framework that can be used to infer the true 3D shape from multiple 2D graphs, as shown in Fig. 9.5.

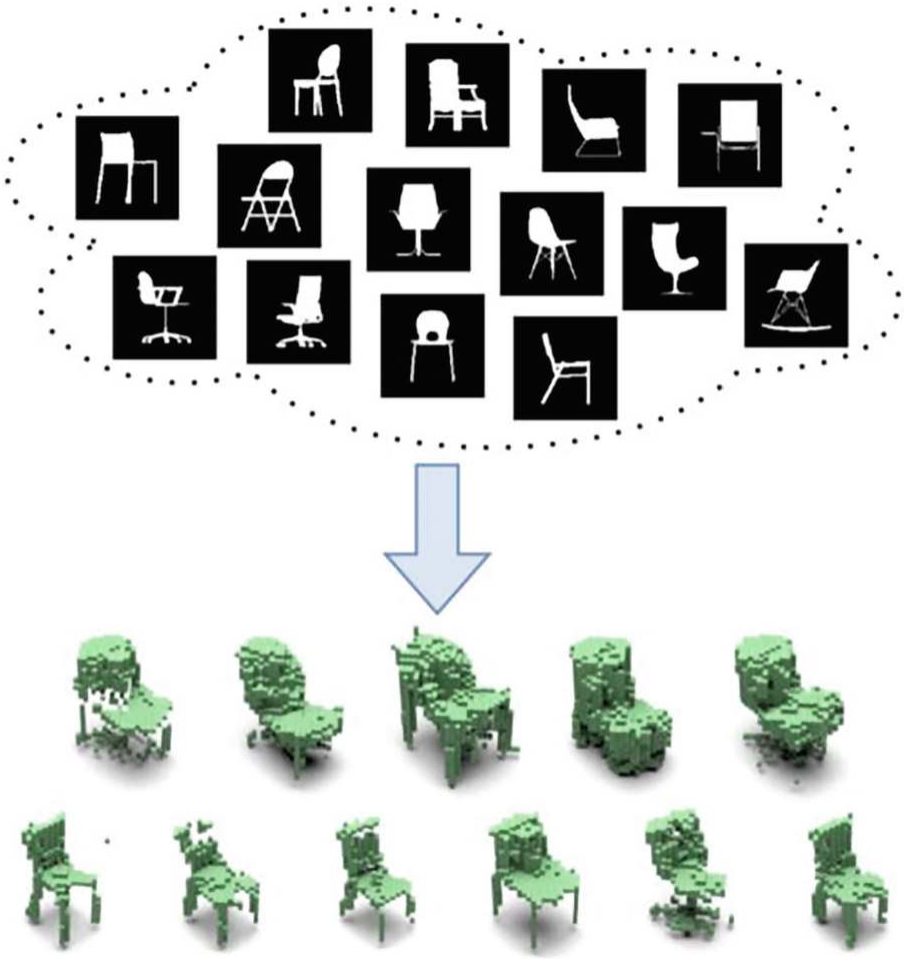


Fig. 9.5 Predicting 3D shapes based on 2D drawings

Figure 9.6 shows the flow of the framework, containing the 3D shape generator, the projection module, and the discriminator. The input vector is a 201-dimensional random noise, and the generated 3D shape and view angle are obtained by the 3D shape generator and projection module (θ, φ). The generated 3D shape is then projected to obtain a 2D image. The discriminator is used to determine whether the input 2D image is generated or real.

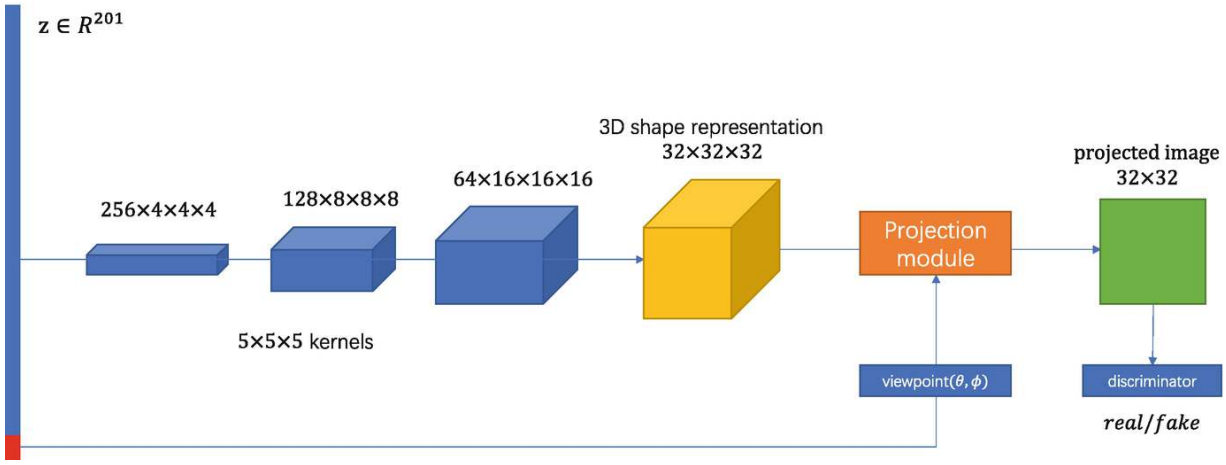


Fig. 9.6 PrGAN framework

1. The generator structure consists of four 3D convolutional layers. The first convolutional layer inputs a 200-dimensional vector, which then passes through a fully connected layer to obtain an output of 256 × 4 × 4 × 4, and then three upsampling layers to output 32 × 32 × 32 voxels, where the convolutional kernel size is all 5 × 5 × 5.
2. The projection module uses the last one dimension of z and outputs the angle(θ, φ). It is actually a uniform division of the y-direction into eight parts, which are randomly selected according to the value of z.

The training method of PrGAN does not differ from the basic 2D GAN generation model. However, the PrGAN framework has some shortcomings, including too low resolution, only binary input information is utilized, and simulated images are used, which can be improved subsequently.

9.3 Video Generation and Prediction Framework

While generative adversarial network technology has developed fast in recent years, image generation can achieve real simulation effects; video generation is an extension of image generation applications and

possesses a higher level of difficulty. Video generation is not only to generate multiple realistic images, but also to ensure the coherence of the motion and even to predict the real motion. In this section, we will introduce the framework for video generation.

9.3.1 Basic Video-GAN

Video-GAN [5] can be considered as the video version of the image generation framework DCGAN, which generates continuous video frames from random noise by 3D convolution, and the framework structure is shown in Fig. 9.7.

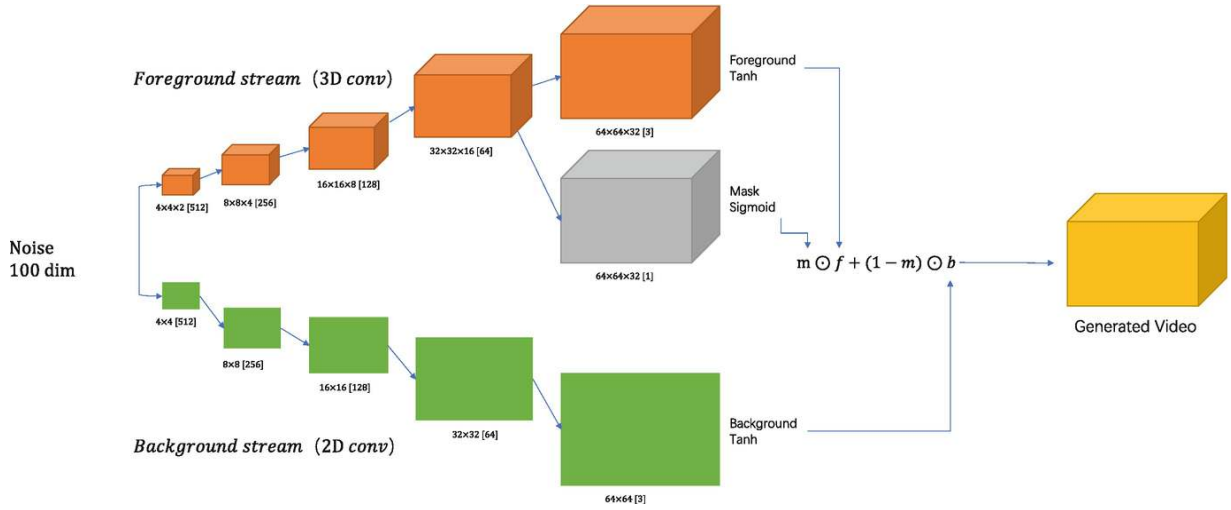


Fig. 9.7 Video-GAN

To simplify the problem, the frame does not take into account the motion of the camera itself, so the background does not move and the whole video consists of a static background and a dynamic foreground.

Video-GAN consists of a two-stream architecture that generates the background and foreground separately. Because the background is static, the background generation branch is a generative network composed of 2D convolution, with 4 × 4 sized features as input and 64 × 64 sized images as output. And the foreground is dynamic, so the foreground generation branch is a generative network composed of 3D convolution with 4 × 4 × 2 sized features as input and 64 × 64 × 32 sized videos as output, each of which includes 32 frames. The generators all use the same architecture as the generators in DCGAN.

The foreground $f(z)$ and background $b(z)$ are fused in the following way, i.e., a mask $m(z)$ is used for linear fusion.

$$G_2(z) = m(z) \bigodot f(z) + (1 - m(z)) \bigodot b(z) \tag{9.1}$$

9.3.2 Multi-Stage MD-GAN

Another more common scenario than direct video generation is to predict the next frames by inputting one frame of an image, which is called the video prediction problem.

The aforementioned Video-GAN model can also be used to do video prediction by simply converting the input image into feature vectors by adding encoders to the front end of the network, which are used to replace the noise vectors, and the subsequent model structure does not need to be adjusted.

Next, we introduce another more refined generation framework, namely MD-GAN [6], which improves the generation of consecutive frames in two steps, and the whole network framework is shown in Fig. 9.8.

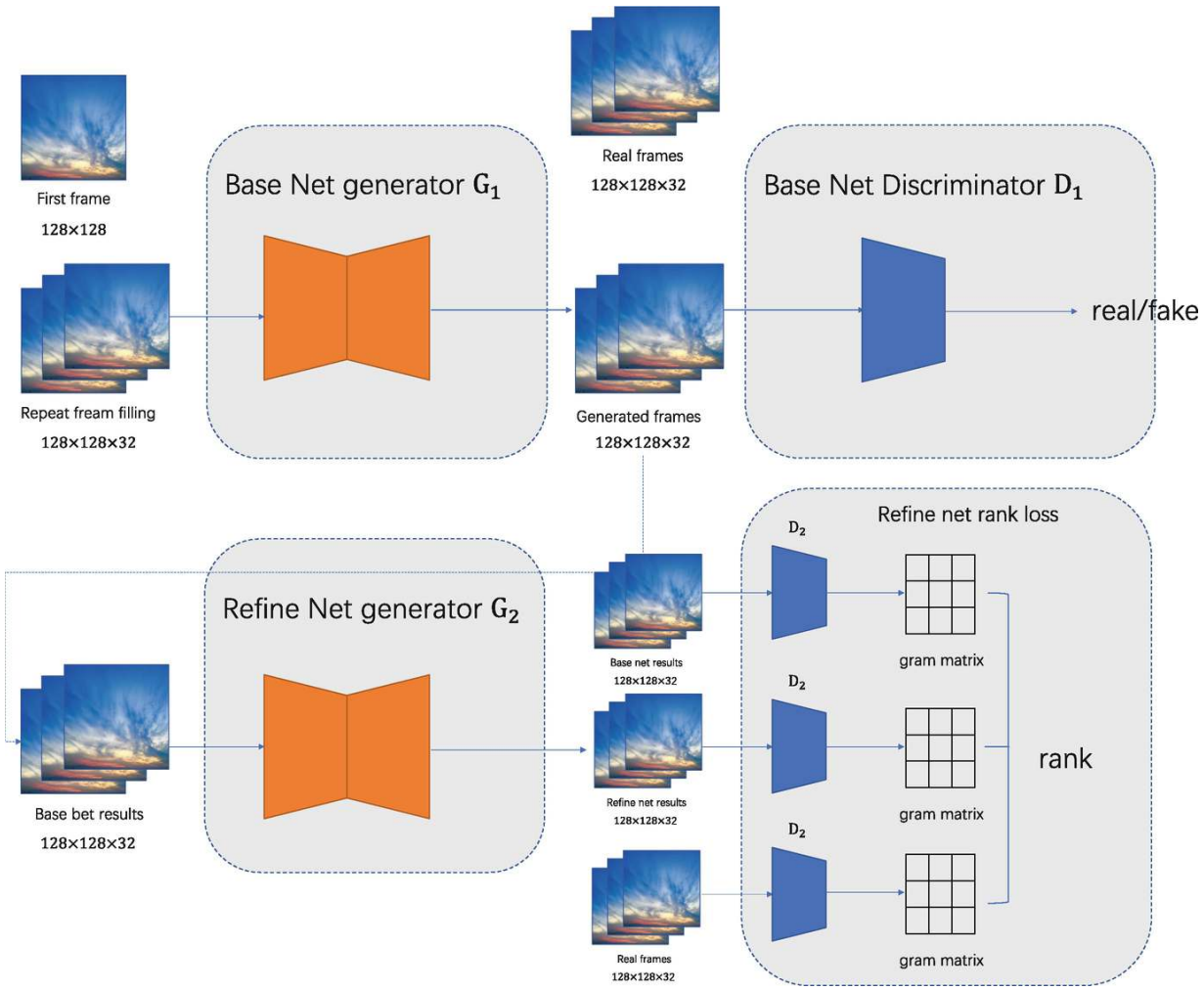


Fig. 9.8 MD-GAN framework diagram

The MD-GAN framework is divided into two phases:

The first stage (Base-Net): generating the content of each frame, it should focus on the realism of each frame content. The second stage (Refine-Net) focuses on optimizing the motion of objects between frames, making the generated results smoother.

First we take a look at Base-Net, which contains the generator G_1 and the discriminator D_1.

- G_1 uses a encoder-decoder structure that contains multiple 3D convolutional layer-deconvolutional layer pairs and uses skip connections, which is a typical U-Net structure that implements modeling of video content with an optimization target of pixel-by-pixel L1 distance.

- D_1 adopts the encoder part of the network from G_1, only replacing the ReLU activation function with the Sigmoid activation function in the last layer.

Then we take a look at Refine-Net, which contains the generator G_2 and the discriminator D_2.

- G_2 is much like G_1, but with some of the skip connections removed, because the authors found that connections with high and low levels of feature can have a negative impact on the dynamic modeling of videos.
- D_2 contains three discriminators, each of which has the same structure as D_1, which focuses on the need to model the Gram matrix and the ranking loss.

The Gram matrix is a covariance matrix between features, which is well suited for characterizing texture features of images and is widely used in the field of stylization. Once the Gram matrix is obtained, it can be used to calculate the ranking loss.

Suppose the video output from Base-Net is Y_1, the video output from Refine-Net is Y_2, and the real video is Y. The sequencing loss is to constrain the distance between Y_2 and the real video to be smaller than the distance between Y_1 and the real video, as defined in Eq. (9.2), which is a classical contrastive loss.

$$L_{\text{rank}}(Y_1, Y_2, Y; l) = -\log \frac{e^{-\|g(Y_2;l)-g(Y;l)\|_1}}{e^{-\|g(Y_2;l)-g(Y;l)\|_1} + e^{-\|g(Y_2;l)-g(Y_1;l)\|_1}} \quad (9.2)$$

In order to constrain the content details of the front and back frames, content reconstruction loss also needs to be added when training Refine-Net.

9.3.3 MoCoGAN with Content and Action Separated

MocoGAN [7] decomposes video generation into two parts, content generation and action generation, to achieve freer control of results in both subspaces, and the schematic diagram of the framework is shown in Fig. 9.9.

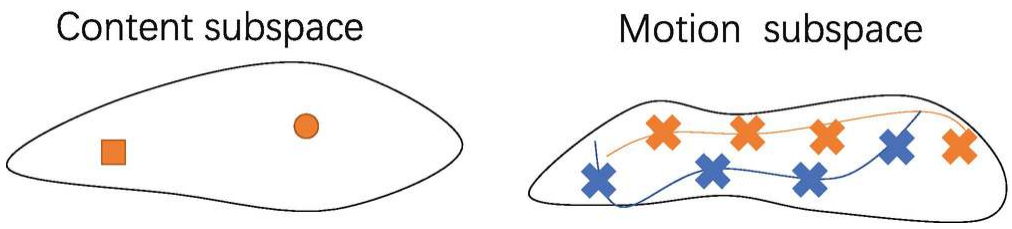


Fig. 9.9 Schematic diagram of content space and motion space

Each point in the latent space represents a image, denoted as Z_I, and a video of length K can be represented using a path of length K as $[Z(1), ..., Z(K)]$, and Z_I is further decomposed into two subspaces, the content space Z_C and the motion space Z_M, respectively. The content space models only content changes unrelated to motion, and the motion space models only motion changes unrelated to content. For example, a smiling face can model identity information using the content space and facial muscle movement using the motion space.

The content space can be modeled using a Gaussian distribution and the same model can be used for each frame. The modeling of the motion space, on the other hand, can be done using RNN models.

The modeling of content space and motion space is shown in Fig. 9.10.

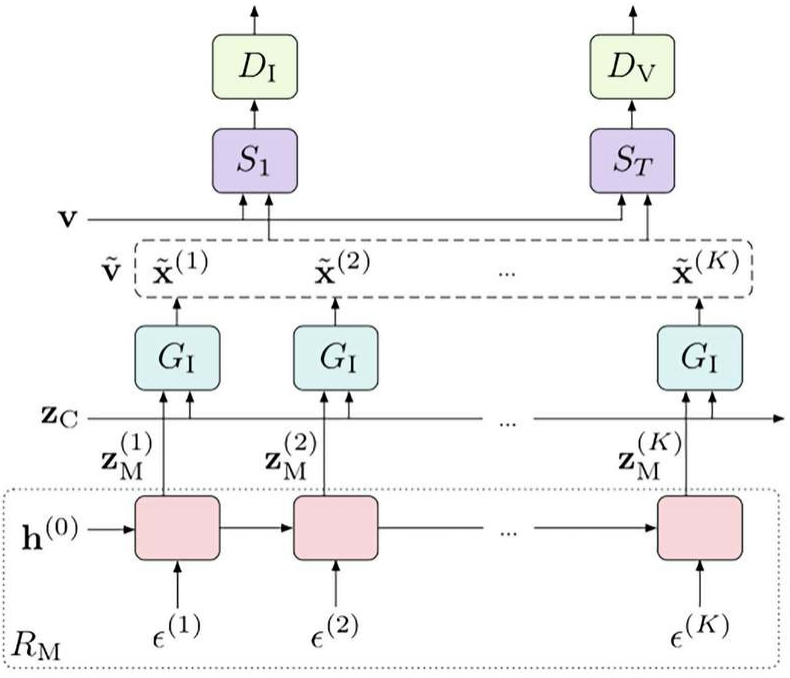


Fig. 9.10 Content space and motion space modeling

In Figs. 9.10, the G_I is the image generation network, D_I is the image discrimination network, and D_V is the video discrimination network, and R_M is the RNN network, and v is the real video, and $\tilde{v}$ is the generated video.

The input fixed-length video of D_V is used to determine whether the video is from a real video or a generated video on the one hand, and to determine the motion of the video on the other hand.

Subsequent researchers have proposed an improved version of MocoGAN [8]. MocoGAN-HD, which is based on the assumption that if there is an image generator that can generate a clear image for every frame, then the video can be represented as a set of hidden variables in the hidden space of this generator, and the video synthesis problem is to discover a set of hidden variables that satisfy temporal consistency.

StyleGAN, which we introduced in the previous sections, is the compliant image generator, and we have shown in the face image editing section that smooth changes in face attributes can be achieved based on the editing of latent vectors.

MocoGAN-HD is the video generation framework based on StyleGAN v2. Figures 9.11 and 9.12 show the video generation results given in reference [8] for the same initial content vector, different action vectors, and different initial content vector, same action vector, respectively.

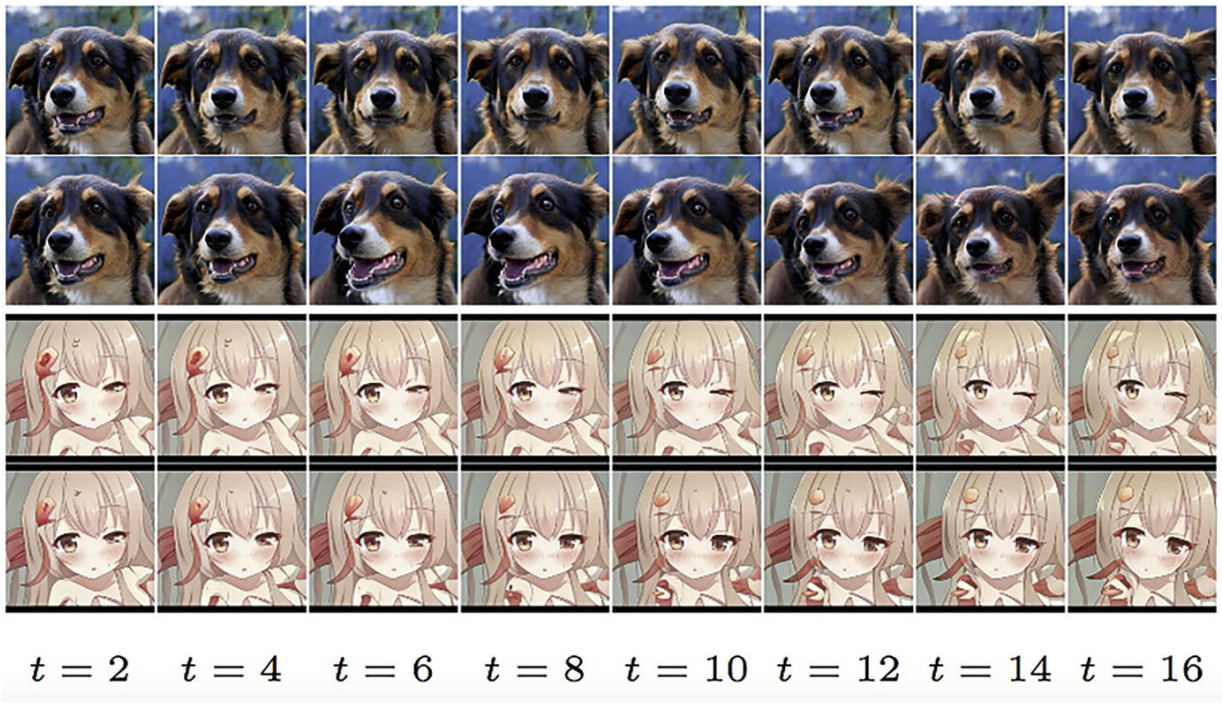


Fig. 9.11 Same initial content vector, different action vectors, *t* indicates different moments

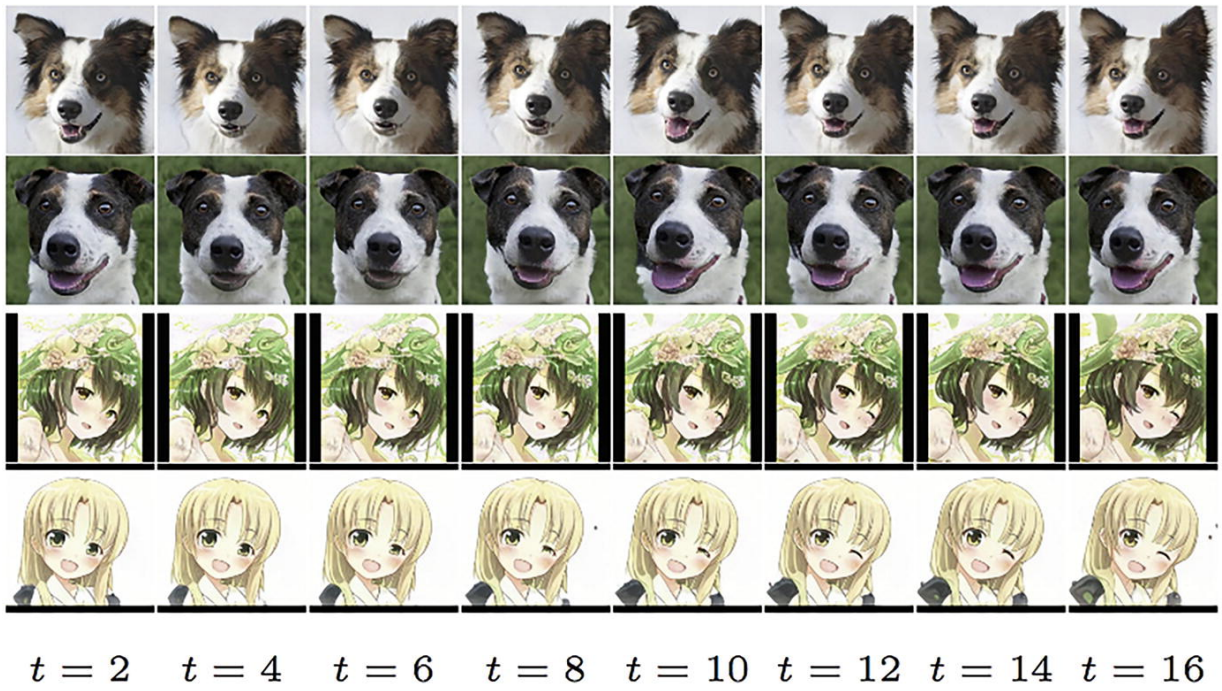


Fig. 9.12 Different initial content vectors, same action vector, *t* indicates different moments

It can be seen that the generated video frames already have a better realism, and readers can look for the corresponding open source

projects to see more video generation results.

Of course, the current research on 3D image generation framework and video generation framework is not mature, so we do not add practical content later in this chapter. Readers can expand their own learning, related practice. With the maturity of 2D image processing and computer vision development, 3D image and video is the current direction of rapid progress in the field of imaging, readers can follow on their own.

References

1. Wu J, Zhang C, Xue T, et al. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling[J]. Advances in neural information processing systems, 2016, 29.

2. Zhu J Y, Zhang Z, Zhang C, et al. Visual object networks: Image generation with disentangled 3D representations[J]. Advances in neural information processing systems, 2018, 31.

3. Chang A X, Funkhouser T, Guibas L, et al. Shapenet: An information-rich 3d model repository[J]. arXiv preprint arXiv:1512.03012, 2015.

4. Gadelha M, Maji S, Wang R. 3d shape induction from 2d views of multiple objects[C]//2017 international conference on 3d vision (3DV). IEEE, 2017: 402–411.

5. Vondrick C, Pirsiavash H, Torralba A. Generating videos with scene dynamics [C]//Advances in neural information processing systems. 2016: 613–621.

6. Xiong W, Luo W, Ma L, et al. Learning to generate time-lapse videos using multi-stage dynamic generative adversarial networks [C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018: 2364–2373.

7. Tulyakov S, Liu M Y, Yang X, et al. Mocogan: Decomposing motion and content for video generation [C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2018: 1526–1535.

8. Tian Y, Ren J, Chai M, et al. A good image generator is what you need for high-resolution video synthesis[J]. arXiv preprint arXiv:2104.15069, 2021.

P. Long, X. Guo, *Generative Adversarial Network*
https://doi.org/10.1007/978-981-96-9404-4_10

10. General Image Editing

Peng Long[1] ✉ and Xiaozhou Guo[2]
(1) Beijing YouSan Educational Technology, Beijing, China
(2) • China Electronics Technology Group Corporation No. 54 Research Institute, Shijiazhuang, China

Abstract
This chapter covers advanced image editing tasks: depth-of-field manipulation (e.g., RefocusGAN for bokeh effects via deblurring/refocusing), image fusion (e.g., GP-GAN blending Poisson equations with GANs), and interactive editing (e.g., SPADE using spatially adaptive normalization for semantic synthesis). Depth editing simulates optical effects via focus control. Fusion frameworks balance color consistency and gradient alignment. SPADE enables high-fidelity image generation from semantic masks by preserving spatial context. The chapter emphasizes practical tools (e.g., Focos for depth editing) and challenges in realism and user accessibility.

Keywords Depth editing – Poisson fusion – SPADE – Interactive editing – Generative models

The development of GAN technology has not only greatly enhanced the level of development of some classical computer vision tasks, such as the field of image generation and enhancement, but also brought new solution ideas to some relatively new and complex vision tasks. In Chap. 7, we have introduced some applications of GAN in face image editing, which belong to a specific domain and have achieved results that can be practically implemented. And in this section we introduce some more general image editing tasks, which are still far from product

implementation, but it is meaningful to understand some typical ideas in GAN during the process of solving these tasks.

10.1 Image Depth Editing

Photographs are camera recordings of real physical scenes, which have depth, and there is a research direction in the field of computer vision, namely image depth estimation. By estimating the depth, we can get the distance of the target and help to identify different targets. By editing the depth, it affects the effect of depth of field of the front background in the photo, which has applications in autonomous driving, photographic image processing.

In this section, we introduce the GAN-based deep editing framework.

10.1.1 Depth and Depth of Field

In this section, we will first understand what depth is, what depth of field is, and the changes that are brought to the aesthetics of an image when editing the depth of field of an image.

10.1.1.1 Basic Concepts

Figure 10.1 shows an RGB image with the corresponding color depth map, where similar colors indicate similar depths.

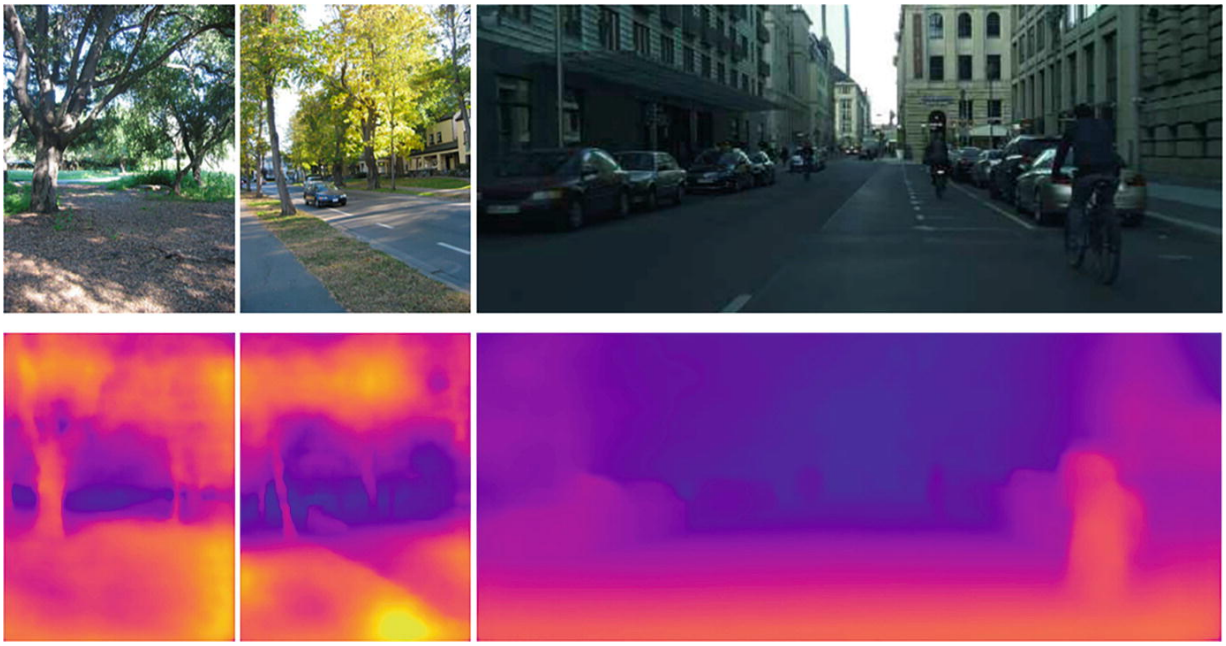


Fig. 10.1 RGB image with depth map

The first row in Fig. 10.1 is the RGB image in the natural scene, and the second row is the corresponding depth map. The depth information can be represented using grayscale values, which are mapped to pseudo-color here to enhance the display.

In photographic images, we can get the aesthetic effect of bokeh by having different targets with different imaging clarity by the parameters of the camera.

For the camera, when the subject is located in front of the lens (The front and back of the focus) within a certain length of space, its imaging on the negative is located between the same dispersion circle, presented to the human eye is the feeling of clear imaging, the length of this space that is the depth of field, also known as DOF (Depth of Field), when more than the depth of field, the imaging gradually blurred, the schematic diagram is shown in Fig. 10.2.

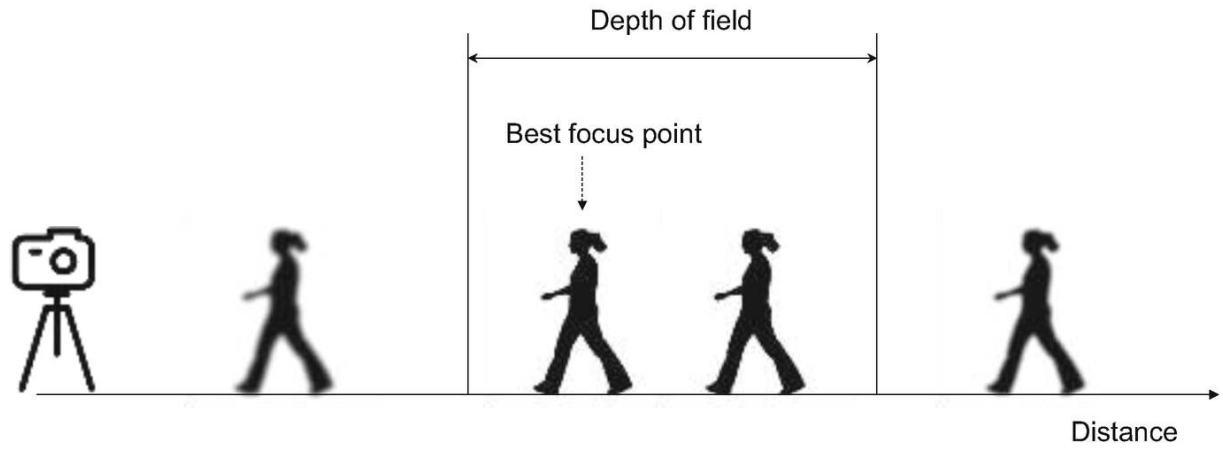


Fig. 10.2 Depth of field and imaging clarity

When capturing images, maintain a certain distance between the subject and the background, adjust the focus ring on the lens so that the subject is within the depth of field and the background is outside the depth of field, resulting in clear subject imaging and blurry background imaging, commonly known as background blurring.

The image captured by a large aperture lens has a shallow depth-of-field effect, that is, the focal length range for clear imaging is relatively small, so it can have a very good background blurring effect, suitable for portrait and still photography, which is shown in Fig. 10.3.

Fig. 10.3 Shallow depth-of-field case

Small aperture lens has a large depth of field, which allows the shooting of the foreground and background targets to be clear, which is shown in Fig. 10.4.

Fig. 10.4 Case of large depth of field

In short, the size of the aperture, the focal length of the lens, and the distance of the shot are important factors which affect the depth of field, and their relationship with the depth of field is concluded as follows:

1. The larger the aperture (the smaller the aperture value *f*), the shallower the depth of field.
2. The longer the zoom magnification of the lens (focal length), the shallower the depth of field.
3. The closer distance of the subject, the shallower the depth of field.

When we want to highlight a target subject, we can choose a larger aperture and focal length to keep the lens as far away from the background as possible and as close to the subject as possible, so as to obtain excellent background blurring effect, highlighting the subject to be performed.

10.1.1.2 Depth-of-Field Editing Effect

To achieve a great depth-of-field blurring effect, the lens needs to have a large aperture, which is not a problem for lenses such as SLR, but cell phone cameras are limited by the size of the sensor and cannot directly capture blurring effects comparable to SLR lenses, so we need to use post-processing tools to edit the simulation of blurring effect.

Photoshop is a commonly used tool, but it is not suitable for the general public due to the high cost of use. Currently, there are also some

classic post-production depth-of-field adjustment tools in mobile terminal. Take Focos as an example, it can take photos first and then focus, allowing for arbitrary editing of depth of field, and it has been recommended by AppStore for 2 years. It uses the iPhone's multi-lens design to get 3D models when shooting, and later to edit and synthesize depth-of-field effects, and add simulated light source lighting. The upgraded version supports depth-of-field simulation for any photo, not limited to photos taken with iPhone, Fig. 10.5 shows the effect of using Focos to process a photo.

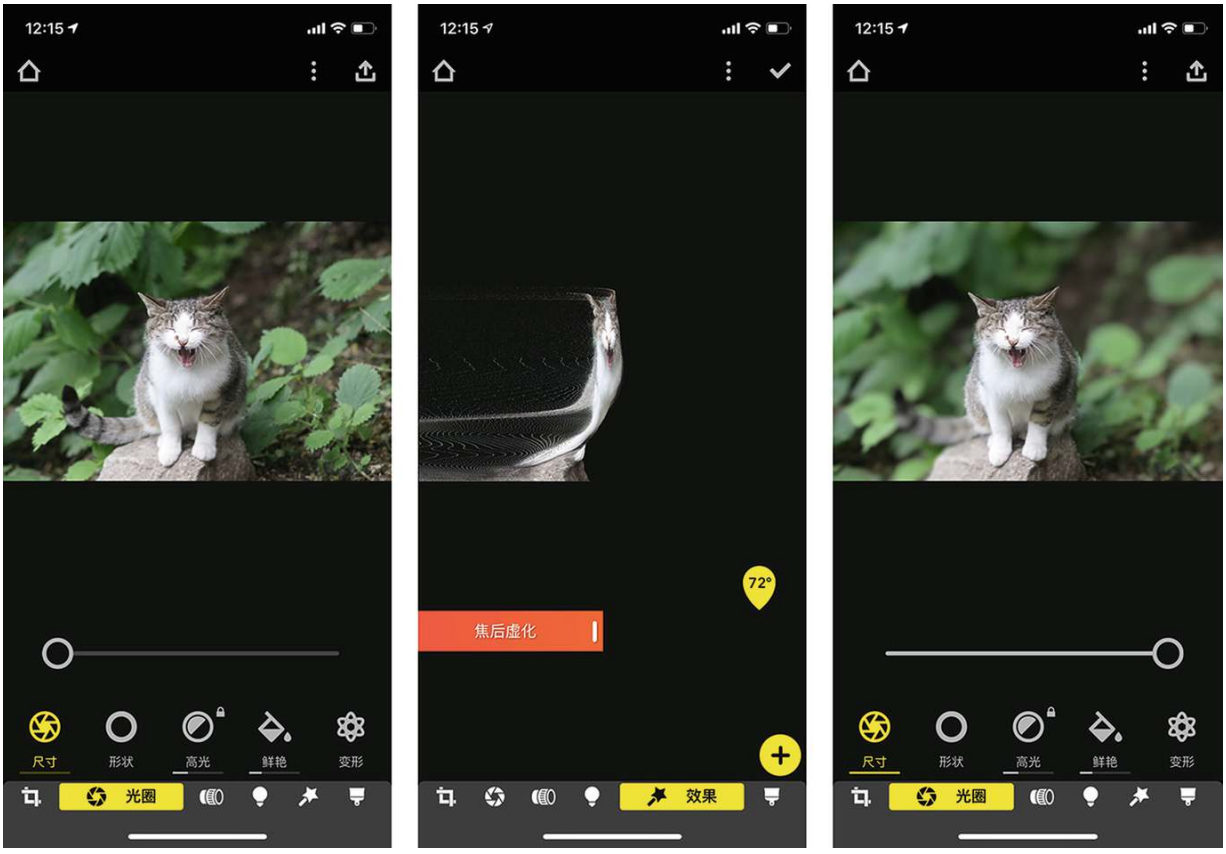


Fig. 10.5 Focos Post Bokeh tool

From left to right, the first picture is the original picture, the second picture is the effect of depth-of-field estimation, and the third picture is the result of adjusting the aperture to the maximum, i.e., effect of editing the depth of field. Today, based on advanced machine learning algorithms, we can also simulate the effect of a large aperture directly in the phone camera.

For photographic images, when we modify the depth of pixels, we actually want to change the depth-of-field effect of the front background, so we refer to depth-of-field editing here, more commonly known as depth editing, and we use depth-of-field editing to refer to it uniformly as follows.

10.1.2 Image Depth-of-Field Editing Framework

Next we present a typical generative adversarial network-based depth-of-field editing framework, RefocusGAN [1], which consists of two steps, deblurring and focusing, to achieve refocusing and thus editing of the foreground depth of field, both of which are done based on conditional generative adversarial networks.

The first step is deblurring: a focus-complete image (Near-Focus) and its focus response estimate (Focus Measure Response) are used as input. Near-Focus means that targets close to the camera are in focus and targets far from the camera are blurred, and the focus response estimate is essentially an edge detection of the subject target. The two are stitched together and input to the generator to estimate a clear focus map (Generated In-Focus), i.e., all targets are in focus, and the whole framework process is shown in Fig. 10.6.

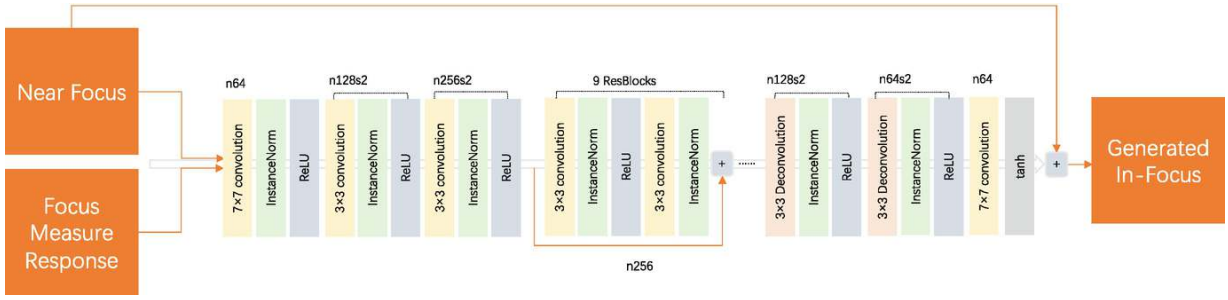


Fig. 10.6 RefocusGAN deblurring

The second step is the refocusing: By the original Near-Focus image and the Generated In-Focus image stitched into the generator to generate the far-field focus image, so as to simulate the editing of the depth of field, to achieve the near target and far target refocusing, the whole framework process is shown in Fig. 10.7.

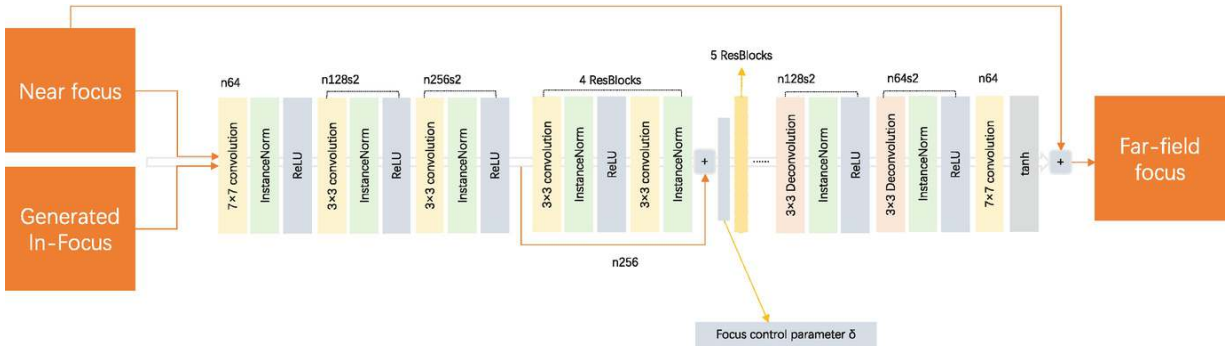


Fig. 10.7 RefocusGAN refocusing

Figures 10.8 show the Near-Focus image, the Ground-Truth In-Focus image, the Ground-Truth Far-Focus image, the Focus Measure Response image, the generated full-focus image, generated Far-Focus image.

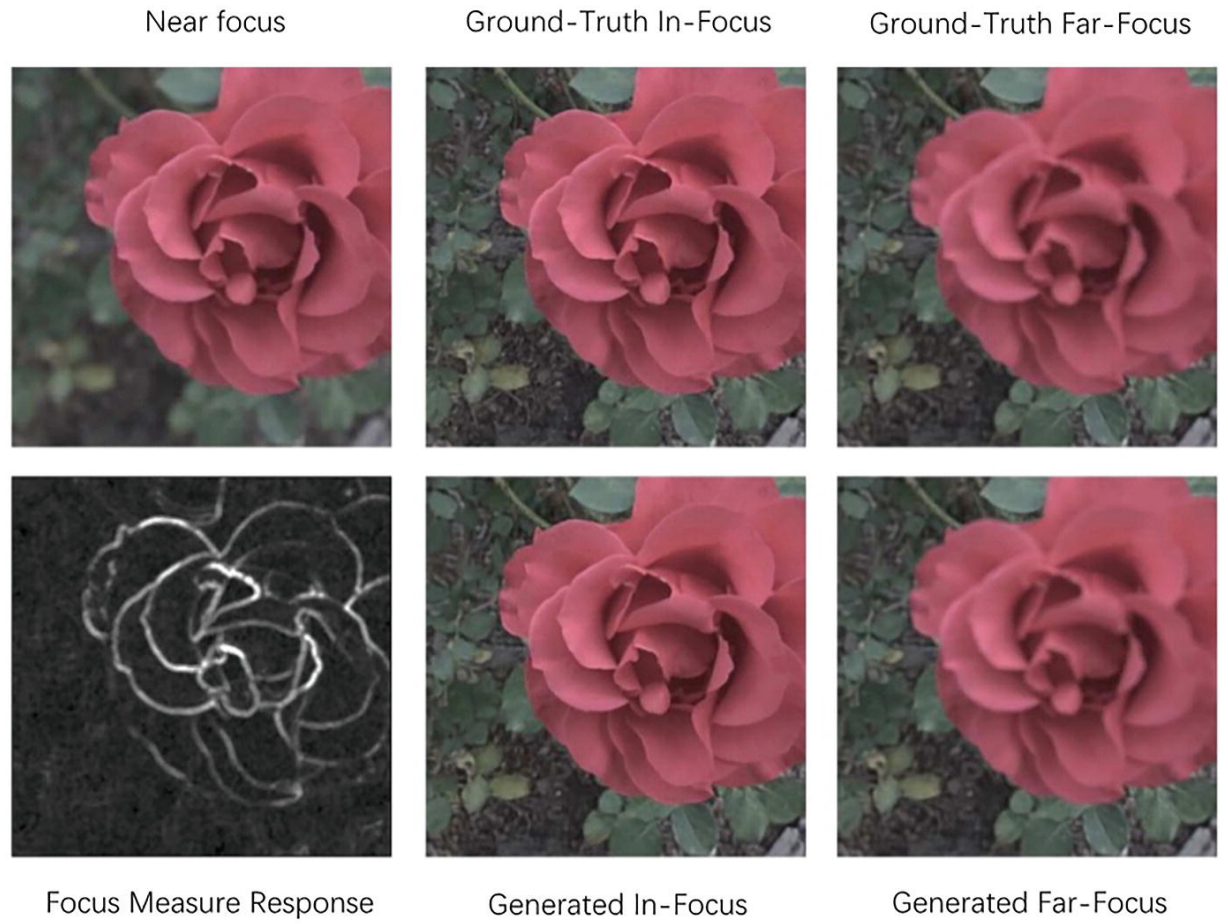


Fig. 10.8 RefocusGAN results

The same model structure is used for the above two steps, and the optimization objective uses both adversarial loss and perceptual loss. The mutual switching of the distant target focus and the near target focus can be controlled interactively by the focus control parameters.

10.2 Image Fusion

The so-called image fusion, i.e., to achieve the fusion of two images, or to insert the target of one image into a new background image, and the face swapping algorithm we introduced in Chap. 7 can essentially be classified as image fusion.

10.2.1 Image Fusion Problem

If we only obtain the target region that needs to be fused into one image, and parameters such as transparency are not available, if the replacement is done directly in the original image, the color of the part will often be significantly different from the surrounding area, and there is no smooth transition at the edges, at this time the region needs to be transformed under the constraints of color and gradient, where the classical method is Poisson fusion [2], which is to solve the following problem:

$$\min \iint_{\Omega} \left| \nabla f - v \right|^2 \text{ with } f \Big| \partial\Omega = f^* \mid \partial\Omega \tag{10.1}$$

If we want to fuse the source image B on the target image A, let f denote the fused result image C, f^* denote the target image A, v denote the gradient of the source image B, ∇f denotes the gradient of f namely the first-order gradient of the result image C, Ω denotes the region to be fused, and $\partial\Omega$ represents the edge part of the fused region.

The meaning of Eq. (10.1) is to make the gradient of the resultant image C in the fusion part closest to the gradient of the source image B in the fusion part when the edges of the target image A remain unchanged. Thus, the color and gradient of the source image B will change during the fusion process in order to blend into a natural one with the target image A.

Figure 10.9 shows some examples of image fusion generated by using the Poisson fusion method, where columns 1 and 2 are fused to obtain column 3.

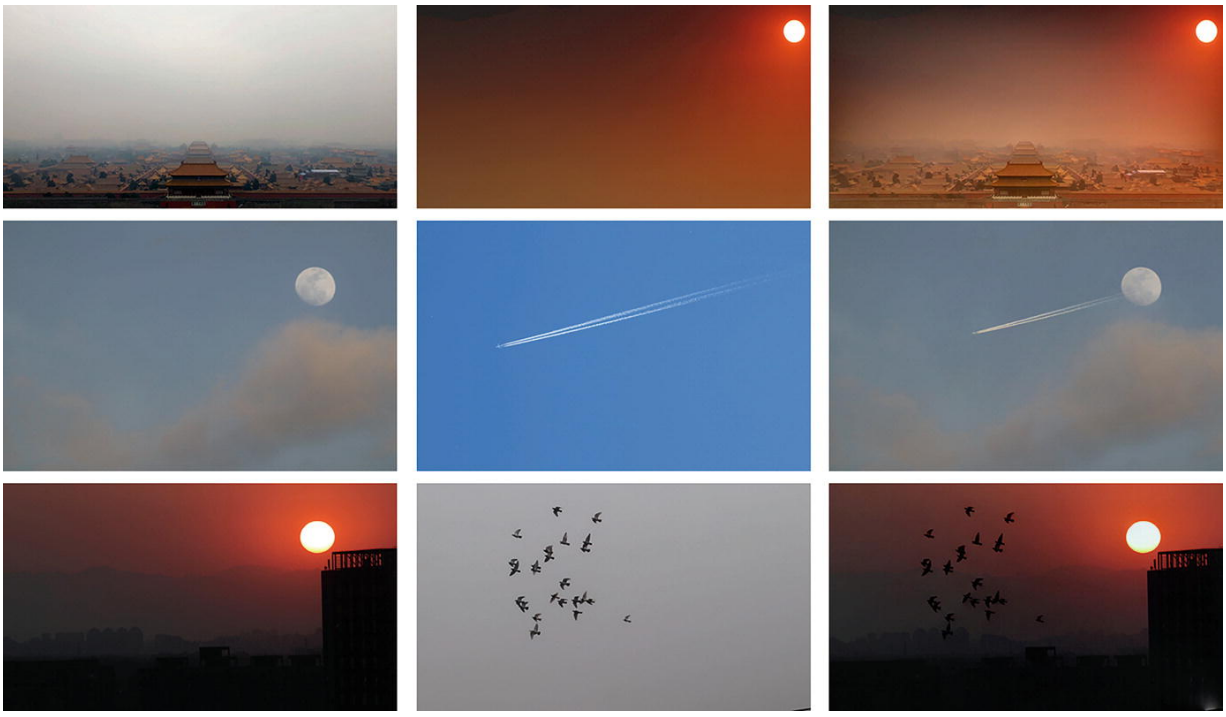


Fig. 10.9 Poisson fusion case

10.2.2 GAN-Based Image Fusion Framework

With the development of technologies such as deep learning and generative adversarial networks, current deep learning-based image fusion is also proposed by researchers, represented by GP-GAN (Gaussian-Poisson GAN) [3]. GP-GAN is a GAN-based image fusion network that combines GAN model and Poisson fusion, and the framework flow is shown in Fig. 10.10.

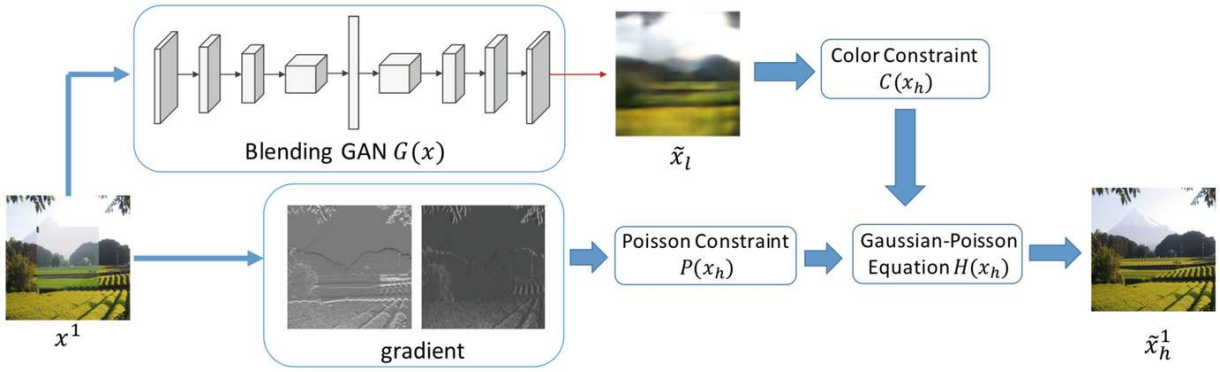


Fig. 10.10 GP-GAN framework

The GP-GAN framework consists of two main parts: the Blending GAN and the Gaussian-Poisson Equation.

Blending GAN is an encoder-decoder structure that uses the L2 distance as the reconstruction loss for the output, and then adds the adversarial loss as the optimization target. This structure can be used as a color constraint to make the generated image more realistic and natural, resulting in a relatively blurred low precision output map.

Since the original images (src) and the target image (dst) used for fusion in this framework are the same scene under different shooting conditions, the authors used the target image as the reconstructed true value. When this condition is not satisfied, the training is performed using an unsupervised approach.

The Gaussian-Poisson Equation (GPE) is a pyramidal high-resolution structure that acts as a gradient constraint to further increase the resolution of an image with realistic texture details.

The optimization objectives include Poisson fusion objectives and color constraints, which is as follows:

$$H(x_h) = P(x_h) + \beta C(x_h) \tag{10.2}$$

Where $P(x_h)$ is the standard Poisson equation, whose goal is to make the generated map have the same high-frequency signal as the original synthetic map. $C(x_h)$ is the color consistency constraint, whose objective is to make the generated map have the same low-frequency signal as the original synthetic map. Equation (10.3) is written in discrete form as follows:

(10.3)

$$H(x_h) = \|u - Lx_h\|_2^2 + \lambda\|x_l - Gx_h\|_2^2$$

where L is the Laplace operator, G is the Gaussian operator, and u is the dispersion of vector field v, x_h is the image for which the solution is requested, and x_l is the input low-resolution image, the v is taken from the original image and the target image, respectively, according to whether it is a fused region, defined as follows:

$$v(i,j) = \begin{cases} \nabla x_{\text{src}}, if\ \text{mask}(i,j) = 1 \\ \nabla x_{\text{dst}}, if\ \text{mask}(i,j) = 0 \end{cases} \tag{10.4}$$

Equation (10.4) has an analytical solution, and the specific solution is solved by continuously increasing the resolution according to the pyramid model, and the x_h obtained from the previous level serves as the x_l resolution of the next level.

Compared with methods such as Poisson fusion, GP-GAN can use the generative ability of generative models to fuse more complex regions. Current image fusion techniques based on deep learning models are under development, and readers who are interested in it can stay tuned.

10.3 Interactive Image Editing

To re-edit and recreate images, technicians often need long-term professional training and have a high threshold. Although tools like Photoshop have been popular for many years, they are still limited to professional and in-depth enthusiasts. Thus, developing a foolproof, minimalist interactive image editing tool is meaningful for the public who have image editing needs but do not have time to learn professional skills, and in this section we introduce the interactive image editing framework.

10.3.1 Interactive Image Editing Problem

The so-called interactive image editing means that users can use less work to create complex images, such as drawing real RGB images based on simple color strokes, as shown in Fig. 10.11.

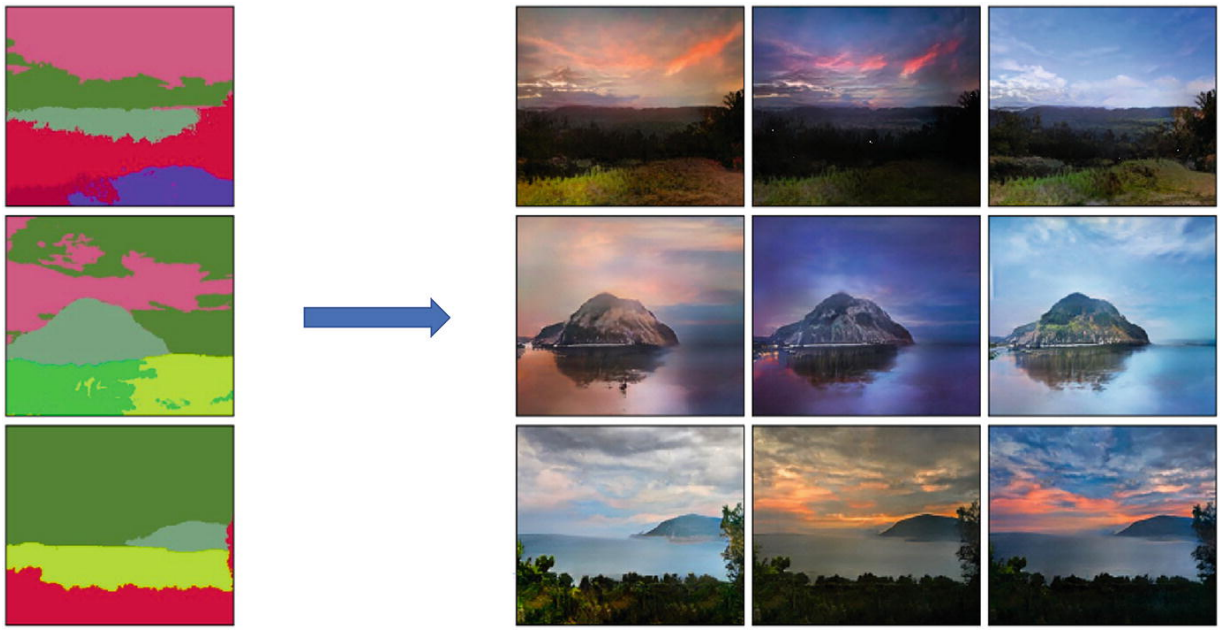


Fig. 10.11 Generating complex, high-quality images based on simple inputs

The first column represents the simple color strokes for drawing, which only requires the user to have a basic knowledge of life and does not require deep drawing skills. The three columns on the right represent the high-quality results generated by the algorithm. We can change the semantics of the different colors of the strokes to create works which are rich in content and style.

10.3.2 GAN-Based Interactive Image Editing Framework

Next, we introduce the current representative interactive image editing framework.

SPADE [4] is an image translation framework which is based on Spatially Adaptive Normalization Layer, and the input of a semantically segmented mask map can output a synthetic image with a high degree of realism.

The general image generation GAN stacks convolutional, normalization, and nonlinear layers together to form a generative model. The SPADE framework points out that the existing normalization layer usually transforms the input data of the layer into a distribution with mean 0 and standard deviation 1, and if the input label values are the same, all the data will become 0 (mean 0), the result is that the semantic information of the input semantic label map

is often “erased,” so that the generated image will have a large gray result or wrong pattern, which affects the authenticity of the generated image.

To solve this problem, the SPADE framework uses a new normalization layer called Spatially Adaptive Normalization Layer (SPADE layer). This layer learns the input semantic label image *m* by convolutional layers and learns two sets of transformation parameters corresponding to the normalization parameters of BN, i.e., γ and β, which are matrices with dimensions equal to the size of image and are no longer vector coefficients, which is shown in Fig. 10.12.

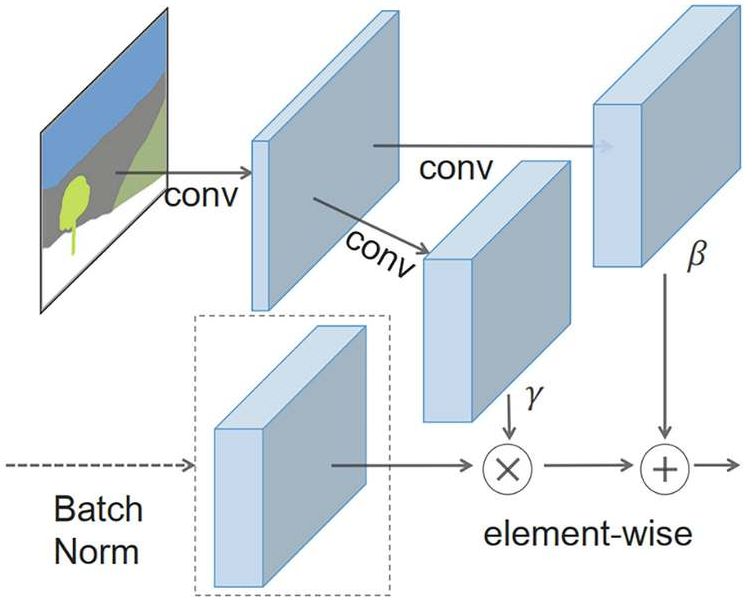


Fig. 10.12 Schematic of SPADE layer

The equation for the SPADE layer is shown in Eq. (10.5):

$$\mathrm{BN} = \gamma^{i}_{c,y,x}(m)\frac{h^{i}_{n,c,y,x}}{\sigma^{i}_{c}} + \beta^{i}_{c,y,x}(m) \tag{10.5}$$

where *y*, *x* denotes the spatial location of the image, *c* denotes the channel, μ and σ denote the mean and standard deviation of each channel feature map, and each channel will be computed separately.

The generative model structure uses Pix2pixHD to incorporate the SPADE normalization layer into the generative model, thus allowing the semantic information to be effectively preserved and passed

throughout the generative model. Since the SPADE layer already learns the input semantic annotation image information well, the encoding part of the generative model is no longer needed, only the decoding part needs to be kept, and the input of the network can be directly set as noisy data.

The overall architecture of SPADE is shown in Fig. 10.13.

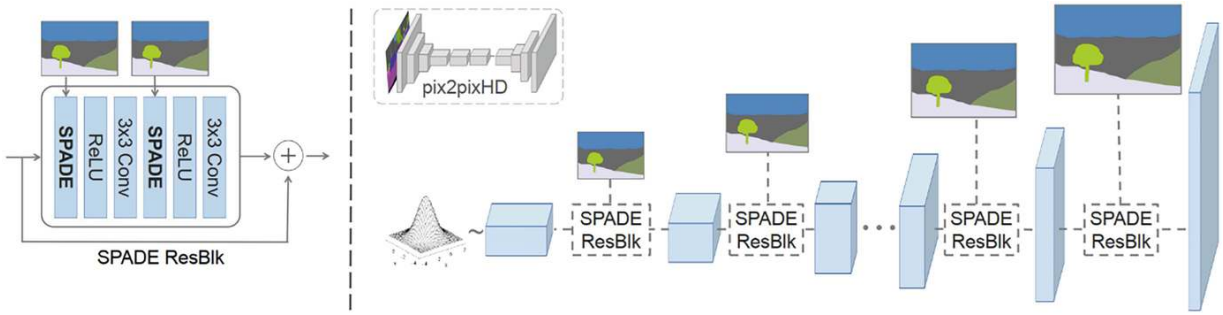


Fig. 10.13 SPADE model architecture

When we want to generate different styles of images, we can also add an encoder in the front end to encode the style of a specific style of image as the input noise data, thus generating a landscape image of the specified style, which are shown in Fig. 10.14.

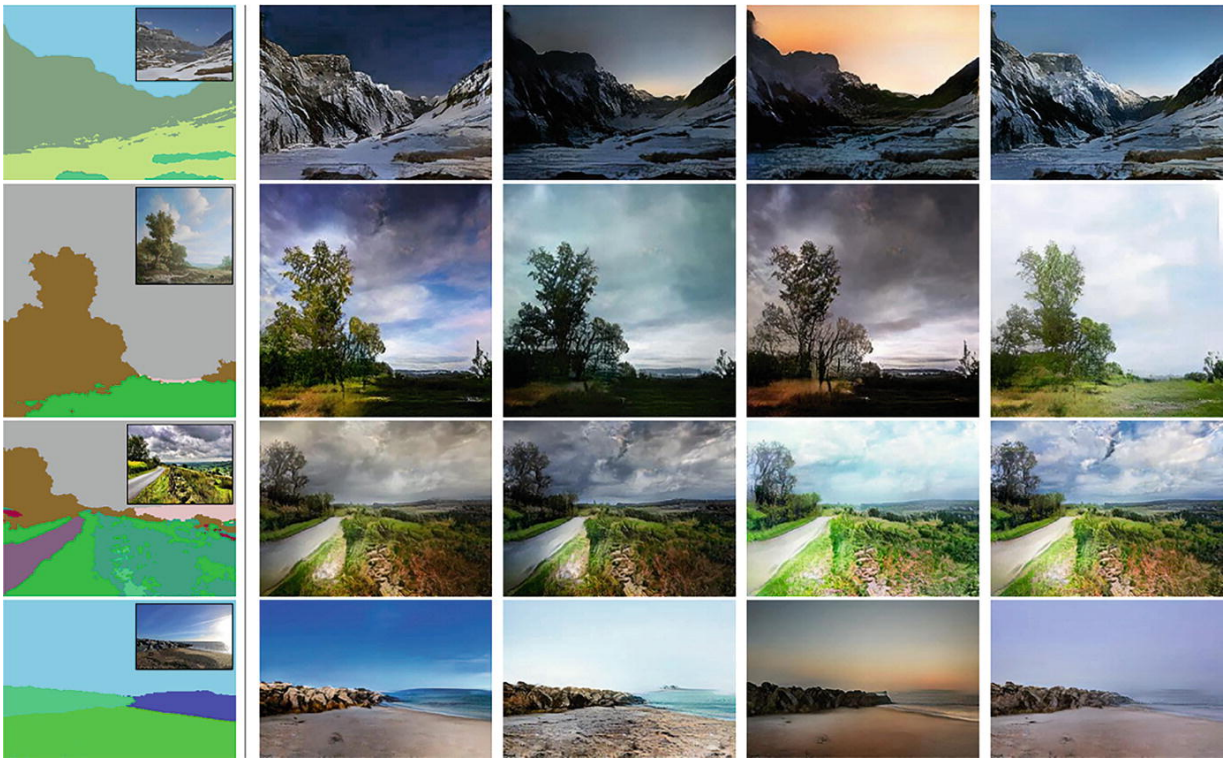


Fig. 10.14 Specifying style output

Column 1 in Fig. 10.14 is the input mask with the specified style map in its upper right corner, and columns 2–5 are the generated result maps under the input control of sunny day, evening, evening sunset, and daytime style maps, respectively.

10.4 Outlook

In this chapter, we introduce several problems of image editing. However, no practice has been implemented, mainly because the direction covered in this chapter is not mature at present, and there is still a big gap from the algorithm landing. Although deep editing and image fusion is a relatively niche direction, it has great application prospects in cell phone camera image processing, and is a noteworthy technical direction. Interactive image editing, on the other hand, has considerable application prospects in content creation and games, and readers can follow the relevant content on their own.

References

1. Sakurikar P, Mehta I, Balasubramanian V N, et al. Refocusgan: Scene refocusing using a single image [C]//Proceedings of the European Conference on Computer Vision (ECCV). 2018: 497–512.

2. Pérez P, Gangnet M, Blake A. Poisson image editing [M]//ACM SIGGRAPH 2003 Papers. 2003: 313–318.

3. Wu H, Zheng S, Zhang J, et al. Gp-gan: Towards realistic high-resolution image blending [C]//Proceedings of the 27th ACM International Conference on Multimedia. 2019: 2487–2495.

4. Park T, Liu M Y, Wang T C, et al. Semantic image synthesis with spatially-adaptive normalization[C]//Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2019: 2337–2346.

P. Long, X. Guo, *Generative Adversarial Network*
https://doi.org/10.1007/978-981-96-9404-4_11

11. Adversarial Attack

Peng Long[1] ✉ and Xiaozhou Guo[2]
(1) Beijing YouSan Educational Technology, Beijing, China
(2) • China Electronics Technology Group Corporation No. 54 Research Institute, Shijiazhuang, China

Abstract
This chapter introduces the relevant concepts of adversarial attacks and presents related examples. It elaborates in detail on the principles of three common types of attack algorithms, namely optimization-based, gradient-based, and generation-based algorithms. Moreover, it explains the principles of common defense algorithms such as defense distillation, adversarial training, denoising networks, and adversarial sample detectors. Subsequently, in terms of the application of GAN attacks, three GAN-based adversarial sample generation models are introduced, including Perceptual-Sensitive GAN, Natural GAN, and AdvGAN. Regarding the defense of GANs, taking APE-GAN as an example, its defensive effect is demonstrated. Finally, a practical introduction to the widely used open-source tool AdvBox is provided.

Keywords Adversarial attack – Adversarial sample – GAN attack – GAN defense

This chapter introduces the content of adversarial attacks and the applications of GAN. In the first part, we first introduce the basic concepts of adversarial attacks and show related examples, then introduce the common adversarial attack algorithms in detail, and also describe the defense algorithms. In the second part, we introduce three GAN-based adversarial sample generation models, including Perceptual-Sensitive GAN, Natural GAN, and AdvGAN, and in the third part, we take APEGAN as an example to show its effectiveness in defense. Finally, we provide a practical illustration of the more applied AdvBox open source tool.

- Section 11.1. Adversarial attack and defense algorithms

- Section 11.2. GAN-based adversarial sample generation
- Section 11.3. GAN-based defense against attacks
- Section 11.4. Adversarial Toolkit AdvBox

11.1 Adversarial Attack and Defense Algorithm

11.1.1 Adversarial Attack

In deep learning, neural networks are very susceptible to perturbations or noise, and often the noise or perturbations are so slight that they are not detectable to the human eye, but they greatly affect the classification results or even make obvious errors, as shown in Fig. 11.1.

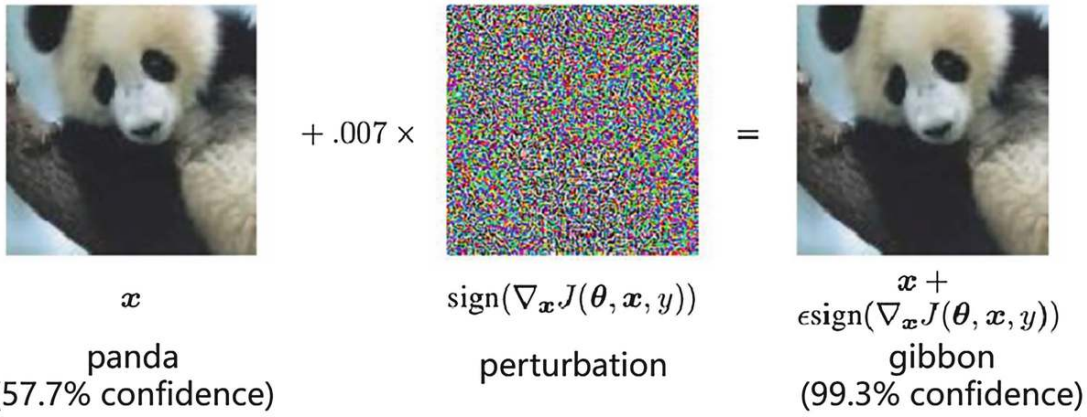


Fig. 11.1 Schematic diagram of adversarial attack

Classifier classifies the input sample as panda with 57.7% confidence, and when noise is added, there is no significant change in the image before and after perturbation from the human eye's visual point of view, but the classifier completely confirms that the scrambled sample is a gibbon, which is obviously inconsistent with common sense. In general, we call such perturbed samples as adversarial samples. Adversarial samples are commonly found in machine learning models, which have been theoretically elaborated from various perspectives, such as finite robustness, boundary skew, and linearity assumptions, and will not be expanded in this book. In practice, adversarial samples can be computationally obtained by certain algorithms and also exist directly in the real world, as shown in the following figure (Fig. 11.2):

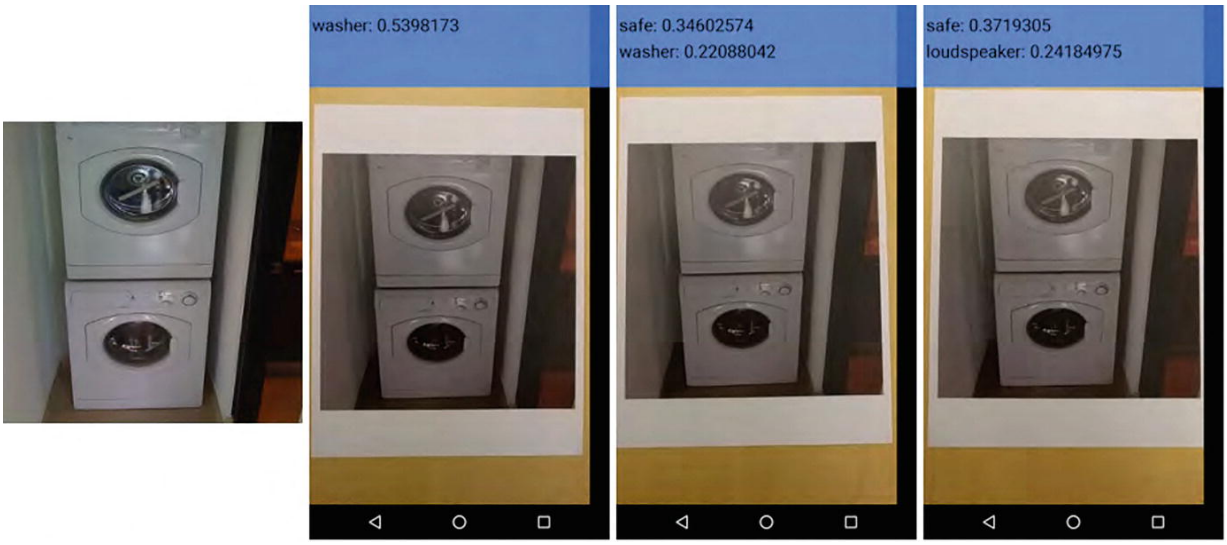


Fig. 11.2 Real-world adversarial sample

The image on the left is used as a training set to train the classifier, and the sample is fed into the photo software and printed out to get a new sample, which shows that the classifier does not always give correct results due to small differences in shooting angle and distance.

The study of adversarial samples has very important security implications. Real-world AI models have been deployed on a large scale, and attackers are able to attack machine learning models with carefully designed adversarial samples to make obvious errors, thus generating security threats, i.e., adversarial attacks. For example, an attacker can tamper with the design of traffic sign road signs, causing self-driving cars to recognize them as other commands and cause traffic accidents; an attacker can deceive the face recognition security system by facial disguise and perform intrusions. Adversarial samples can also play a positive role, for example, by providing novel perspectives to study the weaknesses and blind spots of deep neural networks and improve their robustness; they can also be used to regenerate the image data uploaded by users through adversarial sample generation techniques to avoid unscrupulous elements from capturing and analyzing that user's data through neural network-based image recognition systems, thus protecting user privacy.

The adversarial attacks can be divided into targeted and untargeted attacks, where targeted attacks are performed by adding noise to the sample x so that the perturbed samples are classified by the classifier C into a specified new class l. The aim of the untargeted attack is to make the classifier wrong, as long as the perturbed sample is not classified into its original right labeled class. From the degree of knowledge of the model, the adversarial attack can be further divided into black-box attack and white-box attack, where white-

box attack means that the attacker knows the structure, weight parameters, hyperparameters, and other information of the model and can construct a complete copy of the model, while in black-box attack, the attacker can only obtain the output results of the model, such as confidence, label vector and other information, which is obviously much more difficult than the white-box attack.

11.1.2 Commonattack Algorithm

This subsection will introduce the general attack algorithms. The goal of the adversarial attack algorithm is to generate adversarial samples that cause errors in the classifier. Based on the knowledge of the classifier model, we classify the attack algorithms into white-box attack algorithms and black-box attack algorithms, where white-box attack algorithms can be classified into three major categories based on the principle: optimization-based attack algorithms, gradient-based attack algorithms, and generation-based attack algorithms.

11.1.2.1 Optimization-Based Attack Algorithm

The optimization-based attack algorithm generates an adversarial sample by finding the changed value of a pixel through a global optimization search. We can correspondingly describe it as a mathematical problem. For a clean normalized image I (with pixel values ranging from [0, 1]), we compute a perturbation or noise δ. If the value range of the new image $x + \delta$ remains within [0, 1]) and the discriminator $C(x + \delta)$ can classify it into a new target category l, then δ should be made as small as possible to be barely perceptible to humans. Collectively, we can formulate the following optimization problem:

$$\min d(x, x + \delta)$$

$$s.t. C(x + \delta) = l;$$

$$x + \delta \in [0, 1]^m \tag{11.1}$$

Where $d(x, x + \delta)$ denotes the distance between two images. The first equation constraint of this problem $C(x + \delta) = l$ is difficult to handle, and the CW (Carlini & Wagner) algorithm converts it to an inequality constraint: the

$$f(x + \delta) \leq 0 \tag{11.2}$$

Where $f(\cdot)$ is set by human, for example,

$$f(x') = \max\left(\max_{i \neq t}\left(F(x')_i\right) - F(x')_t, -k\right) \tag{11.3}$$

Where $F(\cdot)$ is softmax function, and using the p-norm to represent the image distance. Then the original problem can be further transformed into

$$\min \| \delta \|_p + c \cdot f(x+\delta) \; s.t. \; x + \delta \in [0,1]^m \tag{11.4}$$

We call $x + \delta \in [0, 1]^m$ is the box constraint. For the box constraint, CW [1] gives three ways to deal with it:

1. Data pruning, $x + \delta$ is trimmed to fall within the range of each iteration of the calculation;
2. Modify $f(x + \delta)$ in the objective function to $f(\min(\max(f(x + \delta), 0), 1))$, so that transform the box constraint into a "soft" constraint in the objective function;
3. Introduce the variable w, and make $x + \delta = (\tanh(w) + 1)/2$, so that satisfy the box constraint.

Deepfool [2] is another typical optimization-based attack algorithm that aims to find the minimum perturbation that can cause the classifier to misclassify. As shown in Fig. 11.3 left, in a linear binary classification problem, the original sample can be formed into a new sample by adding a perturbation and the classifier will give a misclassification result for that sample because it is on the classification plane, where the minimum perturbation is the distance from the original sample to the classification plane. Extending the above idea to a multi-classification task, as shown on the right of Fig. 11.3, the distance from the original sample to each classification plane can be calculated and the smallest of these distances can be chosen as the perturbation, and the new sample can be misclassified by the classifier. In fact, we are basically facing a nonlinear multi-classification problem. The Deepfool algorithm first computes the approximate classification hyperplane in each iteration and then computes the nearest perturbation to the classification surface and adds the perturbation to the sample to get a new sample.

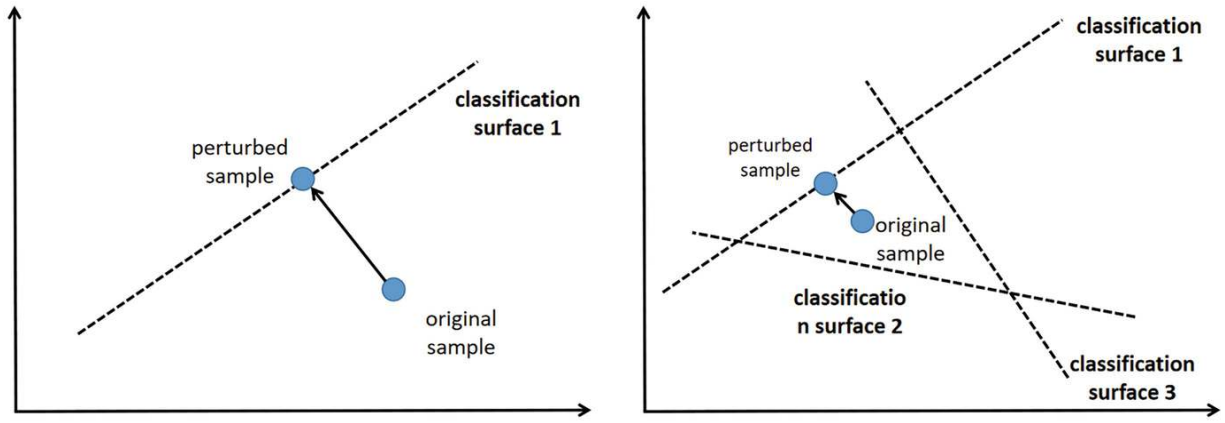


Fig. 11.3 Schematic diagram of Deepfool

11.1.2.2 Gradient-Based Attack Algorithm

The basic idea of the gradient-based attack algorithm is to find the sample that maximizes the loss function, and the most direct approach is gradient ascent since the direction of the gradient indicates the direction in which the value of the loss function rises fastest. fGSM [3] (Fast Gradient Sign Method), a single-step attack algorithm, is calculated as follows:

$$x' = x + \epsilon \cdot \text{sign}\left(\nabla_x L\left(f(x), \hat{y}\right)\right) \tag{11.5}$$

Where sign(·) is the sign function, L is the loss function, and $\hat{y}$ is the label of sample x. The FGSM attack algorithm does not attack the target class, and when the classifier and label are given, the loss function $L(x)$ is a function of the sample x. We want to reach a sample point with a relatively large loss function because a larger loss function means that the sample should not be classified into category $\hat{y}$.

The BIM (Basic Iterative Method) [4] attack algorithm uses stepwise iterations to compute the adversarial samples based on the FGSM, which also tends to work better. First make $x_0 = x$, and then iterating sequentially is:

$$x_{t+1} = \text{clip}\left(x_t + \alpha \cdot \text{sign}\left(\nabla_{x_t} L\left(f(x_t), \hat{y}\right)\right)\right) \tag{11.6}$$

In the reverse FGSM algorithm, we first predict the least-likely label $\check{y}$ for an original sample x. A smaller loss function $L(f(x), \check{y})$ means that the classification result provided by the classifier is closer to $\check{y}$. Naturally, we can mimic FGSM using the gradient descent method:

$$x' = x - \alpha \cdot \text{sign}\left(\nabla_x L(f(x), \check{y})\right) \tag{11.7}$$

Same as BIM, the ILCM (Iterative Least-likely Class Method) attack algorithm generalizes the above method to a stepwise iterative algorithm, i.e.,

$$x_{t+1} = \text{clip}\left(x_t - \alpha \cdot \text{ sign } \left(\nabla_{x_t} L\left(f(x_t), \hat{y}\right)\right)\right) \tag{11.8}$$

11.1.2.3 Generation-Based Attack Algorithm

Generation-based adversarial algorithms refer to the direct use of neural networks to directly generate adversarial samples, mainly represented by ATN (Adversarial Transformation Networks) [5] and GAN. In ATN, the neural network $g_\theta(x)$ takes the sample x as input and outputs the perturbed sample x', where θ represents the parameters of the network. The training objective function for $g(x)$ is defined as:

$$\min_\theta \sum_{x_i} \beta L_A(g_\theta(x_i), x_i) + L_B\left(C(g_\theta(x_i)), r\left(\hat{y}, l\right)\right) \tag{11.9}$$

Here, L_A is a loss function based on similarity, which describes the distance between the perturbed sample $g_\theta(x_i)$ and the original sample. L_B is a loss function based on the label vector. $C(g_\theta(x_i)$ represents the output label vector of the perturbed sample, and $r\left(\hat{y}, l\right)$ indicates the reranking of labels, which maximizes the value for class l in the label vector through some mapping. Regarding the form of the reordering function Reranking, we will not expand on this further. Finally, the L2 norm can be used to measure between L_A and L_B, where β represents the weight between the loss functions. The content on generating adversarial samples using GAN will be elaborated in Sect. 11.2.

11.1.2.4 Black-Box Attack Algorithm

The above algorithms are white-box attack algorithms, in practice white-box attack algorithms have achieved impressive attack results, black-box attack algorithms due to the attack on the black-box model, little knowledge of the model although the difficulty increases but the effect is also satisfactory; its typical representative algorithms are: single-pixel attack, UPSET, ANGRI, RP2, and other algorithms.

The general attack algorithms mostly perform some transformations on the pixel points of the whole image, and the perturbation is limited by the total transformation size. The single-pixel attack algorithm limits the number of changeable pixel points by changing the value of only one pixel point without limiting its transformation size, so as to achieve the purpose of misclassification of the transformed image. The single-pixel attack uses a

heuristic evolutionary algorithm to search for an adversarial sample. The evolutionary algorithm is less likely to fall into local optima, does not require gradient information, and is easy to implement. First, a set of five-dimensional perturbation vectors containing pixel positions and RGB pixel values is constructed, and new perturbation vectors are generated continuously during iteration according to specific rules, which compete according to the adaptation function to maintain the population size and finally find the solution that satisfies the termination condition.

In the UPSET [6] algorithm, the optimization problem is

$$\max\left(\min\left(sU(l)+x,1\right),-1\right) \tag{11.10}$$

The perturbation is obtained by multiplying a scalar *s* and *U*(*l*), where the input to the neural network *U* is the target class *l*, and the output consists of *N* (where *N* is the total number of classes) perturbation noises. When perturbation noise is added to images not belonging to class *l*, the perturbed image can be classified by the classifier as the target class *l*. The UPSET algorithm generates universal, image-agnostic perturbation noise, while the ANGRI algorithm produces perturbations specific to the original image. The neural network *A* in ANGRI takes the original sample *x* and the target class *l* as input and outputs the perturbed sample. The loss function used during the training of networks *U* or *A* includes two components: the first component is classification accuracy, and the second component is reconstruction error loss. Both UPSET and ANGRI algorithms can achieve relatively high success rates on comparably simple datasets.

RP2 (Robust Physical Perturbations) [7] generates an adversarial sample for a specific region. The original images of different physical situations are first acquired, then preprocessed so as to determine the location of the target to be attacked in the original image, to obtain the attack mask, and to use an iterative attack algorithm to attack the mask region of the original image, so that the perturbations are concentrated in the most vulnerable target region, maximizing the reduction of the recognition performance of the neural network model and generating the adversarial samples. Finally, the adversarial samples need to be printed out and pasted onto real objects to generate the adversarial objects. Compared with other methods, RP2 is more realistic, but requires the collection of multiple data in different physical situations.

11.1.3 Common Defense Algorithm

In this section, we introduce several commonly used methods for defending against attacks. Like adversarial attack algorithms, there is a wide variety of adversarial attack defense methods, and we select four representative

categories of methods to introduce them so that readers have a basic understanding of defense algorithms.

11.1.3.1 Distillation Defense Algorithm

The first defense method is the distillation defense algorithm [8]. In order to improve the robustness of a trained classifier, it is necessary to retrain a new classifier in order to invalidate the adversarial samples, but retraining the classifier requires some computational resources. Distillation network is a feasible solution, which is a training method proposed by Hinton to migrate knowledge from complex networks to simple networks, reducing the complexity of the model without decreasing the generalizability, and finally enhancing the robustness of the classifier to adversarial samples.

The concept of distillation networks is very simple and effective. First, a classifier C_1 is trained using sample x and label $\hat{y}$. The final layer of the classifier typically uses a softmax layer, producing an output category probability z after the samples pass through the classifier, where z is an N-dimensional vector (N is the number of categories). This category knowledge is then distilled to obtain new category probabilities p.

$$p_i = \frac{\exp\left(z_i/T\right)}{\sum_j \exp\left(z_j/T\right)} \tag{11.11}$$

Among them, T represents the temperature, which controls the degree of distillation. A new classifier C_2 is trained using sample x and the new class probability p. During training, the temperature T is generally set to a relatively high value to reduce the gradient of the new classifier (here, the gradient refers to the gradient of classifier input with respect to the input), making the classifier smoother and thereby decreasing its sensitivity to adversarial perturbations. This is essentially a method of gradient masking. During forward inference, it is necessary to reset the temperature T to 1. Experiments show that as the temperature increases, the success rate of adversarial attacks decreases, while the accuracy of the classifier shows no significant decline, and the gradient of the classifier diminishes, resulting in an overall improvement in the model's robustness. This indicates that the distillation defense algorithm has a certain degree of effectiveness, but its limitation lies in the fact that it remains a static defense algorithm, incapable of preventing the existence of adversarial samples.

11.1.3.2 Adversarial Training

The second defense method is adversarial training [9]. The basic idea of adversarial training is that when training the classifier, the training samples do not only include the original samples, but some of the adversarial samples are constructed and added to the training set, then as the classifier is trained iteratively, not only the accuracy of the original samples can be increased, but also the robustness of the classifier can be improved. The adversarial training can be written uniformly as the following min-max problem:

$$\min_{\theta} \mathbb{E}_x \left[\max_{||\delta|| \leq \epsilon} L\big(C_\theta(x+\delta), \hat{y}\big) \right] \tag{11.12}$$

Here, θ represents the weight parameters of classifier C. The minimization process indicates that the fundamental objective of adversarial training is to train the classifier to improve accuracy, while the maximization process indicates that to enhance robustness, samples with the maximum loss function value within the ϵ-neighborhood of sample x are chosen to replace the original sample x for training. Due to space limitations, this section will only introduce the most basic Fast Gradient Method (FGM) algorithm, which calculates the perturbation δ using the gradient ascent method:

$$\delta = \epsilon \frac{\nabla_x L\big(C_\theta(x+\delta), \hat{y}\big)}{\big\| \nabla_x L\big(C_\theta(x+\delta), \hat{y}\big) \big\|} \text{ or } \delta = \epsilon \ \text{sign} \left(\nabla_x L\big(C_\theta(x+\delta), \hat{y}\big) \right) \tag{11.13}$$

The PGD (Projected Gradient Descent) algorithm is an improved version of the FGM algorithm, which decomposes the one-step iterative process into multiple steps to complete, initializing the sample x_0, the iteration is calculated as follows:

$$x_{t+1} = x_t + \frac{\nabla_x L\big(C_\theta(x+\delta), \hat{y}\big)}{\big\| \nabla_x L\big(C_\theta(x+\delta), \hat{y}\big) \big\|} \tag{11.14}$$

In the iteration, if the perturbation exceeds a certain range then it needs to be mapped to the neighboring domain of ϵ.

Additionally, defense methods based on integrated adversarial training have also received significant attention. The basic idea is to use adversarial training to generate adversarial samples by using multiple pre-trained models, and add all these adversarial samples to the original training set to train the classifier, which is to upgrade the original one-to-one training model to a one-to-many model, using more types of adversarial samples to the original

dataset The data augmentation is performed by using more types of adversarial samples to the original dataset.

Adversarial training belongs to a brute force approach to enhance the robustness of classifiers. Experimental results show that general adversarial training is more suitable for defense against white-box attacks, while integrated adversarial training shows stronger robustness against black-box models. Adversarial training has some limitations, because of its mechanism, enhancing robustness relies on high-intensity adversarial samples, requires the classifier to have sufficient expressive power, and adversarial training still cannot avoid dropping the emergence of new adversarial samples.

11.1.3.3 Denoising Network

The third defense method is to use a denoising network [10, 11]. The process of adversarial attack is to add noise perturbation to a pure sample to form an adversarial sample to attack the classifier, while the defense method using denoising network reverses the process completely, when the classifier faces an adversarial sample, it first rebuilds the adversarial sample into a pure sample by denoising network, and then feeds the pure sample to the classifier, as shown in Fig. 11.4, which greatly improves the classifier's ability to resist. The ability of the classifier to resist attacks is greatly improved.

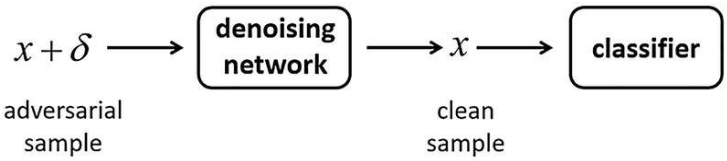


Fig. 11.4 Defense methods based on denoising network

There have been a large number of attempts on denoising network, such as the use of self-encoders to reconstruct the adversarial samples and the use of convolutional neural networks to compress and reduce images. This section briefly introduces the high-level feature-guided denoiser algorithm as a representative case. The most direct loss function when training the denoiser is based on pixel value error, $|x - x'|$, which aims to keep the noisy image as close as possible to the original image. The drawback of this method is the decline in the accuracy of the classifier because the denoiser cannot completely eliminate noise; the residual disturbances propagate through the classifier, causing disruptions in abstract features and leading to a decrease in classifier accuracy. The High-Level Representation Guided Denoiser (HGD) uses the differences in abstract features from the last few layers of the classifier as the loss function to train the denoiser, thereby avoiding the issue of disturbances amplifying layer by layer. The loss function is defined as

$|f(x) - f(x')|$, where $f(\cdot)$ represents the output of the final layers of the classifier. Experimental results indicate that HGD demonstrates strong robustness against both white-box and black-box attacks. Defense methods based on denoising networks exhibit significant robustness when the attacker is unaware of the existence of the denoising network; however, this method remains susceptible to white-box attacks.

11.1.3.4 Adversarial Sample Detector

The fourth defense method is to use the detector [12]. When the classifier is confronted with an unknown sample (unclassified sample), it first uses the detector to determine whether the sample is an adversarial sample, and if it is an adversarial sample, it refuses to perform classification on it, and if it is a non-adversarial sample, it is input to the classifier normally for classification, as shown in Fig. 11.5.

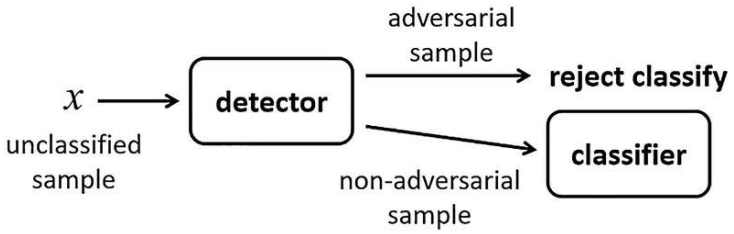


Fig. 11.5 Adversarial sample detector

There are various detector-based adversarial defense algorithms, and we list several methods here: (1) extract the ReLU layer output as the input features of the adversarial detector and detect the adversarial sample by RBF-SVM classifier; (2) train a simple binary classification network to detect the perturbation in the input sample; (3) use CNN to convolve the input image and then achieve the detection of the adversarial sample based on the statistical features; (4) add an adversarial sample class directly to the classifier; (5) perform spatial smoothing on the input image and reduce its color depth to obtain a new feature-compressed image, compare the two images, and consider it as an adversarial sample if the difference is large; (6) learn the stream shape of the pure sample, and consider it as an adversarial sample if the sample to be detected is far from the stream shape; (7) train the classifier using the minimized reverse cross-entropy and use the threshold strategy as the detector.

11.2 Adversarial Sample Generation Based on GAN

GAN itself possesses a very strong ability to generate images and text; therefore, it can also generate adversarial samples. Perceptual-Sensitive GAN, Natural GAN, and AdvGAN are among the representatives. This section will introduce these three models separately; the former generates adversarial image samples through generative adversarial blocks, while the latter two directly produce adversarial samples that are naturally real.

11.2.1 Perceptual-Sensitive GAN

General confrontation samples are synthesized from pure images with perturbation noise, and the resulting perturbed image is not different from the original image to the naked eye. Firstly, we introduce a novel confrontation sample, which is synthesized from the original image with a small image block, which is a very common phenomenon in the real world, as shown in Fig. 11.6. For example, if there are some small stickers or graffiti on the street traffic signs, we can interpret the traffic sign image as a pure image and the stickers and other contents as the aforementioned image blocks. This is a form of disturbance, and the traffic sign image with stickers can be understood as a disturbed sample. Practical evidence shows that these disturbed image samples can also serve as adversarial samples, causing the classifier to misclassify the types.

Fig. 11.6 Adding small image blocks generate adversarial examples

In this section, an algorithm that uses GAN to generate the aforementioned adversarial samples PSGAN (Perceptual-Sensitive GAN) [13] is introduced. Compared with other methods for generating adversarial blocks, PSGAN focuses not only on the attack capability of the adversarial blocks but also on the perceptual sensitivity, i.e., it wants to generate natural-looking adversarial blocks that are associated with the image context, as shown in Fig. 11.7. Also as an adversarial sample, the adversarial block on the right has better results in terms of perceptual sensitivity than the adversarial block on the left, and its spatial location and semantics are more natural.

Fig. 11.7 Sensitivity comparison of image perception

The basic framework of PSGAN includes: attention model M, generator G, discriminator D, and attack target F (i.e., classifier), as shown in Fig. 11.8.

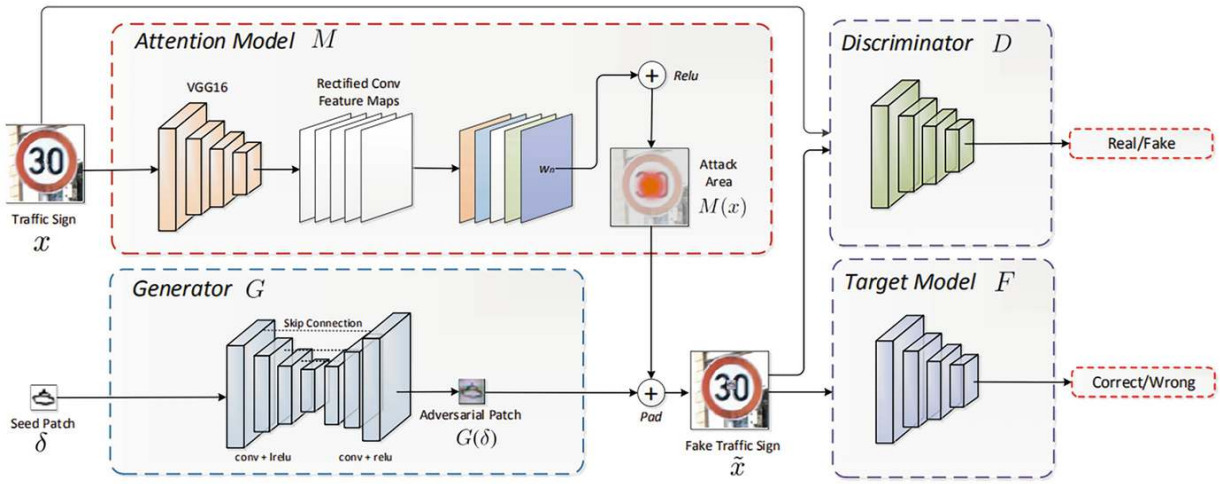


Fig. 11.8 Network of PSGAN

To enhance the authenticity and coherence of generated adversarial blocks, PSGAN has designed a block-to-block generation process. In this process, the input is a seed block δ and a sample image x, while the output is an adversarial block that is similar to the seed block and coherent with the sample image. Specifically, the generator is responsible for generating the adversarial block $G(\delta)$ that is coherent with the sample image x, while the discriminator is tasked with distinguishing between the sample image and the image containing the adversarial block. Essentially, the discriminator is learning the divergence distance between the two classes of images, and the generator is

trained to minimize this distance to increase their similarity. Furthermore, to model spatial location sensitivity, PSGAN incorporates an attention model *M* into the block-to-block generation process. The attention model *M* aims to capture the classification-sensitive areas *M*(*x*) of the target relative to the sample *x*, and adding adversarial blocks in these sensitive areas helps to enhance the attack effectiveness. In summary, the method of generating adversarial samples in PSGAN can be expressed as follows:

$$\widetilde{x} = x +_{M(x)} G(\delta) \tag{11.15}$$

The loss function of PSGAN *L* consists of three components: L_{GAN}, L_{patch} and L_{adv}. In order to generate adversarial blocks with better visual fidelity, PSGAN constructs a loss function L_{GAN} borrowed from the standard GAN

$$L_{GAN}(G, D) = \mathbb{E}_x[\log D(\delta, x)] + \mathbb{E}_{x,z}\left[\log\left(1 - D(\delta, x +_{M(x)} G(\delta))\right)\right] \tag{11.16}$$

Where *z* is the input noise. Experiments have shown that when the noise *z* is removed and the seed block δ is retained, PSGAN can still perform effectively.

In order to make the generated adversarial block more perceptually relevant to the input image context and to make the adversarial block and the seed block as similar as possible, a loss function L_{patch} is introduced:

$$L_{patch}(\delta) = \mathbb{E}_{\delta}[\| \delta - G(\delta) \|_2] \tag{11.17}$$

In addition, in order to generate adversarial samples to attack model *F*, it is necessary to include the loss function of the adversarial attack L_{adv}:

$$L_{adv}(G, F) = \mathbb{E}_{x,\delta}[\log p_F(\widetilde{x})] \tag{11.18}$$

This loss function minimizes the probability output values of adversarial examples after being processed by the classifier *F*, thereby enhancing the attack capability. In summary, the training loss function of PSGAN is:

$$\min_G \max_D L_{GAN} + \lambda L_{patch} + \gamma L_{adv} \tag{11.19}$$

Among them, λ and γ are both positive real numbers used to balance various loss components. In PSGAN, the internal structure of generator *G* is similar to that of an autoencoder, with each layer employing a convolutional kernel size of 2, utilizing layer normalization, and utilizing the LeakyReLU activation function. The number of convolutional kernels in each layer is 16,

32, 64, 128, 64, 32, 16, and 3, respectively. The discriminator *D* utilizes the same convolutional layers as generator *G*, with the number of convolutional kernels in each layer being 64, 128, 256, and 512, followed by a fully connected layer using the sigmoid activation function, ultimately outputting a one-dimensional scalar value. In the attention model *M*, PSGAN employs the Grad-CAM algorithm to compute the attention map of the classifier, thereby identifying attack-sensitive areas. Specifically, the sample *x* is fed into the classical VGG16 model; the feature map *A* obtained after the last convolutional layer has *k* channels, with each channel's feature map size being $u \times v$. The class probability y^c corresponding to the labels is calculated by summing the gradients of the feature map *A* for each channel, denoted as α_k^c:

$$\alpha_k^c = \frac{1}{u \times v} \sum_{i=1}^{u} \sum_{j=1}^{v} \frac{\partial y^c}{\partial A_{ij}^k} \tag{11.20}$$

α_k^c denotes the importance weight of each channel, and the final weighted calculation of the attention map yields:

$$L_{Grad-CAM}^c = \mathrm{ReLU}\left(\sum_k \alpha_k^c A^k \right) \tag{11.21}$$

If the attention map does not match the resolution of the input image, the size of the attention map needs to be resized. In the attention map, important regions are highlighted, and PSGAN pastes the adversarial blocks here can greatly increase the attack efficiency.

Similar to the general GAN training process, PSGAN trains the discriminator *k* times before training the generator once in each iteration. During each iteration for training the discriminator, *N* training images $\{x_1, \ldots, x_N\}$ and *N* seed blocks $\{\delta_1, \ldots, \delta_N\}$ are first sampled. Then, the generator *G* generates *N* adversarial blocks $\{G(\delta_1), \ldots, G(\delta_N)\}$. The Grad-CAM algorithm is used to compute the attention map for each training image. The *N* adversarial blocks are pasted onto each training image, resulting in $N \times N$ adversarial samples $\left\{ x_i +_{M(x_i)} \delta_j \middle| i, j = 1, \ldots, N \right\}$. The discriminator *D* is trained according to the loss function L_{GAN}. When training the generator, *N* training images $\{x_1, \ldots, x_N\}$ and *N* seed blocks $\{\delta_1, \ldots, \delta_N\}$ are again sampled, and the attention map is computed for each training image using Grad-CAM. The generator *G* is then trained based on the loss function $L_{GAN} + \lambda L_{patch} + \gamma L_{adv}$. Other training details can be found in the original paper.

PSGAN uses an adversarial generative model to generate adversarial blocks, uses attention graphs to find sensitive region locations, and combines the two to construct perceptually natural adversarial samples, achieving more valuable attack effects. In addition, the experimental results show that PSGAN achieves equally good attack effects in semi-white-box and black-box models.

11.2.2 Natural GAN

Most of the time, we not only wish to obtain adversarial examples that can cause the classifier to err, but we also hope that the adversarial example $\widetilde{x}$ is as similar as possible to the original sample *x*. This will make the generated adversarial examples appear more natural. As shown in Fig. 11.9, the leftmost image is the correctly classified original sample. The middle image (generated using FGSM) and the right image (generated using GAN) are both adversarial examples incorrectly classified as the digit "2". However, the right image clearly appears more natural than the middle image, making such adversarial examples more aligned with reality and possessing greater research value.

Fig. 11.9 Comparison of the Naturalness of Adversarial Samples

Optimization-based and gradient-based adversarial sample generation algorithms often produce images that lack coherence. In contrast, GAN possess unparalleled advantages in image generation tasks. The NaturalGAN introduced in this section is a typical example of generating adversarial samples using GAN. NaturalGAN [14] comprises four main components: discriminator *D*, generator *G*, inverter *I*, and targeted attack *F* (the classifier). The generator *G* takes Gaussian noise *z* as input and outputs generated sample $x = G(z)$. Conversely, the inverter *I* receives sample *x* as input and outputs the representation vector $z = I(x)$. The standard GAN discriminator takes sample input and produces a scalar output. The targeted attack F serves as a classifier, which in the case of NaturalGAN is a black-box model.

In NaturalGAN, discriminator *D* and generator *G* have the same roles as in WGAN, training *D* and *G* using a set of unlabeled image sample sets $\{x^{(1)}, \ldots, x^{(N)}\}$ that enable the generator to learn the latent distribution $p_{data}(x)$ of the

training sample set. In order to find the disturbance noise, the most direct idea is to search for adversarial samples near the generated sample $G(z)$, and finally form a given noise z, generate a clean image x by the generator, and find the process of finding the adversarial sample $\widetilde{x}$ near x. However, NaturalGAN does not search in the sample space, but in the space of the representation vector z, that is, for any sample x, first use the inverter I to map it to the representation vector z, then look for the representation $\tilde{z}$ of the adversarial sample near z, and finally use the generator to obtain the adversarial sample $\widetilde{x} = G(\tilde{z})$, so that the adversarial sample is more realistic, and the semantic association between the noise disturbance and the context is stronger.

In Natural GAN, the discriminator D and generator G are first trained, and the objective functions is

$$\min_{G}\max_{D} \ \mathbb{E}_{x\sim p_{data}}[D(x)] - \mathbb{E}_{z\sim p_z}[D(x)] \tag{11.21}$$

Then the inverter I is trained, and the loss function consists of two parts, the first part is the reconstruction error, so that the sample x is close to the sample obtained after passing through the inverter and the generator, and the second part is the distribution distance, that is, hope the distribution of z is the same as that of $I(G(z))$:

$$\min_{I} \mathbb{E}_{x\sim p_{data}}[\| \ x - G(I(x)) \ \|] + \lambda \mathbb{E}_{z\sim p_z}[L(z, I(G(z)))] \tag{11.22}$$

where λ is the hyperparameter of the balance loss function. NaturalGAN hopes that the mapping of the inverter and the generator will be reversed to avoid large errors in the conversion of the sample vector x and the representation vector z. After the neural network is trained, we need to look for the representation vector $\tilde{z}$ near the representation vector z. In the face of a black-box model that needs to be attacked, a well-designed search algorithm can be used to find it, and two search algorithms designed for this purpose will be described in this section.

In the first search algorithm, we gradually expand the search area from near to far. The length of the search radius change Δr and the number of samples N in each round of search are fixed. Firstly, N samples are randomly generated in the area with the representation vector z of the original sample x as the center of the circle, and the N samples are sent into the generator and the target model F in turn, and whether the classification results given by F are consistent with the categories of the original samples are compared, and if they are consistent, it indicates that there are no adversarial samples and continue to expand to a farther range for searching, and the search radius

range is [Δr, 2Δr]. Note that the search is no longer repeated for the [0, Δr] region that has already been searched. The process continues until the end of a round appears as an adversarial sample representation vector, here we select the nearest $\tilde{z}$ as the final result, and its corresponding $\tilde{x}$ is the adversarial sample, as shown in Fig. 11.10.

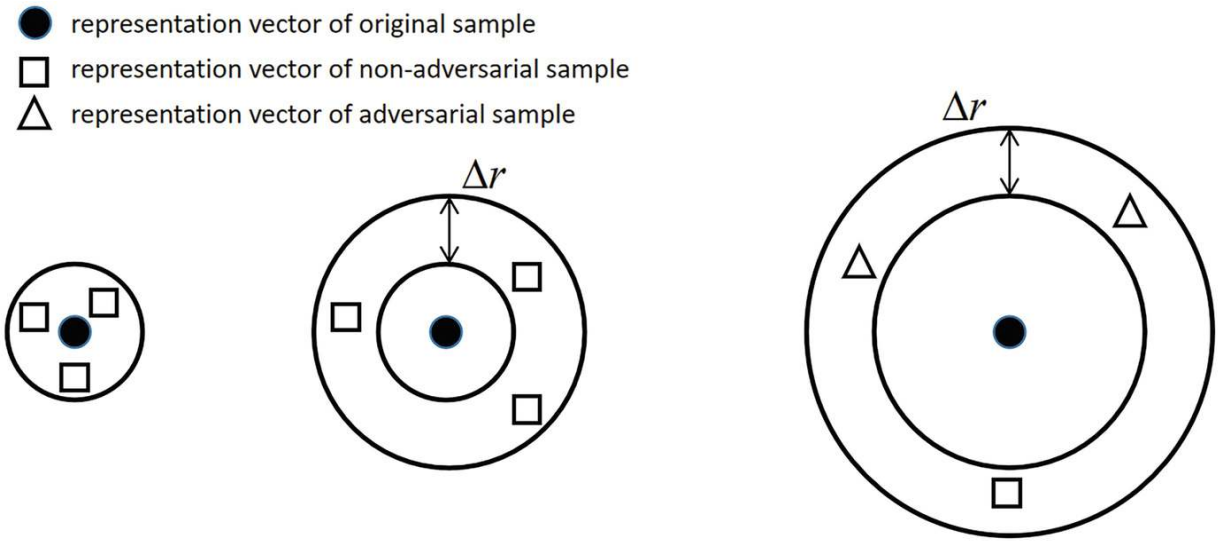


Fig. 11.10 Schematic diagram of the first search algorithm

The first search algorithm is intuitive and simple, but the search efficiency is relatively low, here we introduce the second search algorithm, which introduces a dichotomy and increases the search efficiency. Similar to the first algorithm, the change length of the search radius Δr of each round of search is fixed, the upper bound of the search radius r, the number of samples in each round of search is N, and the dichotomy method is used to conduct a rough search. First in the range of [0, r], if there are no adversarial samples, the search radius is halved, and the search range is narrowed to [$r/2$, r], and if there are still no adversarial samples, the range is further reduced to [$3r/4$, r] …… until the adversarial samples are found at the end of a round, calculate the nearest distance from the representation vectors of the adversarial samples to the representation vectors of the original samples, and use this distance as the upper bound of the search radius r to start a fine iterative search, as shown in Fig. 11.11. Fine search adopts an outside-in search strategy, which first searches in the range of [r − Δr, r] and narrows it down to [r − 2Δr, r] if there are no adversarial samples…… until the end of a certain round, an adversarial sample appears, and the nearest representation vector is selected as the adversarial sample representation vector $\tilde{z}$, and its corresponding $\tilde{x}$ is the adversarial sample, as shown in Fig. 11.12.

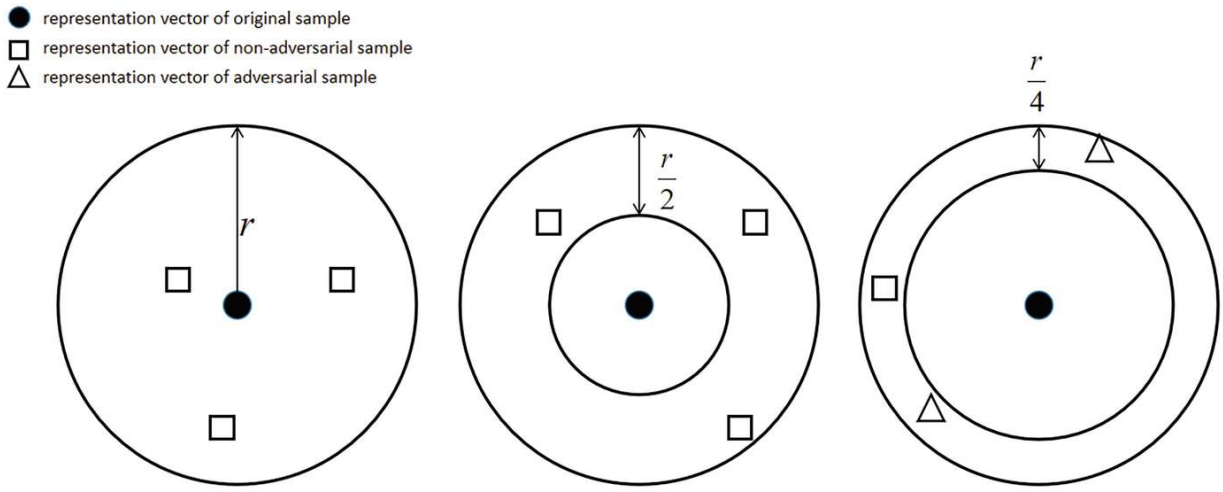


Fig. 11.11 Dichotomous search schematic

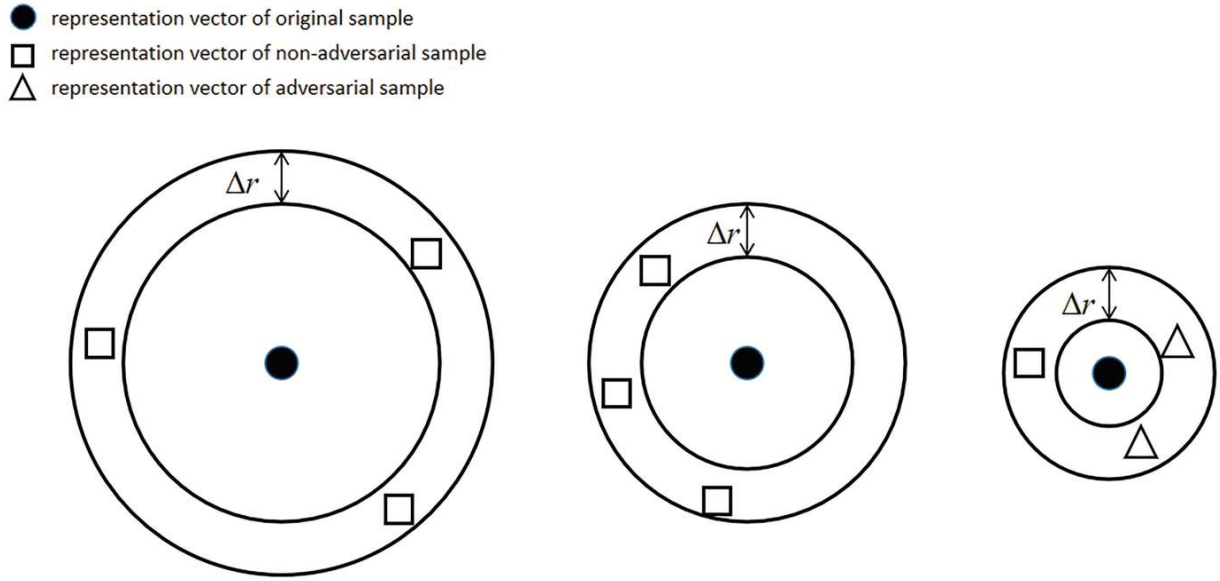


Fig. 11.12 Schematic diagram of the search from outside to inside

Experiments show that NaturalGAN can generate adversarial samples not only in the visual domain for images, but also in natural language processing for textual implication and machine translation, which indicates that the approach of finding adversarial samples in the representation vector space is not only more efficient in attacking, but also can generate more natural attacking samples.

11.2.3 AdvGAN

AdvGAN [15] is a generative adversarial network model that generates adversarial samples to white-box attack the classifier of target model *F*, and its basic idea is to generate adversarial samples similar to the original image

through GAN, with relatively small perturbation, which can deceive the target model *F*.

The basic structure of AdvGAN is shown in Fig. 11.13. The generator G receives the original sample *x* as input and outputs interference noise *G*(*x*) of the same size as the original sample. By adding the original sample x and the noise *G*(*x*), we obtain the adversarial sample $x' = x + G(x)$, which can then be sent to the target model *F* to carry out the attack.

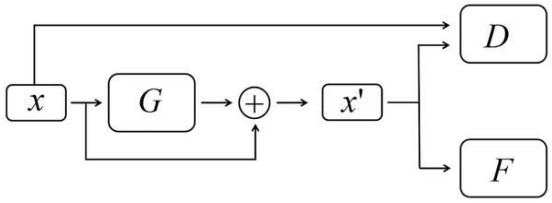


Fig. 11.13 AdvGAN structure diagram

To ensure that the adversarial sample x' and the original sample *x* are as similar as possible, AdvGAN employs a discriminator *D* to facilitate this function. Similar to the principles of standard GAN, the discriminator *D* continuously seeks the differences between *x* and x', striving to distinguish between them, while the generator *G* continuously improves the quality of the generated noise, making it difficult for the discriminator to determine whether the sample is the original one or the synthesized adversarial sample. For this purpose, we use the objective function L_{GAN} for training:

$$L_{GAN} = \mathbb{E}_x[\log D(x)] + \mathbb{E}_x[\log (1 - D(x + G(x)))] \tag{11.23}$$

In order for the synthesized adversarial samples to deceive the target model *F*, it is necessary to train the generator *G* such that the output category of the adversarial sample x' after passing through the target model *F* is not the original label $\hat{y}$ of *x* or the designated target attack category *l*. For the former, the loss function should be set to maximize the distance between the predicted distribution $F(x')$ and the label $\hat{y}$; for the latter, it is necessary to minimize the distance between the predicted distribution $F(x')$ and the target category *l*. Defining $\text{loss}_F(x, l)$ as the loss function used to train the target model *F* with sample *x* and label *l*, the objective function L_{adv} for this project is:

$$L_{adv} = \mathbb{E}_x[\text{los s}_F(x + G(x), l)] \tag{11.24}$$

In order to avoid excessive amplitude of the perturbation noise *G*(*x*), a penalty term is applied when the perturbation noise exceeds a certain

threshold c. The hinge loss function can be used to construct the objective function L_{hingle} as follows:

$$L_{hingle} = \mathbb{E}_x[\max\ (0, \parallel G(x)\ \parallel_2 -c)] \tag{11.25}$$

In summary, the objective function of AdvGANL is:

$$\min_G \max_D\ L_{adv} + \alpha L_{GAN} + \beta L_{hingle} \tag{11.26}$$

Among them, α and β are used to balance the objective functions. It should be noted that the aforementioned model is suitable for white-box attacks but not for black-box attacks, as training using the L_{adv} in the objective function requires knowledge of the structure and parameters of the target model F; otherwise, gradient backpropagation cannot be performed. AdvGAN introduces network distillation technology, enabling it to attack black-box model F. The basic idea is to train a neural network f using sample x and the output $F(x)$ after passing through the black box, aiming to make $f(x)$ as close as possible to $F(x)$. For instance, the cross-entropy loss function between $F(x)$ and $f(x)$ can be minimized. We use the distilled network $f(x)$ instead of the black-box model $F(x)$ for generating training adversarial samples, and after training AdvGAN, we attack the black-box model $F(x)$.

Considering the discrepancies between the distilled network $f(x)$ and the black-box model $F(x)$, which cannot be quantified, AdvGAN employs an iterative distillation method to further enhance the attack effectiveness. Iterative distillation consists of two steps that continually iterate back and forth: Step 1 is to fix the distilled network $f_{i-1}(x)$ and train the discriminator G_i and generator D_i:

$$G_i, D_i = \arg\min_G \max_D\ L_{adv} + \alpha L_{GAN} + \beta L_{hingle} \tag{11.27}$$

Step 2 is to fix the discriminator G_i and generator D_i, and train the distilled network f_i:

$$f_i = \arg\min_f\ \mathbb{E}_x H(f(x), F(x)) + \mathbb{E}_x H\Bigg(f(x + G_i(x)), F(x + G_i(x)) \tag{11.28}$$

Where $H(\cdot, \cdot)$ denotes the cross-entropy loss function of the two distributions, it can be seen that the objective function of iterative distillation requires that the distillation network can learn not only the knowledge of the original

samples, but also the knowledge of the synthetic adversarial samples. The experimental results show that AdvGAN can not only handle white-box attacks, but it can also achieve very high success rate on semi-white-box attacks and black-box attacks.

11.3 GAN-Based Adversarial Attack Defense

The previous section introduced three models that utilize GAN for adversarial attacks, demonstrating their role in such attacks. This section will present models that use GAN for adversarial attack defense, primarily including APEGAN and DefenseGAN, which achieve perturbation removal from different perspectives to accomplish their defensive functions.

11.3.1 APEGAN

In the previous section, we introduced three models for adversarial attacks using GAN, and in this section we present models for adversarial attack defense using GAN.

The theory and a large body of practical results indicate that GAN can not only map noise z to image samples x through the generator G, but can also establish mappings from image to image (e.g., in style transfer tasks). Therefore, it is natural to consider using the generator of GAN to establish a mapping from adversarial sample images x' to clean sample images x to achieve the effect of denoising and removing disturbances. When faced with an adversarial sample x', it is first fed into the generator G to obtain the clean sample $x = G(x')$, which is then input into the target model F. As long as the generator G is trained sufficiently well, it can eliminate the noise in the image, thus preventing adversarial attacks and enhancing the robustness of the model.

For the training sample image set $\{x^{(1)}, x^{(2)}, \ldots, x^{(N)}\}$, algorithms such as FGSM are used to generate the corresponding adversarial samples $\left\{x_{adv}^{(1)}, x_{adv}^{(2)}, \ldots, x_{adv}^{(N)}\right\}$. To avoid confusion, this section uses x_{adv} to denote the adversarial samples generated by the attack algorithm. The task of the generator G is to denoise the samples, while the task of the discriminator is to measure the difference between $G(x_{adv})$ and x. By employing adversarial training, the goal is to ensure that $G(x_{adv})$ approaches x, which is entirely consistent with standard GAN, except that the noise z is replaced with the adversarial sample x_{adv}. The optimization problem for training the discriminator D is as follows (Fig. 11.14):

(11.29)

$$\max_{D} \mathbb{E}_x[\log D(x)] - \mathbb{E}_{x_{adv}}[\log (1 - D(G(x_{adv})))]$$

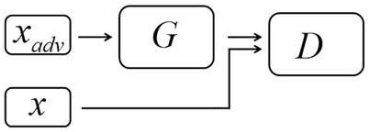


Fig. 11.14 Diagram of APEGAN [16]

The loss function of generator G consists of two components: the content loss function L_{con} and the adversarial loss function L_{adv}. The content loss function is used to measure the pixel-level differences between the clean image x and $G(x_{adv})$. It employs the Mean Squared Error (MSE) for measurement, namely:

$$L_{con} = \mathbb{E}_x \left[\frac{1}{WH} \sum_{i=1,j=1}^{W,H} \left(x_{i,j} - G(x_{adv})_{i,j} \right)^2 \right] \tag{11.30}$$

Among them, W and H represent the length and width of the image, respectively. The adversarial loss function L_{adv} remains consistent with the loss function in GAN, namely:

$$L_{adv} = \mathbb{E}_{x_{adv}}[\log (1 - D(G(x_{adv})))] \tag{11.31}$$

The optimization objective of generator G is:

$$\min_{G} \alpha L_{con} + \beta L_{adv} \tag{11.32}$$

Among them, α and β are the weights that balance the two loss terms. After training is completed, all samples must first pass through the generator for denoising before being sent to the target model. The idea behind APEGAN is both simple and effective, and it can be utilized for black-box attacks, demonstrating favorable performance on the MNIST, CIFAR10, and ImageNet datasets.

11.3.2 DefenseGAN

DefenseGAN [17] also utilizes GANs to achieve noise reduction from another perspective. For a GAN model that has already been trained using training samples (it is recommended to use the overall better-performing WGAN model), we only use its generator G, which can map noise to sample images

$x = G(z)$, and can be used to learn the distribution of clean images. When faced with adversarial samples, we generate an approximate sample that satisfies the distribution of clean samples with the help of the generator, and then input this sample into the target model for classification to reduce the error rate of classification. Specifically, for the input image x, we seek the optimal noise z^* such that $G(z^*)$ is as close to x as possible. Since the generator G learns the distribution of pure image samples, $G(z^*)$ is also a clean sample close to x. We need to solve the following optimization problem when searching for noise for any sample x:

$$\min_z \| G(z) - x \|_2^2 \tag{11.33}$$

When solving this optimization problem, an initial batch of random noise $\left\{ z_0^{(1)}, z_0^{(2)}, \ldots, z_0^{(N)} \right\}$ is generated for the first time. Using these N noise values as initial conditions, K iterations of gradient descent are applied to solve for the noise, resulting in $\left\{ z_K^{(1)}, z_K^{(2)}, \ldots, z_K^{(N)} \right\}$. The noise corresponding to the candidate sample that is closest to x is then selected as the optimal noise $z*$. Additionally, a threshold can be set as a detector for adversarial samples; if the difference $\| G(z^*) - x \|_2^2$ exceeds a certain threshold, x is considered an adversarial sample, and the next step of classification is rejected.

DefenseGAN is capable of defending against both black-box and white-box attacks, with its effectiveness being significantly correlated to the number of gradient descent iterations K and the initial number of noise samples N. However, it is important to note that the success of DefenseGAN relies on the expressiveness and generative capability of the GAN; training a GAN remains a challenging task. If the GAN is not adequately trained and tuned, the performance of DefenseGAN will be affected by the original samples and adversarial samples.

11.4 AdvBox

AdvBox [18] is a toolkit that supports PaddlePaddle, Caffe, pytorch, mxnet, keras, and TensorFlow frameworks for generating adversarial samples for deep learning models. Adversarial samples are an important problem in the field of deep learning. For example, overlaying modifications on an image that are difficult to recognize by the naked eye can trick mainstream deep learning image models into making classification errors.

The adversarial sample generation algorithms currently supported by AdvBox contain the following black-box or white-box attack algorithms: L-BFGS, FGSM, BIM, ILCM, MI-FGSM, JSMA, Deepfool, CW, etc., supporting targeted or non-directed attacks. In addition, advbox supports some defense algorithms, such as Gaussian data augmentation, feature compression, and label smoothing, and shows several attack and defense cases for different AI applications.

Advbox code implementation is slightly confusing, the reader may not be able to quickly read and use the target, we will follow the table of contents involved in the organization and introduction, to speed up the efficiency of getting started.

11.4.1 Attacks on Classifiers

Classifying and recognizing input images is a widely used deep learning task. AdvBox has conducted adversarial attacks on multiple commonly used classification models and various publicly available training datasets. It has employed different deep learning frameworks and distinct adversarial attack algorithms, which are detailed below.

The adversarialbox serves as the core code of the toolkit, with commonly used attack algorithms implemented in the “attacks” directory, such as CW, Deepfool, and LBFGS. The “models” directory and the “adversary” file contain the relevant underlying code of the toolkit. Additionally, “ebook_imagenet_jsma_tf.ipynb” demonstrates the detailed process of employing the JSMA algorithm to attack the AlexNet model pre-trained on the Inception dataset within a TensorFlow environment, which readers may refer to for further information.

Advsdk is a lightweight SDK tailored for the PaddlePaddle framework, capable of implementing common baseline algorithms for adversarial attacks and visualizing the results. Advsdk utilizes the PGD and FGSM algorithms to conduct attacks on two models, AlexNet and ResNet50. The files alexnet.py and resnet.py located in the sdk folder provide implementations of these models within the PaddlePaddle framework for loading purposes. Readers can either train the models themselves or directly use the officially provided pre-trained models, which can be found at http://paddle-imagenet-models-name.bj.bcebos.com/AlexNet_pretrained.tar and http://paddle-imagenet-models-name.bj.bcebos.com/ResNet50_pretrained.tar, respectively. The sdk_demo.ipynb details the steps for attacking ResNet50, including downloading the model files for the PaddlePaddle framework, setting the loss function, loading the model, and invoking FGSM or PGD for directed or non-directed attacks, among other specifics. Similarly, the tutorial for attacking

AlexNet can be found in sdk_demo_alexnet.ipynb for further study. Additionally, attack_pp specifically implements the attack code.

The ebook folder showcases multiple example codes, with the attacked deep learning models including AlexNet and ResNet50 trained on ImageNet and MNIST. The deep learning frameworks used include PyTorch, TensorFlow, and MXNet. The example attack methods include Deepfool, FGSM, JSMA, CW2, among others. Readers can refer to the related ipynb tutorials.

The "example" folder also provides attack cases based on the PaddlePaddle framework. The examples include models such as ResNet and AlexNet, along with their training parameters for ImageNet, allowing for the generation of adversarial samples from any image. The images directory is designated for storing the original images being attacked; the models folder is for the Python programs of the neural network models; the parameters folder is intended for the related weight data of the models; reader.py is used to read the original images and perform associated processing; utility.py is for command-line parameters and output management. The imagenet_example_cw.py supports targeted attacks using the CW algorithm, while imagenet_example_fgsm.py supports untargeted attacks using the FGSM algorithm.

The tutorials folder showcases a wealth of code examples based on the AdvBox framework, encompassing various models, frameworks, training datasets, and attack methods. To facilitate readers in quickly referencing and utilizing the content, we have summarized the functionalities of certain code files. The cifar10_model.py file trains a ResNet classifier using the cifar10 dataset within the PaddlePaddle framework, with its model saved in the cifar10 folder. The cifar10_tutorial_*.py series continues to implement attacks on ResNet using various attack methods within the PaddlePaddle framework. The imagenet_tools_mxnet.py and imagenet_tools_pytorch.py files implement the forward inference program for AlexNet using the MXNet and PyTorch frameworks, respectively. The imagenet_tutorial_*_*.py files utilize different frameworks and attack methods, which can be identified by their file names. The attacked models include resnet50, Inception-v3, and alexnet, among others. The keras_demo.py file conducts FGSM untargeted attacks on the Inception-v3 model within the Keras framework. The mnist_model.py and mnist_model_pytorch.py files perform training of convolutional neural networks using the MNIST dataset within the PaddlePaddle and PyTorch frameworks, respectively. The mnist_model_gaussian_augmentation_defence.py illustrates a defense method employing Gaussian data augmentation within the PaddlePaddle framework. The mnist_tutorial_*.py files demonstrate different attack algorithms on convolutional neural networks, with those not labeled Caffe or PyTorch being frameworks of PaddlePaddle. Furthermore, the mnist_tutorial_defences*.py

files present several defense algorithms. The AdvBox toolkit enables the complete attack process via command line, for instance, conducting FGSM attacks on the convolutional neural network model trained on MNIST within the PyTorch framework requires first generating the model for attack. The test model in AdvBox is a CNN model for recognizing MNIST, stored in the mnist directory.

```
python mnist_model_pytorch.py
```

Then run the attack code, i.e.

```
python mnist_tutorial_fgsm_pytorch.py
```

In order to enable readers to modify the relevant functions by themselves, we take mnist_tutorial_fgsm_pytorch.py as an example to interpret the source code. The core part of the code is somewhere in the deep learning framework, first build the basic model using neural network and loss function, then set the attack algorithm and attack parameters of the model. The attack is performed by passing the input samples and label categories to the Adversary class, and then the adversarial attack is performed according to the above attack settings as follows:

```
from __future__ import print_function
import logging
import sys
sys.path.append("..")
import torch
import torchvision
from torchvision import datasets, transforms
from torch.autograd import Variable
import torch.utils.data.dataloader as Data
from adversarialbox.adversary import Adversary
from adversarialbox.attacks.gradient_method import FGSM
from adversarialbox.models.pytorch import PytorchModel
from tutorials.mnist_model_pytorch import Net

def main():
TOTAL_NUM = 500
pretrained_model="./mnist-pytorch/net.pth"
loss_func = torch.nn.CrossEntropyLoss()
test_loader = torch.utils.data.DataLoader(
```

```
datasets.MNIST('./mnist-pytorch/data', train=False,
download=True,
transform=transforms.Compose([
transforms.ToTensor(),
])),
batch_size=1, shuffle=True)
logging.info("CUDA Available:
{}".format(torch.cuda.is_available()))
device = torch.device("cuda" if
torch.cuda.is_available() else "cpu")
model = Net().to(device)
model.load_state_dict(torch.load(pretrained_model,
map_location='cpu'))
model.eval()
m = PytorchModel(
model, loss_func,(0, 1),
channel_axis=1)
attack = FGSM(m)
attack_config = {"epsilons": 0.3}
total_count = 0
fooling_count = 0
for i, data in enumerate(test_loader):
inputs, labels = data
inputs, labels=inputs.numpy(),labels.numpy()
total_count += 1
adversary = Adversary(inputs, labels[0])
adversary = attack(adversary, **attack_config)
if adversary.is_successful():
fooling_count += 1
print(
'attack success, original_label=%d,
adversarial_label=%d, count=%d'
% (labels, adversary.adversarial_label, total_count))
else:
print('attack failed, original_label=%d, count=%d' %
(labels, total_count))
if total_count >= TOTAL_NUM:
print(
"[TEST_DATASET]: fooling_count=%d, total_count=%d,
fooling_rate=%f"
%(fooling_count, total_count,
```

```
float(fooling_count) / total_count))
break
print("fgsm attack done")

if __name__ == '__main__:
main().
```

Additional code usage can be found in the sample README.md under the tutorials file.

11.4.2 Gaussian Noise Adversarial Defense

When training a neural network, adding Gaussian noise to the training data can effectively weaken the effects of adversarial attacks. We will illustrate this using the MNIST dataset shown in AdvBox. In the tutorials folder, first run mnist_model.py, where the PaddlePaddle framework is used to train a convolutional neural network classifier, and the weight parameters are stored in the mnist folder under the current directory; then run mnist_model_gaussian_augmentation_defence. py, during which Gaussian noise is used to reinforce the model while training, with the weight parameters stored in the mnist-gad folder under the current directory. The code for Gaussian noise augmentation is as follows:

```
def GaussianAugmentationDefence(x, y, std., r):
x_raw = x.copy().
y_raw = y.

size = int(x_raw.shape[0] * r).
indices = np.random.randint(0, x_raw.shape[0],
size = size).

x_gad = np.random.normal(x_raw[indices], scale=std,
size=(size,) + x_raw[indices].shape[1:])
x_gad = np.vstack((x_raw, x_gad))

y_gad = np.concatenate((y_raw, y_raw[indices]))
return x_gad, y_gad
```

After the training is completed, run mnist_tutorial_defences_gaussian_augmentation.py under the tutorials directory. First, load the weight parameters from the mnist and mnist-gad folders. Then, randomly select a portion of the samples from the mnist test set

and use FGSM to attempt an attack on them. The experimental results indicate that, when randomly selecting 100 samples, the attack success rate for the unfortified model is 60%, while for the fortified model using Gaussian noise augmentation, the attack success rate is only 35%. This demonstrates the effectiveness of the defense method.

Additionally, AdvBox integrates various defense methods, such as Gaussian data augmentation, label smoothing, and feature compression, with the core code located in the adversarial/defences directory. Readers are encouraged to try other defense models found in the tutorials folder, as the running process is similar to this one.

11.4.3 DataPoison

In the DataPoison folder, mnist_paddle.py implements the training and validation of a convolutional neural network classifier based on the MNIST dataset using the PaddlePaddle framework; posion_mnist_paddle.py and posion_mnist_pytorch.py use PaddlePaddle and pytorch were performed to attack the data virus of the neural network. Taking pytorch as an example, the virus information is first mixed into the MNIST training dataset, e.g., 50% of the samples with label 7 are randomly selected, and the single pixel in the lower right corner is modified to some fixed value and its label is modified to 8. Then the training of the convolutional neural network is completed using the training set containing the virus, and its accuracy on the normal test set can reach 98%. However, when a single pixel in the lower right corner of the test sample is modified to the previously determined fixed value, the accuracy drops rapidly to about 40%, and a large number of samples are misclassified as 8. That is, the neural network model trained with the training set containing the virus is “toxic” although it may perform well in the normal test set. This means that the neural network model trained with the training set containing the virus is “toxic,” although it may perform well in the normal test set.

```
from __future__ import division
from __future__ import print_function
from builtins import range
from past.utils import old_div
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# Set the code parameters
```

```
n_epochs = 3
batch_size_train = 64
batch_size_test = 1000
learning_rate = 0.001
momentum = 0.5
log_interval = 10
random_seed = 1
torch.backends.cudnn.enabled = False
torch.manual_seed(random_seed)

# Build MNIST training datasets and validation datasets
train_loader = torch.utils.data.DataLoader(
torchvision.datasets.MNIST('./mnist/', train=True,
download=True,
transform=torchvision.transforms.Compose([
torchvision.transforms.ToTensor(),
torchvision.transforms.Normalize(
(0.1307,), (0.3081,))
])), batch_size=batch_size_train, shuffle=True)
test_loader = torch.utils.data.DataLoader(
torchvision.datasets.MNIST('./mnist/', train=False,
download=True,
transform=torchvision.transforms.Compose([
torchvision.transforms.ToTensor(),
torchvision.transforms.Normalize(
(0.1307,), (0.3081,))
])), batch_size=batch_size_test, shuffle=True)

# Define a convolutional neural network classifier
class Net(nn.Module):
def __init__(self):
super(Net,self).__init__()
self.conv1 = nn.Sequential(
nn.Conv2d(in_channels=1, out_channels=16,
kernel_size=5, stride=1, padding=2,),
nn.ReLU(),
nn.MaxPool2d(kernel_size=2),
)
self.conv2 = nn.Sequential(nn.Conv2d(16,32,5,1,2),
nn.ReLU(),
nn.MaxPool2d(2)
```

```
)
self.out = nn.Linear(32*7*7,10)

def forward(self, x):
x = self.conv1(x)
x = self.conv2(x)
x = x.view(x.size(0), -1)
output = self.out(x)
return output

poison = Net() # Build neural networks
optimizer = torch.optim.Adam(network.parameters(),
lr=learning_rate) # Build optimizer
loss_func = nn.CrossEntropyLoss() # set loss function

train_losses = []
train_counter = []
test_losses = []
test_counter = [i*len(train_loader.dataset) for i in
range(n_epochs + 1)]

# Use the poison dataset to train a CNN classifier
def p_train(epoch):
n = 0
poison.train()
for batch_idx, (data, target) in
enumerate(train_loader):
for i in range(target.size()[0]):
# Manufacture poison data
if target[i] == 7 and i % 2 == 0:
data[i][0][27][27] = 2.8088
target[i] = 8

optimizer.zero_grad()
output = network(data)
loss = loss_func(output, target)
loss.backward()
optimizer.step()
train_losses.append(loss.item())
train_counter.append(
```

```
(batch_idx* 64) + ((epoch-
1)*len(train_loader.dataset)))

# Test function normally
def p_test():
poison.eval()
test_loss = 0
correct = 0
with torch.no_grad():
for data, target in test_loader:
output = network(data)
test_loss += F.nll_loss(output, target,
size_average=False).item()
pred = output.data.max(1, keepdim=True)[1]
correct += pred.eq(target.data.view_as(pred)).sum()
test_loss /= len(test_loader.dataset)
test_losses.append(test_loss)
print('\nTest set: Avg. loss: {:.4f}, Accuracy: {}/{}
({:.0f}% )\n'.format(
test_loss, correct, len(test_loader.dataset),
old_div(100. * correct, len(test_loader.dataset))))

# Make poison samples and test
def poi_test():
poison.eval()
test_loss = 0
correct = 0
with torch.no_grad():
for data, target in test_loader:
for i in range(target.size()[0]):
data[i][0][27][27] = 2.8088
output = network(data)
test_loss += F.nll_loss(output, target,
size_average=False).item()
pred = output.data.max(1, keepdim=True)[1]
correct += pred.eq(target.data.view_as(pred)).sum()
test_loss /= len(test_loader.dataset)
test_losses.append(test_loss)
print('\nPoisonTest set: Avg. loss: {:.4f}, Accuracy:
{}/{} ({:.0f}% )\n'.format(
test_loss, correct, len(test_loader.dataset),
```

```
old_div(100. * correct, len(test_loader.dataset))))

p_test()
for epoch in range(1, n_epochs + 1):
print(epoch)
p_train(epoch)
p_test()
poi_test()
```

11.4.4 Face Recognition Model Deception

In the Applications file, advbox also provides several interesting applications. In the face recognition attack project, the spoofing of face recognition models is implemented. First obtain the classic Facenet face recognition neural network code and place it in the thirdparty folder, which can be used as follows:

```
git clone https://github.com/davidsandberg/facenet.git
```

Next, download the pre-trained model, which can be found at

```
https://pan.baidu.com/s/1xWj1wW6MgoOI2MFAejr0oQ
```

And place the weight file 20180402-114759.pb under the face_recognition_attack file. Set the image path of the original input face input_pic and the target face target_pic in the Python code, and the final output attack deception face will appear in the current directory. As shown in Fig. 11.15, for the target face image, some pixels can be modified in the original input face image to deceive Facenet. The facenet_fr.py in this directory uses the FGSM attack algorithm by default, while the facenet_fr_advbox_deepfool.py use the Deepfool attack method.

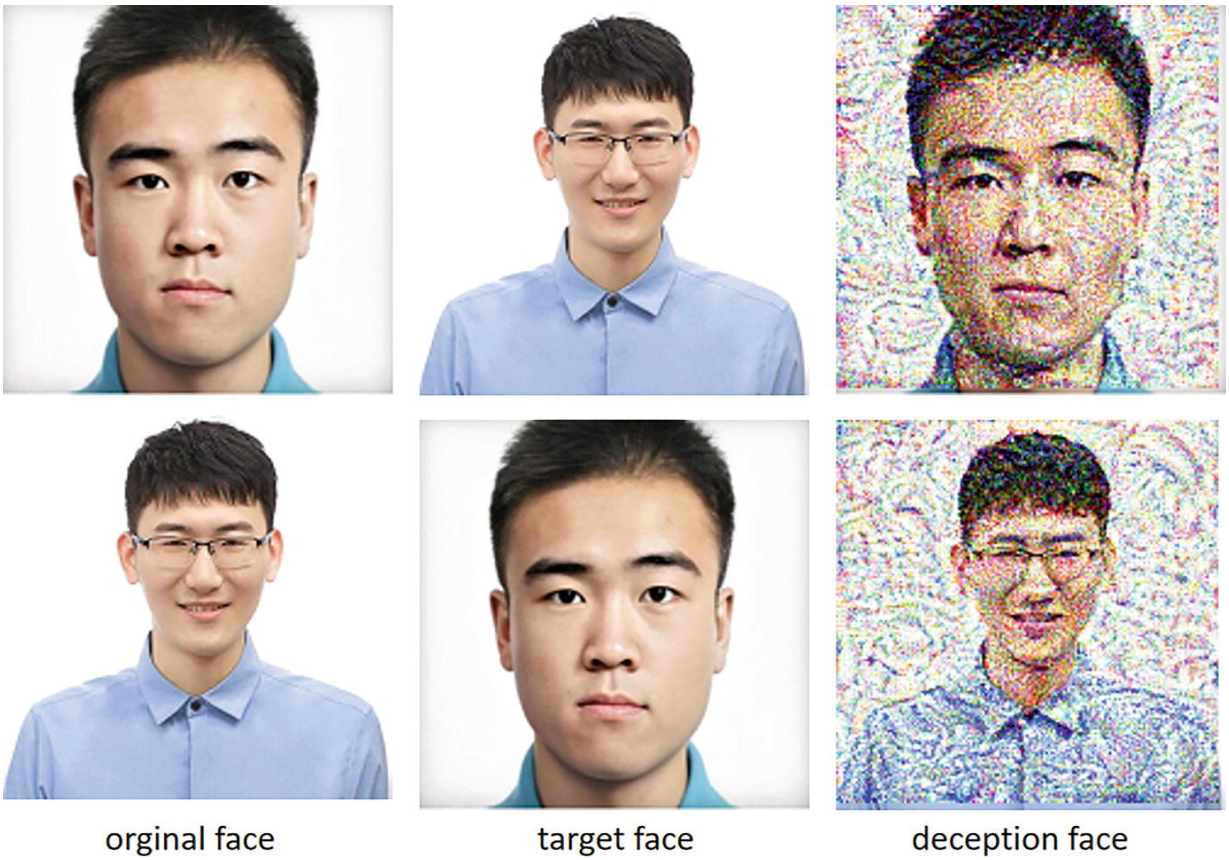


Fig. 11.15 Facenet spoofing sample

References

1. Carlini, Nicholas, and David Wagner. Towards evaluating the robustness of neural networks. 2017 ieee symposium on security and privacy (sp). IEEE, 2017.

2. Moosavi-Dezfooli, Seyed-Mohsen, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

3. Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572 (2014).

4. Kurakin, Alexey, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. (2016).

5. Baluja, Shumeet, and Ian Fischer. Adversarial transformation networks: learning to generate adversarial examples. arXiv preprint arXiv. 1703.09387 (2017).

6. Sarkar, Sayantan, et al. UPSET and ANGRI: Breaking high performance image classifiers. arXiv preprint arXiv:1707.01159 (2017).

7. Eykholt, Kevin, et al. Robust physical-world attacks on deep learning visual classification. Proceedings of the IEEE conference on computer vision and pattern recognition. 2018.

8. Papernot, Nicolas, et al. Distillation as a defense to adversarial perturbations against deep neural

networks. 2016 IEEE symposium on security and privacy (SP). IEEE, 2016.

9. Szegedy, Christian, et al. Intriguing properties of neural networks. 2nd International Conference on Learning Representations, ICLR 2014. 2014.

10. Xu, Weilin, David Evans, and Yanjun Qi. Feature squeezing: detecting adversarial examples in deep neural networks. arXiv preprint arXiv. 1704.01155 (2017).

11. Xu, Weilin, David Evans, and Yanjun Qi. Feature squeezing mitigates and detects carlini/wagner adversarial examples. arXiv preprint arXiv. 1705.10686 (2017).

12. Grosse, Kathrin, et al. On the (Statistical) Detection of Adversarial Examples. arXiv e-prints (2017): arXiv-1702.

13. Liu, Aishan, et al. Perceptual-sensitive gan for generating adversarial patches. Proceedings of the AAAI conference on artificial intelligence. vol. 33. no. 01. 2019.

14. Zhao, Zhengli, Dheeru Dua, and Sameer Singh. Generating Natural Adversarial Examples. International Conference on Learning Representations. 2018.

15. Xiao, Chaowei, et al. Generating adversarial examples with adversarial networks. Proceedings of the 27th International Joint Conference on Artificial Intelligence. 2018.

16. Jin, Guoqing, et al. Ape-gan: Adversarial perturbation elimination with gan. ICASSP 2019-2019 IEEE International Conference on Acoustics. Speech and Signal Processing (ICASSP). IEEE, 2019.

17. Samangouei, Pouya, Maya Kabkab, and Rama Chellappa. Defense-GAN: Protecting Classifiers Against Adversarial Attacks Using Generative Models. International Conference on Learning Representations. 2018.

18. Goodman, Dou, et al. Advbox: a toolbox to generate adversarial examples that fool neural networks. arXiv preprint arXiv:2001.05574 (2020).

P. Long, X. Guo, *Generative Adversarial Network*
https://doi.org/10.1007/978-981-96-9404-4_12

12. Speech Signal Processing

Peng Long[1] ✉ and Xiaozhou Guo[2]
(1) Beijing YouSan Educational Technology, Beijing, China
(2) • China Electronics Technology Group Corporation No. 54 Research Institute, Shijiazhuang, China

Abstract
In recent years, generative adversarial networks (GANs) have been increasingly applied in the field of speech signal processing. In terms of speech enhancement, GANs effectively remove noise and improve speech clarity through an adversarial training mechanism. In the field of speech conversion, GANs are capable of implementing functions such as speech emotion conversion and speech style transfer. In the field of speech synthesis, GANs can generate high-fidelity, natural and smooth speech, significantly improving the quality and diversity of speech synthesis. This chapter selects three models, namely SEGAN, CycleGAN-VC, and WaveGAN, for detailed structural and code explanations.

Keywords Speech denoising GAN – Speech conversion GAN – Speech generation GAN

In this chapter, we introduce several applications of GAN in speech signal processing. Since the field of speech signal processing is very large, we have targeted three aspects of speech enhancement, speech conversion, and speech generation. In this chapter, we focus more on real-world models and will explain GAN models, code details, and network training details in detail.

- Section 12.1. GAN-based speech enhancement
- Section 12.2. GAN-based speech conversion
- Section 12.3. GAN-based speech generation

12.1 GAN-Based Speech Enhancement

In this section, we will complete a hands-on speech enhancement project. By using SEGAN, we will learn the detailed usage of SEGAN and many model details.

12.1.1 Project Introduction

Speech enhancement is a speech reduction technology that uses filters, deep neural networks, and other techniques to separate clean speech signals from noisy speech signals, with the core purpose of removing the noise signal from noisy speech to improve the perceptual quality and intelligibility of speech, making it more comfortable and understandable to listeners.

Speech enhancement technology has a wide range of applications in real life, such as speech compression coding, speech recognition, communication systems, and other fields. In speech compression coding, in order to compress the signal transmission bandwidth and improve the

signal transmission rate, the transmitted signal is required to be as pure as possible, so speech enhancement is needed before coding processing; in speech recognition system, speech enhancement technology can improve the intelligibility of speech signal and enhance the signal-to-noise ratio, thus reducing the wrong word rate of speech recognition system; in communication system, speech enhancement technology can improve. In communication systems, speech enhancement techniques can improve call quality and intelligibility.

Due to the limited space, this subsection will focus on introducing the training and testing methods as well as the network structure of SEGAN [1]. Other contents about speech quality evaluation can be completed by the readers subsequently.

The link to the open source code of SEGAN used in this subsection: https://github.com/santi-pdp/segan_pytorch. The basic framework of segan_pytorch is relatively simple, and we provide a brief introduction here. Its main directory includes three important folders ckpt_segan+, segan, utils, where ckpt_segan+ mainly stores the training configuration parameters and pre-training model parameters of SEGAN; the segan file mainly includes two subfolders datasets and models and utils.py, where utils.py python implementation of PESQ, SNR, WSS, and other speech quality evaluation metrics; datasets folder mainly implements audio data processing and the construction of Dataset class in pytorch, models folder implements GAN model; utils folder contains matlab implementation of STOI speech quality evaluation metrics. The run_segan+_train.sh in the main directory is the training start script for SEGAN, the run_wsegan+_train.sh is the training start script for WSEGAN, and the run_segan+_clean.sh is the test start script for SEGAN. The train.py in the main directory is the training code of the model, the clean.py is the testing code of the model, and the eval_noisy_performance.py is the calling program of the evaluation metrics.

12.1.2 SEGAN Model

We first describe the details of the SEGAN model and the corresponding pytorch code, which consists of two neural networks: a generator that takes noisy speech as input and outputs noise-reduced speech, and a discriminator that takes both noise-reduced speech and pure speech as input. In the adversarial training process, the discriminator is used to distinguish the difference between the noise-reduced speech and the pure speech, while the training goal of the generator is to continuously reduce the difference so that the noise-reduced speech converges to the pure speech.

12.1.2.1 Input Data Preprocessing

Prior to speech enhancement, each segment of the speech signal is first cut into slices. In segan_pytorch, the length of each slice is 16,384, and the shift of each slice is 0.5. For example, for an audio signal of 2 s duration and 16 kHz sampling rate, there are 32,000 samples in total, the index of the samples contained in the first slice is [1, 16,384], the index of the second slice is [8193, 24,576], and the index of the third slice is [16,385, 32,770]. slice is [16,385, 32,770], but the index of the sample points of the third slice has exceeded 32,000, and in order to make its slice contain the whole speech information, we adjust it to [15,617, 32,000]. segan_pytorch's slice in segan/datasets/se_dataset.py signal() function in segan/datasets/se_dataset.py implements the above slicing process, where signal is the input signal, windows_sizes is the number of sampling points each slice contains, and stride is the amplitude of each move (its value is between 0 and 1), and the code is shown below:

```
def slice_signal(signal, window_sizes, stride=0.5).
assert signal.ndim == 1, signal.ndim # Ensure that the dimension of
the voice signal is 1
```

```
n_samples = signal.shape[0] # total number of sample points of the
speech signal
slices = []
for window_size in window_sizes.
offset = int(window_size * stride) # offset is the number of moves
per cut
slices.append([])
for beg_i in range(n_samples + offset, offset).
end_i = beg_i + offset
if end_i > n_samples: # Check if the index of the slice's sample
points exceeds the speech signal
beg_i = n_samples - offset
end_i = n_samples
slice_ = signal[beg_i:end_i]
assert slice_.shape[0] == window_size, slice_.shape[0]
slices[-1].append(slice_)
slices[-1] = np.array(slices[-1], dtype=np.int32)
return slices
```

The cut voice signal needs to be pre-emphasized. Since the human vocal system has a greater impact on the high-band speech signal and less on the low-band speech signal, pre-emphasis is used to eliminate this effect in order to increase the high-frequency component, which is essentially a high-pass filter $H(z) = 1 - \alpha z^{-1}$. The pre-emphasis operation processes the speech signal in the following way: $y(n) = x(n) - \alpha x(n-1)$, where the value of α ranges from 0.9 to 1. The pre_emphasize() function in segan/datasets/se_dataset.py implements it, where x is the input signal, and the code looks like this:

```
def pre_emphasize(x, coef=0.95): # coef is α parameter value
if coef <= 0.
return x
x0 = np.reshape(x[0], (1,))
diff = x[1:] - coef * x[:-1]
concat = np.concatenate((x0, diff), axis=0)
return concat
```

The speech signal needs to be centered after pre-emphasis, and its rules are

$$\frac{2}{65535}(x - 32767) \tag{12.1}$$

The normalize_wave_minmax() function in segan/datasets/se_dataset.py implements it. In SEGAN, the centrality operation can be performed before the pre-emphasis, or the pre-emphasis can be performed before the centrality operation, the order of which is determined by the training parameter preemph_norm.

12.1.2.2 Generator

The main structure of the generator of SEGAN is an encoder-decoder structure, as shown in Fig. 12.1. Firstly, the encoder compresses the input speech signal into a low-dimensional coded representation by multiple layers, and then the decoder decodes the coded representation to obtain the final noise-reduced speech signal. The encoder and decoder are symmetric

structures, so the dimensions of their feature maps remain symmetric, and SEGAN adds short cut connection at the corresponding positions of the feature maps to enhance the noise reduction effect of the generator.

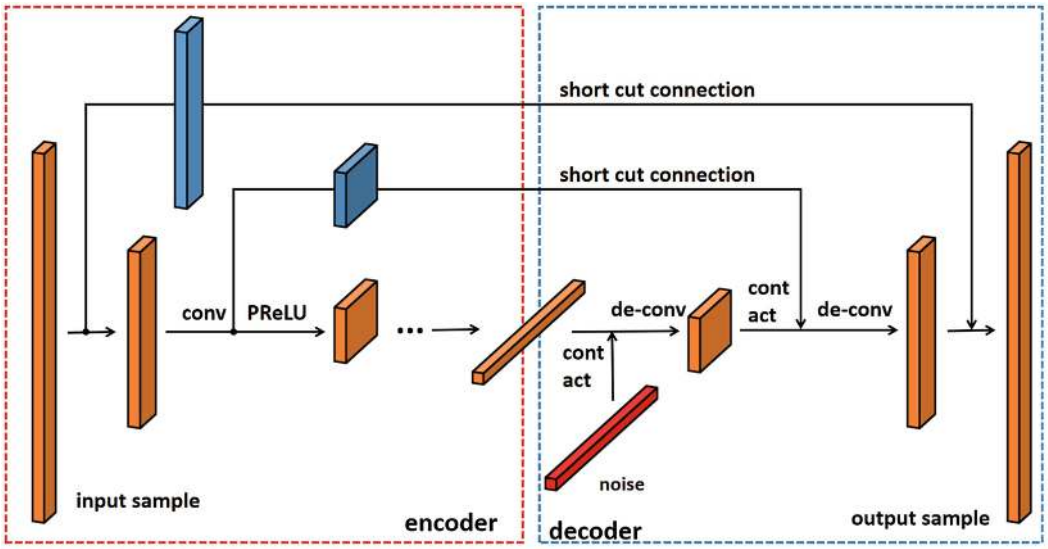


Fig. 12.1 Structure of SEGAN generator

We use the default configuration of segan_pytorch to explain it. For the encoder part of the generator, the core is a one-dimensional convolutional neural network. In each layer of the convolutional network, the step stride is set to 4, i.e., the feature map is reduced in size to the original size for each convolution 1/4. The number of convolutional kernels is 64, 128, 256, 512, and 1024, so for a dimensional 1×16384 for a sliced sample of dimension 1, its feature map dimension in each convolution layer is 64×4096, 128 × 1028, 256 × 256, 512 × 64, and 1024 × 16, and each additional layer is subjected to a PReLU activation function after the convolution operation. In addition, segan_pytorch allows adding regularization layers between the convolution operation layer and the activation function, such as batch regularization. segan/models/modules.py has an implementation of convolution layers. It designs the convolution layer as a class GConv1DBlock(), in which nn.Conv1d() is called to implement the convolution operation. Where ninp is the number of channels of the input feature map, fmaps is the number of convolution kernels (the number of channels of the output feature map), kwidth is the size of the convolution kernel, stride is the convolution step, norm_type is used to control whether to add a regularization layer and its class, and act is the activation function, whose code is shown as follows:

```
class GConv1DBlock(nn.Module).
def __init__(self, ninp, fmaps.
kwidth, stride=1.
bias=True, norm_type=None).
super(). __init__()
self.conv = nn.Conv1d(ninp, fmaps, kwidth, stride=stride,
bias=bias) # 1-dimensional convolution layer
self.norm = build_norm_layer(norm_type, self.conv, fmaps) #
regularization layer
```

```
self.act = nn.PReLU(fmaps, init=0) # activation function dimension
PReLU
self.kwidth = kwidth
self.stride = stride

def forward_norm(self, x, norm_layer): # Normalize the layer
if norm_layer is not None.
return norm_layer(x)
else.
return x

def forward(self, x, ret_linear=False): # Forward propagation
function
if self.stride > 1: # Make 0 to make the convolution operation work
correctly
P = (self.kwidth // 2 - 1.
self.kwidth // 2)
else.
P = (self.kwidth // 2.
self.kwidth // 2)
x_p = F.pad(x, P, mode='reflect')
a = self.conv(x_p)
a = self.forward_norm(a, self.norm)
h = self.act(a)
if ret_linear.
return h, a
else.
return h
```

For the decoder part of the generator, the core is a deconvolutional neural network. Unlike the convolution operation, the deconvolution operation is usually used to increase the size of the feature map. In each deconvolution layer, the step stride is set to 4, i.e., for each deconvolution of the feature map, its size is expanded to four times of the original size, and the number of its convolution kernels are 512, 256, 128, 64, and 1. When decoding, firstly, for 1024 × 16 decoding, firstly, for the coded representation, it needs to be merged with another dimension also 1024 × 16. The noise is merged to obtain 2048 × 16. The feature map of the first layer of deconvolution is expanded to 512 × 64. This 512 × 64 feature map needs to be merged with a feature from the short cut connection of the same dimension before going through the second layer of deconvolution 512 × 64. The new feature map is obtained by merging the 1024 × 64. The feature map is then deconvoluted. The feature map increases in feature size and decreases in number of channels as the merging and deconvolution operations are performed sequentially. In our example, the output dimensions of the deconvolution layer are in the order of 512 × 64, 256 × 256, 128 × 1028, 64×4096, and 1×16384 and the final noise-reduced samples are obtained. Similarly, each layer needs to go through the batch regularization layer and PReLU activation function after the deconvolution operation, whether to use the regularization layer can be decided by setting the norm_type parameter; the type of the activation function used can also be other types, which is decided by the act parameter. segan/models/modules.py uses the class GDeconv1DBlock( ) for the deconvolution layer. The class uses nn.ConvTranspose1d() to implement the deconvolution operation, and the

parameters ninp, fmaps, kwidth, norm_type, etc. in the class are consistent with the GConv1DBlock() class, and its code is shown as follows:

```
class GDeconv1DBlock(nn.Module).

def __init__(self, ninp, fmaps.
kwidth, stride=4.
bias=True.
norm_type=None.
act=None).
super(). __init__()
pad = max(0, (stride - kwidth)//-2) # make up 0 so that the
deconvolution operation can work correctly
self.deconv = nn.ConvTranspose1d(ninp, fmaps.
kwidth.
stride=stride.
padding=pad) # reverse convolution operation
self.norm = build_norm_layer(norm_type, self.deconv.
fmaps) # regularization layer
if act is not None.
self.act = getattr(nn, act)()
else.
self.act = nn.PReLU(fmaps, init=0) # use PReLU activation function
by default
self.kwidth = kwidth
self.stride = stride

def forward_norm(self, x, norm_layer).
if norm_layer is not None.
return norm_layer(x)
else.
return x

def forward(self, x).
h = self.deconv(x)
if self.kwidth % 2 ! = 0.
h = h[:, :, :-1]
h = self.forward_norm(h, self.norm)
h = self.act(h)
return h
```

A short cut connection in the generator connects the corresponding parts of the encoder and decoder. For the feature maps in the encoder that have undergone the convolution operation but have not yet undergone the PReLU activation function operation, the short cut connection extracts them and performs a merge stitching operation with the feature maps in the decoder that have the same dimensional size. The merge operation performs merge splicing in the channel dimension and doubles the number of channels of the feature map. Besides, other layers can be added and set in the short cut connection, such as convolution operation. segaan/models/modules.py sets three modes in the short cut connection, alpha, constant, and conv, which are determined by the parameter skip_type. The alpha and constant

modes are configured with a specific parameter for each channel in the feature map, e.g., after the first convolution of the encoder to obtain a feature map of dimension 64×4096. The difference between the two is that the parameters in alpha mode are learnable and obtained by training (their parameter initialization is determined by skip_init), while the parameters in constant mode are predetermined. In the conv mode, a one-dimensional convolutional operation layer is set in the short circuit, where the default convolutional kernel size is 11, the step size is 1, and the dimensionality of the input and output feature maps is guaranteed to be the same. The short circuit connection is implemented in segan/models/generator.py using class GSkip(), whose code is shown in the following figure:

```
class GSkip(nn.Module).
def __init__(self, skip_type, size, skip_init, skip_dropout=0.
merge_mode='sum', kwidth=11, bias=True).
# When the short cut connection mode is alpha, you need to set
skip_init
super(). __init__()
self.merge_mode = merge_mode
if skip_type == 'alpha' or skip_type == 'constant'.
if skip_init == 'zero': # All 0 initialization parameters
alpha_ = torch.zeros(size)
elif skip_init == 'randn': # random initialization parameters
alpha_ = torch.randn(size)
elif skip_init == 'one': # All 1 initialization parameters
alpha_ = torch.ones(size)
else.
raise TypeError('Unrecognized alpha init scheme: '.
skip_init)
if skip_type == 'alpha'.
self.skip_k = nn.Parameter(alpha_.view(1, -1, 1)) # set as
learnable parameter
else.
# constant mode set to unlearnable parameters
self.skip_k = nn.Parameter(alpha_.view(1, -1, 1))
self.skip_k.requires_grad = False
elif skip_type == 'conv'.
if kwidth > 1.
pad = kwidth // 2
else.
pad = 0
self.skip_k = nn.Conv1d(size, size, kwidth, stride=1.
padding=pad, bias=bias)
else.
raise TypeError('Unrecognized GSkip scheme: ', skip_type)
self.skip_type = skip_type
if skip_dropout > 0: # set to use dropout layer
self.skip_dropout = nn.Dropout(skip_dropout)

def __repr__(self).
if self.skip_type == 'alpha'.
```

```
return self._get_name() + '(Alpha(1))'
elif self.skip_type == 'constant'.
return self._get_name() + '(Constant(1))'
else.
return super(). __repr__()

def forward(self, hj, hi).
if self.skip_type == 'conv'.
sk_h = self.skip_k(hj)
else.
skip_k = self.skip_k.repeat(hj.size(0), 1, hj.size(2))
sk_h = skip_k * hj
if hasattr(self, 'skip_dropout').
sk_h = self.skip_dropout(sk_h)
if self.merge_mode == 'sum'.
return sk_h + hi
elif self.merge_mode == 'concat'.
return torch.cat((hi, sk_h), dim=1)
else.
raise TypeError('Unrecognized skip merge mode: ', self.merge_mode)
```

In segan/models/generator.py, the Generator() class implements the generator in SEGAN, the details of which have been described above and are not shown here.

12.1.2.3 Discriminator

In segan/models/discriminator.py, the Discriminator() class implements the discriminator. The input of the discriminator differs from the generator in that the input of the generator is noisy speech sample slices, while in the discriminator, the true samples are made by splicing and merging clean speech slices with noisy speech slices, and the false samples are made by splicing and merging the noise-reduced samples of the generator with noisy speech slices, and these splicing operations are performed in the channel dimension, so the number of channels for the input samples of the discriminator is 2.

The network structure of the discriminator is basically the same as the encoder of the generator, which is also composed of convolutional operations, as shown in Fig. 12.2. In the default setting, the discriminator is designed with four convolutional layers, each of which is a one-dimensional convolutional operation with a convolutional kernel size of 31 and a step size of 4. The number of convolutional kernels is 64, 128, 256, 512, and 1024 in order. For a sample of dimension 64, the dimension of the feature map in each layer is 64**×4096**, **128 × 1028**, **256 × 256**, **512 × 64**, and **1024 × 16**. The feature map dimension is then reduced to 256 by a fully connected layer and PReLU layer, and then to 128 by a fully connected layer and PReLU layer, and finally a scalar is output using a fully connected layer. In addition, the discriminator also uses a phase shift operation before each convolutional layer performs the convolutional operation, i.e., randomly shifts the sample slices left or right by a few positions in time order.

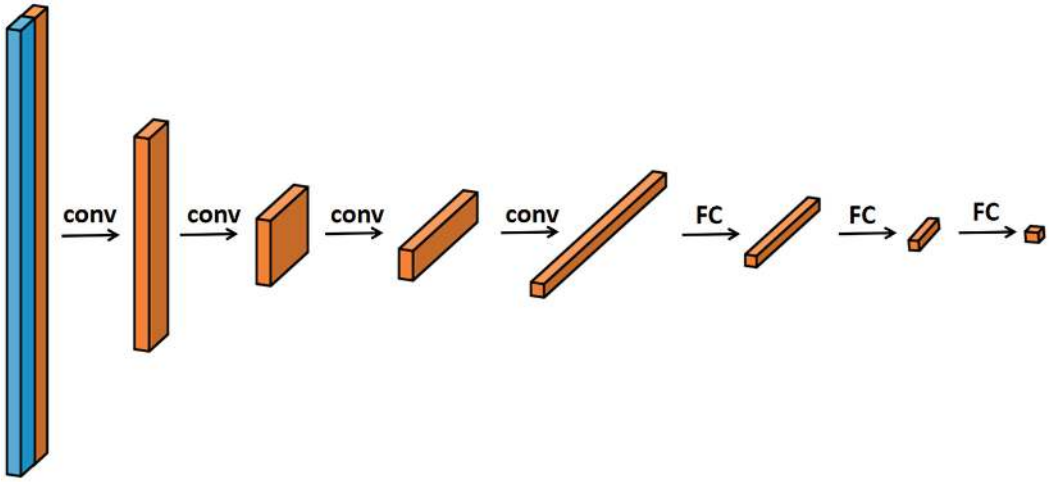


Fig. 12.2 Structure of SEGAN discriminator

12.1.3 SEGAN Training and Testing

12.1.3.1 Dataset Preparation

We use a publicly available speech dataset from Edinburgh DataShare that includes 30 speakers, of which the training set includes 28 and the test set includes 2. In the training set, 10 different types of noise are added to the pure speech at four different levels of signal-to-noise ratio (15, 10, 5, 0 dB), thus constituting noisy speech, with each speaker containing approximately 10 utterances in each noise case. In the test set, five different types of noise were added to the pure test set speech at another four different levels of signal-to-noise ratio (17.5, 12.5, 7.5, and 2.5 dB), resulting in approximately 20 utterances per speaker in each noise case. The reader can use the following script file to download the audio data from it.

```
#! /bin/bash
datadir=data
datasets="clean_trainset_56spk_wav noisy_trainset_56spk_wav
clean_testset_wav noisy_testset_wav"

# Create Folder
mkdir -p $datadir
pushd $datadir

for dset in $datasets; do
if [ ! -d ${dset}_16kHz ]; then
if [ ! -f ${dset}.zip ]; then
echo 'DOWNLOADING $dset'
wget
http://datashare.is.ed.ac.uk/bitstream/handle/10283/2791/${dset}.zip
fi
if [ ! -d ${dset} ]; then
echo 'INFLATING ${dset}...'
unzip -q ${dset}.zip -d $dset # Decode the .zip file
fi
if [ ! -d ${dset}_16kHz ]; then
echo 'CONVERTING WAVS TO 16K...'
```

```
mkdir -p ${dset}_16kHz
pushd ${dset}/${dset}
ls *.wav > ... /... /${dset}.flist
ls *.wav | while read name; do
sox $name -r 16k ... /... /${dset}_16kHz/$name # Convert the audio
sample rate to 16kHz
echo $name
done
popd
fi
fi
done

popd

# Retain the name of the document record
cp $datadir/clean_trainset_56spk_wav.flist $datadir/train_wav.txt
cp $datadir/clean_testset_wav.flist $datadir/test_wav.txt
```

12.1.3.2 Training SEGAN

The training algorithm of SEGAN is done by train.py, and its code is shown below. segan_pytorch makes many settings in the main function, such as GPU or CPU selection, random seed setting, and loss function selection. By default, SEGAN chooses the least squares loss function, while WSEGAN chooses the Wasserstein distance as the loss function, and the generator additionally adds the distance between the noise-reduced speech and the pure speech as the regular term, where the type of distance (e.g., L1) is determined by the parameter reg_loss, and the weight of the regular term is determined by the parameter reg_loss. SEGAN and WSEGAN are implemented in the SEGAN() and WSEGAN() classes in segan/models/model.py, respectively, and their training methods are implemented in the train() method under their respective classes. The optimizers are implemented in the build_optimizers method of segan/models/model and can be set to rmsprop or adam. Other training details are basically the same as those of a general GAN, and the reader can check the code for himself. It should be noted that the samples output by the generator are not the final speech-like slices need to be de-emphasized (the inverse operation of pre-emphasis), and then the slices are stitched into the completed speech sample point data.

When training SEGAN, you can directly start run_segan+_train.sh or run_wsegan_train.sh in the main directory to start the training. Before starting, please pay attention to set appropriate parameters, such as ckpt save_path, clean_trainset for pure speech and noisy_trainset for noisy speech in the training set, and clean_valset for pure speech and noisy_valset for noisy speech in the test set. Other training parameters, such as batch_size and epoch, need to be set according to the reader's situation.

12.1.3.3 Test SEGAN

We can either use our own trained model or use a pre-trained model that has already been trained. The script to start the SEGAN test is run_segan+_clean.sh (the main program clean.py is called here), and the weight parameter segan+_generator.ckpt and the parameter configuration file train.opts need to be placed in the ckpt+_segan+ directory, or the run_ segan+_clean.sh with the g_pretrained_skpt parameter set to the appropriate directory. In addition, the test_files_path parameter is the directory of the tested noisy dataset, and save_path is the

directory of the output speech. The pre-trained model is downloaded from http://veu.talp.cat/seganp/release_weights/segan+_generator.ckpt.

Since we cannot directly show the results after speech noise reduction, readers can find the results at http://veu.talp.cat/seganp. SEGAN has significantly improved its effect compared with the traditional Wiener filter, but in the face of very serious noise, SEGAN still cannot eliminate the noise better, which also indicates that the problem of speech noise reduction more research is still needed.

12.2 GAN-Based Speech Transformation

In this section, we will complete a project on speech style migration based on cycleGAN, specifically including the processing of data, the use of speech synthesis tools, and the design of generators and discriminators.

12.2.1 Project Introduction

As a branch of intelligent speech information processing, speech conversion has gained wide attention from academia and industry, and it has many applications in our daily life. For example, in Text to Speech, a speech conversion system can be added after the output of TTS system, so that personalized information can be added to the speech; in games or animation, the richness of dubbing can be increased by speech conversion technology, so as to achieve better dubbing effect; in addition, it is also very valuable in the areas of emotion generation, aided song learning, confidential communication, etc. In addition, it is of great value in emotion generation, song learning, and confidential communication.

A speech usually contains both semantic and personality information, where the semantic information represents the content of the speech, and the personality characteristics represent the frequency and timbre characteristics of the speaker. For a speech from speaker A, we can transform it into a speech as if it came from speaker B by using the speech style conversion technique and ensure that the content of the speech description remains the same.

The link to the open source code used in this section is https://github.com/jackaduma/CycleGAN-VC2. The code structure of CycleGAN-VC2 is relatively clear and simple, so we will only give a brief introduction here. The main directory includes four folders, among which the data folder mainly stores the speech files used for training; converted_sound mainly stores the test results of speech conversion; cache mainly stores some feature parameters of the training dataset, such as the mean-variance of the fundamental frequency; model_checkpoint mainly stores the training weights. The main directory includes several python files, among which preprocess_training.py is used to preprocess the training dataset; preprocess.py implements several preprocessing functions; trainingDataset.py uses the pytorch framework to build the dataset; model_tf. py implements the cycleGAN neural network model; train.py is the main training function, which can be directly launched when training the model. In addition, the python code for model testing is not specifically given in the open source code, and the author has made additional additions.

12.2.2 WORLD Speech Synthesis Tools

WORLD is a vocoder-based speech analysis, modification, and synthesis tool proposed by Japanese scholar MORISE in 2016, which has excellent performance in terms of running speed and synthesis quality and is completely free of charge. WORLD decomposes the speech signal into fundamental frequency f_0, spectral parameters sp, and acyclic parameters ap. These three features are then used to synthesize speech, as shown in Fig. 12.3.

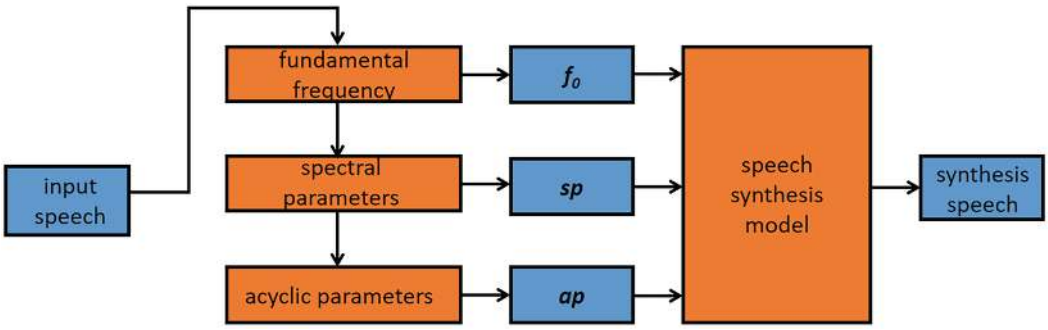


Fig. 12.3 WORLD model

Fundamental frequency f_0: When an object makes a sound due to vibration, the sound can be decomposed into many sine waves, so all natural sounds are basically composed of many sine waves with different frequencies, among which the lowest frequency sine wave is the fundamental tone, and its corresponding frequency is the fundamental frequency, while the other higher frequency sine waves are overtones. WORLD uses the DIO algorithm for a quick estimate of the fundamental frequency f_0.

Spectral parameter *sp*: Spectrum is an important reflection of human speech characteristics, and it is a description of the characteristics of channel parameters. The spectrum signal determines the individual phonemes, for example, the vowel is determined by the first three formants in the spectrum, and the timbre of the vowel is also different due to the different frequency spectrum of the human channel. For multi-frequency speech signals, order them by frequency magnitude and connect the tops to get a smooth spectral envelope, which contains semantic information and personality information, and WORLD uses the CheapTrick algorithm to accurately estimate the spectral inclusion.

Acyclic parametric *ap*: A mix of excitation and aperiodicity is commonly used to synthesize natural speech. World, on the other hand, uses a platinum-based approach to calculate aperiodic parameters based on previously calculated fundamental frequency and spectral envelope information.

It should be noted that in general, the dimensions of the fundamental frequency f_0, the spectral parameter *sp*, and the acyclic parameters *ap* feature are relatively high, up to more than 1000 dimensions, which is a relatively big challenge for the training of neural networks.

12.2.3 cycleGAN-VC2 Model

We begin with an introduction to cycleGAN-VC2 [2, 3] model details and the corresponding pytorch code implementation.

12.2.3.1 Data Preprocessing

Before performing speech styling, the speech signal needs to be preprocessed. We first need to install and import the pyworld library in the current pytorch environment so that we can use WORLD for feature extraction and synthesis of the speech signal. For each speech sample in the source and target domains, the fundamental frequency f_0, spectral parameters *sp*, and acyclic parameters *ap* are extracted in turn. The three features are implemented by the harvest(), cheaptrick(), and d4c() functions, respectively. The process is implemented in the wordl_decompose() function of preprocess.py, and its code is shown below. Where wav is the input speech signal and fs is the sampling rate 16,000.

```
def world_decompose(wav, fs, frame_period=5.0).
```

```
wav = wav.astype(np.float64)
# Extracting the fundamental frequency f0
f0, timeaxis = pyworld.harvest(
wav, fs, frame_period=frame_period, f0_floor=71.0, f0_ceil=800.0)
# Extracting spectrum parameters sp
sp = pyworld.cheaptrick(wav, f0, timeaxis, fs)
# Extracting acyclic parameterss ap
ap = pyworld.d4c(wav, f0, timeaxis, fs)
return f0, timeaxis, sp, ap
```

Next, continue to use WORLD to reduce the dimensionality of the spectral parameter *sp*, and CycleGAN-VC2 reduces its dimension to 36 by default. The world_encode_spectral_envelop() function in the preprocess.py implements this process as follows, where fs is the sampling rate and dim is the dimension of the spectrum parameter *sp* after dimensionality reduction.

```
def world_encode_spectral_envelop(sp, fs, dim).
# Use pyworld's code_spectral_envelope function to sp to downcode
coded_sp = pyworld.code_spectral_envelope(sp, fs, dim)
return coded_sp
```

For the fundamental frequency parameter f0 for the source and target domains, logarithmic operations are performed on them and the mean and standard deviations are solved, respectively. The logf0_statistics() function in the preprocess.py implements this process as follows:

```
def logf0_statistics(f0s).
# Use mask log to ignore invalid or incorrect values
log_f0s_concatenated = np.ma.log(np.concatenate(f0s))
log_f0s_mean = log_f0s_concatenated.mean() # mean calculation
log_f0s_std = log_f0s_concatenated.std() # standard deviation
calculation
return log_f0s_mean, log_f0s_std
```

For the spectrum features after dimensionality reduction, we need to further normalize them. For each dimension of the spectrum feature (36 dimensions by default), we calculate the mean and standard deviation in turn, and then subtract the mean and divide the standard deviation of the spectrum feature to finally obtain the normalized feature, which is also the input of the neural network, that is, CycleGAN learns the mapping relationship between the spectral features of the source domain and the target domain. The coded_sps_normalization_fit_transform () function in the preprocess.py implements the above operation, as shown in the code below:

```
def coded_sps_normalization_fit_transform(coded_sps).
coded_sps_concatenated = np.concatenate(coded_sps, axis=1)
# Calculate the mean value of spectral features
coded_sps_mean = np.mean(coded_sps_concatenated, axis=1,
keepdims=True)
# Calculate the standard deviation of spectral features
```

```
coded_sps_std = np.std(coded_sps_concatenated, axis=1,
keepdims=True)
coded_sps_normalized = list()
for coded_sp in coded_sps.
coded_sps_normalized.append(
(coded_sp - coded_sps_mean) / coded_sps_std) # Normalization
operation
return coded_sps_normalized, coded_sps_mean, coded_sps_std
```

The final part of the preprocessing requires saving some extracted parameters to the cache folder. cycleGAN-VC2 by default stores the mean and variance of the fundamental frequencies in the target and source domains in cache/logf0s_normalization.npz, the mean and variance of the spectral features in the target and source domains in cache/mcep_ normalization.npz, store the normalized source domain spectral features in cache/coded_sps_A_norm.pickle, and store the normalized target domain spectral features in cache/coded_sps_B_norm.pickle. The first two items here are used for speech synthesis during testing, and the last two items are used for training of cycleGAN.

12.2.3.2 Dataset Construction

CycleGAN-VC2 constructs a training dataset trainingDataset() class in the trainingDataset.py file, and since the __getitem__() method in this class is slightly tedious, we introduce it here slightly. In each selection of the index training sample, we first disorder the samples in the source and target domains, and then randomly select 128 consecutive frames for each spectral feature in the source and target domains, temporarily discarding the other frames, so that each sample is represented by only 36 × 128. The tensor in the new source and target domains is finally chosen as the index tensor, respectively. Note that the above process of disrupting, randomly selecting frames, and selecting training samples always has to be repeated once each time the training samples are selected according to the index perhaps. The code of the __getitem__() method is shown below:

```
def __getitem__(self, index).
dataset_A = self.datasetA
dataset_B = self.datasetB
n_frames = self.n_frames # length of consecutive frames
self.length = min(len(dataset_A), len(dataset_B))
num_samples = min(len(dataset_A), len(dataset_B)) # take the
minimum length to ensure alignment
train_data_A_idx = np.range(len(dataset_A))
train_data_B_idx = np.range(len(dataset_B))
np.random.shuffle(train_data_A_idx)
np.random.shuffle(train_data_B_idx)
train_data_A_idx_subset = train_data_A_idx[:num_samples] # randomly
disrupt the source domain samples
train_data_B_idx_subset = train_data_B_idx[:num_samples] # randomly
disrupt target domain samples
train_data_A = list()
train_data_B = list()

for idx_A, idx_B in zip(train_data_A_idx_subset,
```

```
train_data_B_idx_subset).
data_A = dataset_A[idx_A]
frames_A_total = data_A.shape[1]
assert frames_A_total >= n_frames
# Randomly select the starting point of the number of frames in the
target domain sample
start_A = np.random.randint(frames_A_total - n_frames + 1)
end_A = start_A + n_frames
train_data_A.append(data_A[:, start_A:end_A]) # Construct a new
source domain sample

data_B = dataset_B[idx_B]
frames_B_total = data_B.shape[1]
assert frames_B_total >= n_frames
# Randomly select the starting point of the number of frames in the
target domain sample
start_B = np.random.randint(frames_B_total - n_frames + 1)
end_B = start_B + n_frames
train_data_B.append(data_B[:, start_B:end_B]) # Construct a new
sample of the target domain

train_data_A = np.array(train_data_A)
train_data_B = np.array(train_data_B)
return train_data_A[index], train_data_B[index] # return samples
according to index
```

12.2.3.3 Generator

The generator is the core network of speech conversion, and its main function is to realize the mapping of samples between the source and target domains. According to the above, both the input and output samples of the generator are 36 × 128 There are two generators in cycleGAN, and to facilitate the distinction, we use A to denote the source domain and B to denote the target domain, where G_{A2B} accepts the source domain samples as input and outputs the target domain samples, while G_{B2A} accepts the target domain samples as input and outputs the source domain samples. These two generators have exactly the same network structure and are both generated by instantiating the Generator() class in model_tf.py.

The main processing operation of the generator is the convolution operation, and we will introduce the structure of the generator in detail. The network structure of the generator is shown in Fig. 12.4, and the main process is dimensionality reduction—conversion to 1D features—residual layer—conversion back to 2D features—-up-dimensional. In the default configuration of CycleGAN-VC2, the sample first passes through the gated convolution layer, where the first convolution operation is used for 2D convolution, the second convolution operation and the sigmoid activation function limit the result to the range of 0–1 for gating, and then the convolution and gating results need to be multiplied at the corresponding positions, at which time the sample dimension changes to 128 × 36 × 128. Next, two consecutive downsampling operations are performed. The downsampling operation still uses a similar gating mechanism, but it adds instance regularization (IN) after the two-dimensional convolution operation. Unlike batch regularization, instance regularization does not establish links between instances, maintaining independence between each sample instance. Practice shows that strength regularization works very well in tasks such as generative modeling and

especially style migration. In the downSample_Generator() class of model_tf.py, the downsampling network is implemented, and the main way to achieve dimensionality reduction is to set the step size of the convolution operation to 2, but the number of convolutions is set to 256, so the dimensionality of the samples is reduced after downsampling to 256 × 9 × 32 The code is shown below:

```
class downSample_Generator(nn.Module).
def __init__(self, in_channels, out_channels, kernel_size, stride,
padding).
super(downSample_Generator, self). __init__()
# Convolutional operations
self.convLayer = nn.Sequential(nn.Conv2d(in_channels=in_channels.
out_channels=out_channels.
kernel_size=kernel_size.
stride=stride.
padding=padding).
nn.InstanceNorm2d(num_features=out_channels.
affine=True))
# Gated convolutional operations
self.convLayer_gates =
nn.Sequential(nn.Conv2d(in_channels=in_channels.
out_channels=out_channels.
kernel_size=kernel_size.
stride=stride.
padding=padding), nn.InstanceNorm2d(num_features=out_channels.
affine=True))
```

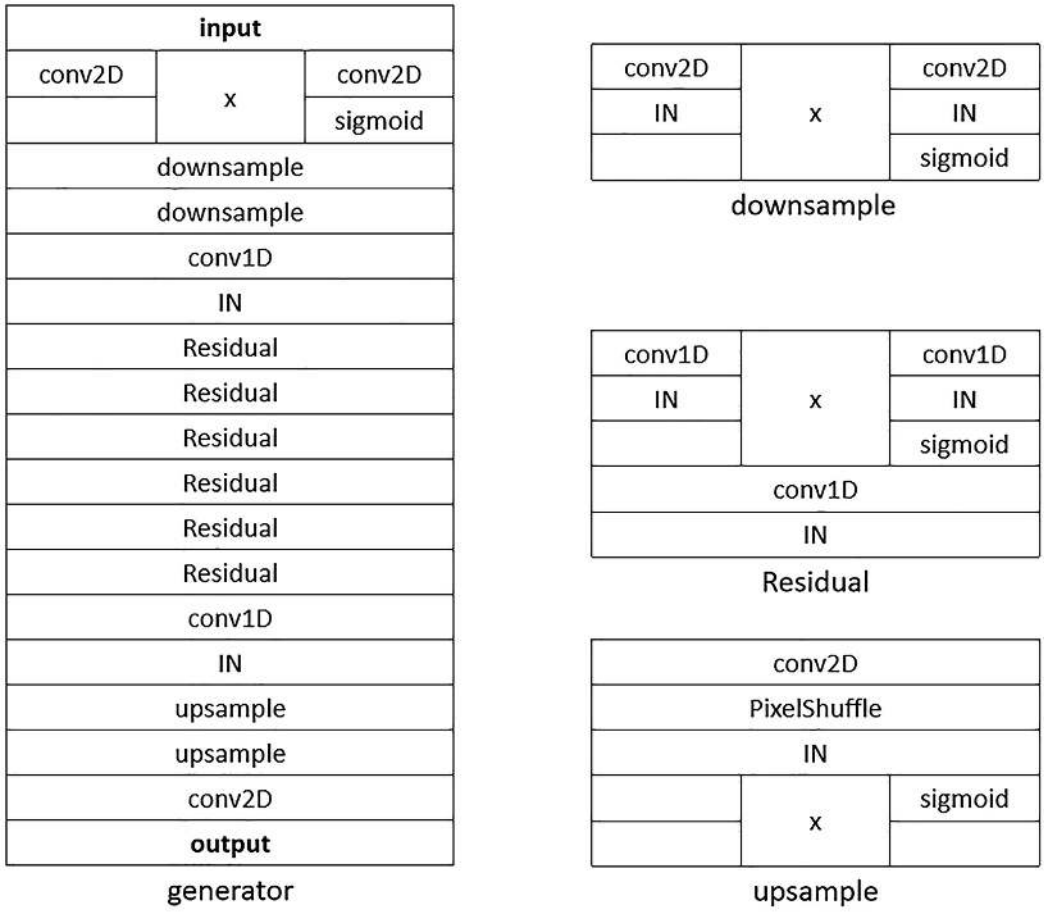


Fig. 12.4 Generator network structure

```
def forward(self, input).
return self.convLayer(input) *
torch.sigmoid(self.convLayer_gates(input))
```

In addition, the gating mechanism is implemented by class GLU(), whose code is as follows:

```
class GLU(nn.Module).
def __init__(self).
super(GLU, self). __init__()
def forward(self, input).
return input * torch.sigmoid(input)
```

Next, CycleGAN-VC2 converts the sample into one-dimensional data and processes it using a one-dimensional convolution operation. This is because one-dimensional convolution is more likely to capture the overall dynamic changes, while two-dimensional convolution is more suitable for maintaining the original structure, so at this point the sample dimension is converted to 2304×32 . Then a 1D convolutional network and a batch regularization layer are used to reduce the dimensionality to 256×32 . The generator uses six consecutive residual layers here. A gating mechanism is used in the residual layers, but all are one-dimensional convolutional operations, and the instance regularization is also one-dimensional. Compared to the downsampling layer, the residual layer adds an additional one-dimensional convolution

layer and strength regularization layer after the gating mechanism, and the residual layer keeps the dimensionality of the samples unchanged. The residual layer is implemented in the ResidualLayer() class of model_tf.py, and its structure is basically similar, so the code is not shown here.

After the residual layer, the generator starts to recover the samples to the dimensionality of the original input. The samples are first converted using one-dimensional convolution and reshape to 256 × 9 × 32 and then the corresponding upsampling operation is used. Note that this operation keeps the length and width dimensions of the sample features unchanged, but the convolution kernels of the two upsamples are 1024 and 512, and then the pytorch comes with the nn.PixelShuffle() function to complete the dimensional expansion of the feature map, where upscale_factor is the upsampling factor. The upSample() method of the Generator() class implements the upsampling process, and the code is shown below:

```
def upSample(self, in_channels, out_channels, kernel_size, stride,
padding).
self.convLayer = nn.Sequential(nn.Conv2d(in_channels=in_channels.
out_channels=out_channels.
kernel_size=kernel_size.
stride=stride.
padding=padding).
nn.PixelShuffle(upscale_factor=2), # upsampling function
nn.InstanceNorm2d(
num_features=out_channels // 4.
affine=True), # Instance regularization function
GLU()) # Gating mechanism layer
```

After upsampling, the sample size is 128 × 36 × 128 and then a two-dimensional convolutional neural network to adjust its feature dimension 1 × 1 × 36 × 128. The final dimensionality adjustment is performed to obtain a 36 × 128 output sample.

12.2.3.4 Discriminator

The main task of the discriminator is to discern whether a sample belongs to the target or source domain. There are two discriminators in CycleGAN, D_A to determine whether the input target belongs to the source domain, and D_B to determine whether the input target belongs to the destination domain, and both discriminators also have the exact same network structure. The Discriminator() class in the model_tf.py implements the discriminator in code.

The discriminator accepts as input 36 × 128 the spectral feature tensor, but its output is a tensor of smaller size, so the discriminator is mainly composed of downsampling layers, whose structure is shown in Fig. 12.5. In the default configuration of CycleGAN-VC2, the samples are first passed through a two-dimensional convolution layer to boost the number of channels to 128, i.e., the dimensionality of the samples is 128 × 36 × 128. After the gating operation, the sample dimension remains unchanged. Then the sample is downsampled three times in succession, where the downsampling layer consists of a two-dimensional convolutional layer, an instance regularization layer, a gating layer with 256, 512, and 1024 convolutional kernels in order, and the convolutional step size is 2 × 2. The dimensionality of the sample is controlled by the convolutional layer, and after each downsampling layer, the dimensionality of the sample is 256 × 18 × 64 and 512 × 9 × 32 and 1024 × 5 × 16. The final two-dimensional convolutional layer has a convolutional kernel of 1, so the number of channels is 1. The width and height of the sample remain unchanged, and the subsequent sigmoid function maps it to the interval

from 0 to 1, so the discriminator output is a 5 × 16 tensor. It should be noted that common discriminators usually end up with a fully connected layer and end up with 1 scalar value, while the discriminator here ends up with a convolution and sigmoid layer and outputs 80 values, each of which indicates the true extent of each block in the sample.

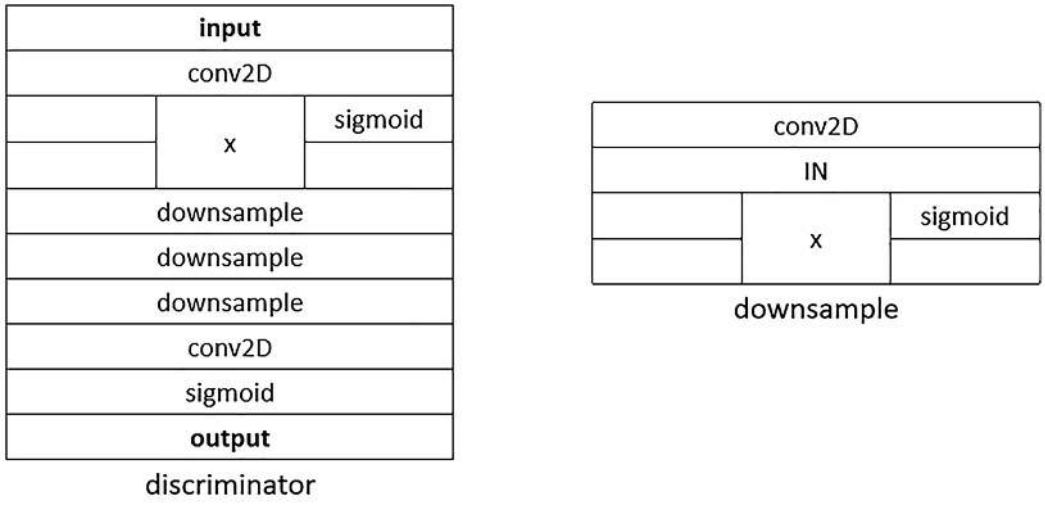


Fig. 12.5 Discriminator network structure

12.2.4 cycleGAN-VC Training

Before training CycleGAN-VC2, we will introduce the loss function in more detail in order to give the reader a better understanding of cycleGAN. cycleGAN-VC2 has two discriminators and two generators with the same network structure and their functions have a symmetric relationship. The loss function of the generators is composed of three parts.

1. G_{A2B} expects the output target domain samples to be real enough to be able to spoof the source domain discriminator D_A, while G_{B2A} also expects the output target domain samples to be real enough to be able to spoof the target domain discriminator D_B, so the adversarial loss should be set like a traditional GAN:

$$\min \mathbb{E}_{x\sim A}[D_B(G_{A2B}(x)) - 1]^2 + \mathbb{E}_{x\sim B}[D_A(G_{B2A}(x)) - 1]^2 \tag{12.2}$$

   It should be noted that since the discriminator outputs non-scalar values, and pytorch can only reverse the derivative for scalar values, the actual code should also include the mean value function, which is not reflected in above object function.

2. In order to maintain semantic consistency, CycleGAN-VC2 designed a cyclic consistent loss function, that is, the input samples of the source domain should be the same as the original input samples after passing through G_{A2B} and G_{B2A} in turn, and the output samples of the input samples in the target domain should be the same as the original input samples after passing through G_{B2A} and G_{A2B} in turn, and the expression is

$$\min \mathbb{E}_{x\sim A}[\| G_{B2A}(G_{A2B}(x)) - x \|_1] + \mathbb{E}_{x\sim B}[\| G_{A2B}(G_{B2A}(x)) - x \|_1] \tag{12.3}$$

3. In CycleGAN-VC2, the identity mapping loss function can limit the model from changing the samples too much, making the output result more stable, that is, the input samples of the source domain should be the same as the original input samples after passing through

source domain should be the same as the original input samples after passing through

G_{B2A}, and the output samples of the input samples of the target domain should be the same as the original input samples after passing through G_{A2B} in turn, and the expression is:

$$\min \mathbb{E}_{x\sim A}[\| G_{B2A}(x) - x \|_1] + \mathbb{E}_{x\sim B}[\| G_{A2B}(x) - x \|_1] \tag{12.4}$$

The loss function of the generator of CycleGAN-VC2 consists of the above three components and the weight of each component is 1. The code of the calculation process is shown below:

```
fake_B = self.generator_A2B(real_A)
cycle_A = self.generator_B2A(fake_B)
fake_A = self.generator_B2A(real_B)
cycle_B = self.generator_A2B(fake_A)
identity_A = self.generator_B2A(real_A)
identity_B = self.generator_A2B(real_B)
d_fake_A = self.discriminator_A(fake_A)
d_fake_B = self.discriminator_B(fake_B)
# Loop Consistent Loss
cycleLoss = torch.mean(torch.abs(real_A - cycle_A)) +
torch.mean(torch.abs(real_B - cycle_B))
# Constant mapping loss
identiyLoss = torch.mean(torch.abs(real_A - identity_A)) +
torch.mean(torch.abs(real_B - identity_B))
# Fighting loss
generator_loss_A2B = torch.mean((1 - d_fake_B) ** 2)
generator_loss_B2A = torch.mean((1 - d_fake_A) ** 2)

# Total loss function
generator_loss = generator_loss_A2B + generator_loss_B2A + \
cycle_loss_lambda * cycleLoss + identity_loss_lambda * identiyLoss
```

For the discriminator of CycleGAN-VC2, its loss function also consists of three parts.

1. For real source domain samples, D_A should make its output scalar close to the value 1, and similarly, for real target domain samples, D_B should make its output scalar close to the value 1, i.e.,

$$\min \mathbb{E}_{x\sim A}[D_A(x) - 1]^2 + \mathbb{E}_{x\sim B}[D_B(x) - 1]^2 \tag{12.5}$$

2. For the samples converted by the generator G_{A2B}, the D_B should treat them as spurious samples, so the scalar of its output should be close to 0; similarly, for the samples converted by the generator G_{B2A}, D_A should treat them as spurious samples, so the scalar of its output should be close to 0, then the objective function is

$$\min \mathbb{E}_{x\sim A}[D_B(G_{A2B}(x))]^2 + \mathbb{E}_{x\sim B}[D_A(G_{B2A}(x))]^2 \tag{12.6}$$

3. D_A should treat samples that are cyclic after G_{A2B} and G_{B2A} as shams, and similarly, DB

should treat samples that are cyclic after G_{B2A} and G_{A2B} as sham samples, then the objective function is

$$\min \mathbb{E}_{x\sim A}[D_A(G_{B2A}(G_{A2B}(x)))]^2 + \mathbb{E}_{x\sim B}[D_B(G_{A2B}(G_{B2A}(x)))]^2 \tag{12.7}$$

For these three parts of the loss function, CycleGAN-VC2 gives weight ratios of 1, 0.5, and 0.5, respectively, and the code of its calculation process is shown below:

```
d_real_A = self.discriminator_A(real_A)
d_real_B = self.discriminator_B(real_B)
generated_A = self.generator_B2A(real_B)
d_fake_A = self.discriminator_A(generated_A)
cycled_B = self.generator_A2B(generated_A)
d_cycled_B = self.discriminator_B(cycled_B)
generated_B = self.generator_A2B(real_A)
d_fake_B = self.discriminator_B(generated_B)
cycled_A = self.generator_B2A(generated_B)
d_cycled_A = self.discriminator_A(cycled_A)

d_loss_A_real = torch.mean((1 - d_real_A) ** 2) # True sample loss
of D A
d_loss_A_fake = torch.mean((0 - d_fake_A) ** 2) # False sample loss
of D A
d_loss_A = (d_loss_A_real + d_loss_A_fake) / 2.0

d_loss_B_real = torch.mean((1 - d_real_B) ** 2) # True sample loss
of D B
d_loss_B_fake = torch.mean((0 - d_fake_B) ** 2) # False sample loss
of D B
d_loss_B = (d_loss_B_real + d_loss_B_fake) / 2.0

d_loss_A_cycled = torch.mean((0 - d_cycled_A) ** 2) # Cyclic sample
loss of D A
d_loss_B_cycled = torch.mean((0 - d_cycled_B) ** 2) # Cycled sample
loss of D B
d_loss_A_2nd = (d_loss_A_real + d_loss_A_cycled) / 2.0
d_loss_B_2nd = (d_loss_B_real + d_loss_B_cycled) / 2.0

# All loss functions
d_loss = (d_loss_A + d_loss_B) / 2.0 + (d_loss_A_2nd +
d_loss_B_2nd) / 2.0
```

CycleGAN-VC2 uses the Adam optimization algorithm for both the discriminator and the generator, with a learning rate of 0.0002 for the generator and 0.0001 for the discriminator. The parameters β are 0.5 and 0.999, respectively.

Before training CycleGAN-VC2, we need to run the preprocess.py file for data preprocessing

```
python preprocess_training.py --train_A_dir . /data/S0913/ --
```

```
train_B_dir . /data/gaoxiaosong/ --cache_folder . /cache/
# Or set the parameters within the program and the following
command line
python preprocess_training.py
```

After that, run train.py to start the training of the model.

```
Python train.py --logf0s_normalization .
/cache/logf0s_normalization.npz --mcep_normalization .
/cache/mcep_normalization.npz --coded_sps_A_norm .
/cache/coded_sps_A_norm.pickle -- coded_sps_B_norm .
/cache/coded_sps_B_norm.pickle --model_checkpoint .
/model_checkpoint/ --resume_training_at .
/model_checkpoint/_CycleGAN_CheckPoint --validation_A_dir .
/data/S0913/ --output_A_dir . /converted_sound/S0913 --
validation_B_dir . /data/gaoxiaosong/ --output_B_dir .
/converted_sound/gaoxiaosong/
# Or set the parameters within the program and the following
command line
python train.py
```

12.2.5 cycleGAN-VC Model Testing

After completing the model training you can start testing the model, either by using your own trained model with weights saved in the model_checkpoint folder in the home directory or by using the completed trained model available on the network with the link https://drive.google.com/file/d/1iamizL98NWIPw4pw0nF-7b6eoBJrxEfj/view?usp=sharing. cycleGAN-VC performs a model test every 200 rounds of training, and the reader can copy it and make simple path modifications to build the test model.

When conducting model testing, for example, for source domain speech signals, the WORLD tool is first used to calculate fundamental frequency f_0, spectral parameters *sp*, and acyclic parameterss *ap*. The fundamental frequency f_0 is then transformed to the target domain:

$$f_{tar} = (f_0 - m_{src})\frac{s_{tar}}{s_{src}} + m_{tar} \tag{12.8}$$

pitch_conversion() function in the preprocess.py implements it. The spectral parameter sp is then encoded into a 36-dimensional eigenvector using WORLD, where the world_encode_spectral _envelop() function is called again. According to the mean and variance parameters of the preprocessed source domain eigenvectors, the encoded eigenvectors were normalized and used as the input samples for G_{A2B}. For the encoded samples output by the generator G_{A2B}, the reverse normalization operation is performed according to the mean and variance parameters of the eigenvector of the target domain, and the decoding operation is performed by WORLD to obtain the spectral parameter sp. The world_decode_spectral_envelop() function in the preprocess.py implements the decoding operation as follows:

```
def world_decode_spectral_envelop(coded_sp, fs).
fftlen = pyworld.get_cheaptrick_fft_size(fs)
decoded_sp = pyworld.decode_spectral_envelope(coded_sp, fs, fftlen)
```

```
return decoded_sp
```

The third parameter acyclic parameters *ap* remains unchanged, and finally the synthesis of the target domain speech is completed using the synthesize tool of WORLD. The conversion process from the target domain to the source domain is in the same order of operation as above, so the reader can try it by himself. Finally, at http://www.kecl.ntt.co.jp/people/kaneko.takuhiro/projects/cyclegan-vc2/index.html, the reader can see the actual results of the speech style conversion.

12.3 GAN-Based Speech Generation

In this section, we talk about using GAN to complete a speech generation project. Through this section, we will experience the powerful generative power of GAN to convert noise into clear speech, and learn the details of waveGAN's model and how to use it.

12.3.1 Project Introduction

Unlike the general speech synthesis task (TTS), which generates speech signal sequences from text sequences, the speech generation task accepts random noise as input and generates realistic speech signals. The speech generation task is a direct test of the generation capability of GAN, which shows that GAN can generate not only data such as images and videos, but also model one-dimensional time-series data such as speech.

This subsection will explain the network structure, training, and testing methods of waveGAN [4] in detail, and readers can implement it by themselves according to the explanation.

We use the waveGAN open source code link: https://github.com/mazzzystar/WaveGAN-pytorch. The code includes several folders and python program files under it, which we briefly introduce separately. sc09 file is the training dataset we use to run the program, the samples in this dataset are the sc09 file which contains three datasets: train, test, and valid, where the train subfile contains 18620 speech signals, test contains 2552 speech items, and valid contains 2494 speech items. It should be noted that for the speech generation task, there is no distinction between training set, test set, and validation set, and we treat them all as training samples. For the file name of any sample, e.g., Eight_00b01445_nohash_0.wav, Eight means the sample is the voice of English Eight, 00b01445 means a different speaker code, and the final 0 means the number of times the voice is repeated. In addition to the sc09 dataset, we can also use the piano dataset, whose samples are different instrument sound effects, and whose download link is http://deepyeti.ucsd.edu/cdonahue/mancini_piano.tar.gz, readers can try it by themselves; the output folder is the final output sample of the model, related setting parameters, model weight parameters, etc. The file name is the time of training; the imgs file includes the network structure graph archi.png and the loss function curve loss_curve.png.

config.py sets parameters such as epoch and batch_size; logger.py sets logging, which is used to record the value of the loss function in the training process and saves the contents of the log in the model.log file in the main directory; utils.py integrates various functions needed for training the model, such as sample wavegan.py implements the generator and discriminator using torch; train.py is the core code for training the model, including building the model, scheduling training data, generating samples, and other functions, which needs to be explained in detail.

12.3.2 waveGAN Model

The waveGAN consists of two models, the generator and the discriminator, where the generator takes random noise as input and outputs the generated speech samples, and the discriminator accepts the generated speech samples or the speech samples from the training set and makes the true/false judgments. This project deals with short duration speech signals, which can be completely transformed into fixed-length samples for processing. The training makes the speech samples generated by the generator very realistic in terms of hearing.

12.3.2.1 Data Preprocessing

We first explain the code of the data processing part. For each speech sample, the sampling rate is firstly adjusted to 16 kHz, and then a window of length 16,384 is set. If the length of the sample is less than the window length, a complementary 0 is performed symmetrically at both ends of the sample to make its length reach 16,384; if the length of the sample exceeds 16,384, a continuous segment of speech signal of length 16384 is randomly selected as a sample in the sample. In addition, waveGAN also performs a normalization operation to keep the value domain of the samples within [−1, 1]. The preprocessing of the data is implemented in the sample_generator() function of utils.py. The code for the final part of the function, which continuously loops through while and throws samples within each loop using yield, is shown below:

```
def sample_generator(filepath, window_length=16384, fs=16000).
try.
audio_data, _ = librosa.load(filepath, sr=fs)
# Normalization operations
max_mag = np.max(np.abs(audio_data))
if max_mag > 1.
audio_data /= max_mag
except Exception as e.
LOGGER.error("Could not load {}: {}".format(filepath, str(e))))
raise StopIteration
audio_len = len(audio_data)
if audio_len < window_length.
pad_length = window_length - audio_len
left_pad = pad_length // 2
right_pad = pad_length - left_pad
audio_data = np.pad(audio_data, (left_pad, right_pad),
mode='constant') # Symmetric zero complement
audio_len = len(audio_data)

while True.
if audio_len == window_length.
sample = audio_data # Directly as a sample
else: # Randomly select a segment of the signal as a sample
start_idx = np.random.randint(0, (audio_len - window_length) // 2)
end_idx = start_idx + window_length
sample = audio_data[start_idx:end_idx]
sample = sample.astype('float32')
assert not np.any(np.isnan(sample))
yield {'X': sample} # work with next as data reader
```

12.3.2.2 Generator

The generator is mainly responsible for mapping the noise to speech samples, which are uniformly distributed noise with values in the range [−1, 1] and a dimension of 100. Since the dimension of the output sample is 16,384, it needs to go through multiple layers of deconvolution (transpose convolution) to enhance the dimensionality, and the network structure of the generator is shown in Fig. 12.6.

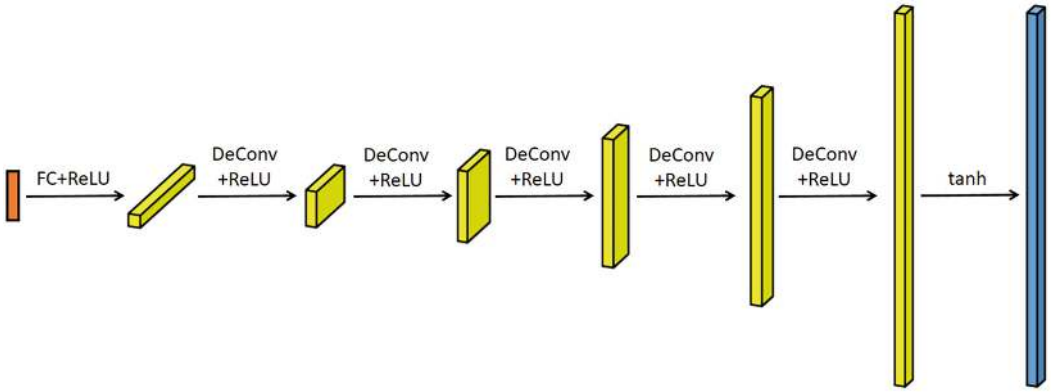


Fig. 12.6 waveGAN generator network structure

The 100-dimensional noise passes through the first layer of the fully connected network to obtain a tensor with a channel number of 1024 and a length of 16. In the next five layers of deconvolution, the number of channels is successively reduced to 512, 256, 128, 64, and 1, and the sequence length is successively increased to 64, 256, 1024, 4096, and 16,384, and finally a 16,384-dimensional sample is output after the tanh activation function. For the deconvolutional network and fully connected layers in it, waveGAN uses kaiming_normal for weight initialization by default. In wavegan.py, class WaveGANGenerator() implements the generator, where model_size is used to control the size of the model, the default is 64, and the reader can also choose according to the needs; num_channels is the number of channels of the output samples, usually set to 1; latent_dim is the dimension of the input noise, the default is 100. The code is shown below (we have omitted the unnecessary parts, which may be different from the github source code):

```
class WaveGANGenerator(nn.Module).
def __init__(self, model_size=64, ngpus=1, num_channels=1.
latent_dim=100, post_proc_filt_len=512.
verbose=False, upsample=True).
super(WaveGANGenerator, self). __init__()
self.ngpus = ngpus
self.model_size = model_size
self.num_channels = num_channels
self.latent_di = latent_dim
self.post_proc_filt_len = post_proc_filt_len
self.verbose = verbose
self.fc1 = nn.Linear(latent_dim, 256 * model_size)
stride = 4
if upsample.
stride = 1
upsample = 4
```

```
self.deconv_1 = Transpose1dLayer(16*model_size, 8*model_size, 25,
stride, upsample =upsample)
self.deconv_2 = Transpose1dLayer(8*model_size, 4*model_size, 25,
stride, upsample =upsample)
self.deconv_3 = Transpose1dLayer(4*model_size, 2*model_size, 25,
stride, upsample =upsample)
self.deconv_4 = Transpose1dLayer(2*model_size, model_size, 25,
stride, upsample =upsample)
self.deconv_5 = Transpose1dLayer(model_size, num_channels, 25,
stride, upsample=upsample)

# Weights initialization
for m in self.modules().
if isinstance(m, nn.ConvTranspose1d) or isinstance(m, nn.Linear).
nn.init.kaiming_normal(m.weight.data)

def forward(self, x).
x = self.fc1(x).view(-1, 16 * self.model_size, 16)
x = F.relu(x)
x = F.relu(self.deconv_1(x))
x = F.relu(self.deconv_2(x))
x = F.relu(self.deconv_3(x))
x = F.relu(self.deconv_4(x))
output = F.tanh(self.deconv_5(x))
return output
```

In the deconvolution layer, the last dimension of the feature map is first boosted by a factor of 4 using the upsampling operation, and then symmetrically complemented by 0 on both sides of the feature map according to the size of the convolution kernel, and finally using the one-dimensional convolution operation, where the default size of the convolution kernel is 25 and the step size is 1. Also in wavegan.py, the class Transpose1dLayer() implements the deconvolution function with code is shown as follows:

```
class Transpose1dLayer(nn.Module).
def __init__(self, in_channels, out_channels, kernel_size, stride,
padding=11, upsample=None, output_padding=1).
super(Transpose1dLayer, self). __init__()
self.upsample = upsample

self.upsample_layer = torch.nn.Upsample(scale_factor=upsample)
#upsample
reflection_pad = kernel_size // 2
self.reflection_pad = nn.ConstantPad1d(reflection_pad, value=0) #
both sides complemented by 0
self.conv1d = torch.nn.Conv1d(in_channels, out_channels,
kernel_size, stride)
self.Conv1dTrans = nn.ConvTranspose1d(in_channels, out_channels,
kernel_size, stride, padding, output_padding)

def forward(self, x).
```

```
if self.upsample.
return self.conv1d(self.reflection_pad(self.upsample_layer(x)))
else.
return self.Conv1dTrans(x)
```

12.3.2.3 Discriminator

The main structure of the discriminator is a five-layer 1D convolutional neural network and a one-layer fully connected neural network, which is responsible for distinguishing the true from false samples, and its structure is shown in Fig. 12.7. waveGAN uses the loss function in WGAN, so the discriminator should output the lowest possible value for the generated samples and the highest possible value for the samples from the training set.

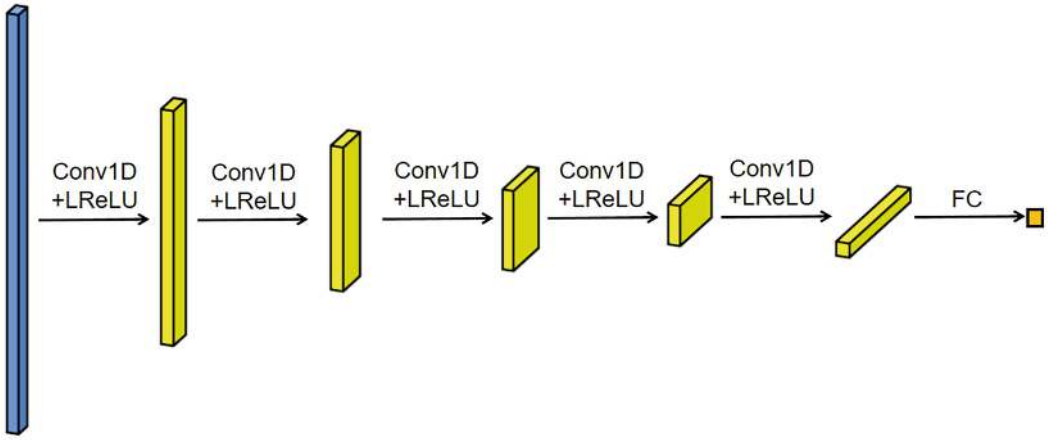


Fig. 12.7 Discriminator network structure

For an input sample of 18,364, after five layers of 1D convolution, the number of channels are 64, 128, 256, 512, and 1024, while the sequence length is reduced to 4096, 1024, 256, 64, and 16 in turn, and finally the convolved tensor is straightened to a one-dimensional vector and the final scalar value is output by the fully connected network. Similar to the generator, waveGAN also uses kaiming_normal for weight initialization by default for its deconvolution network and fully connected layers. After each layer undergoes a 1D convolution operation, it also goes through a LeakyReLu activation function and a phase shift layer, which is mainly for optimizing the training of the discriminator. The phase shift layer performs a shift operation on the samples, where the last dimension (the dimension indicating the length of the sequence) is randomly shifted to the left or right, and the vacated positions are mirrored and filled, as shown in Fig. 12.8. The phase shift layer is implemented by the PhaseShuffle() class, and the shift magnitude is determined by the shift_factor parameter, whose code is shown below:

```
class PhaseShuffle(nn.Module).
def __init__(self, shift_factor).
super(PhaseShuffle, self). __init__()
self.shift_factor = shift_factor # the magnitude of the random
shift
```

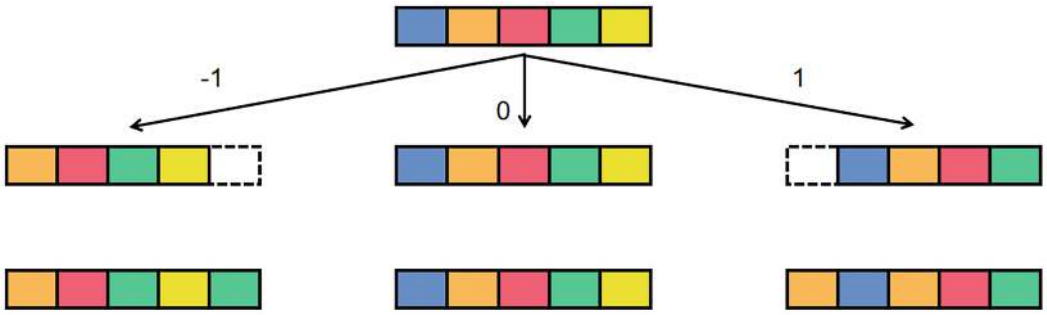


Fig. 12.8 Schematic diagram of the mirror phase shift

```
def forward(self, x).
if self.shift_factor == 0.
return x
# Randomly generated shifts for each sample within the shift range
k_list = torch.Tensor(x.shape[0]).random_(0, 2 * self.shift_factor
+ 1) - self.shift_factor
k_list = k_list.numpy().astype(int)

k_map = {}
for idx, k in enumerate(k_list).
k = int(k)
if k not in k_map.
k_map[k] = []
k_map[k].append(idx)
x_shuffle = x.clone()
for k, idxs in k_map.items().
if k > 0.
x_shuffle[idxs] = F.pad(x[idxs][... , :-k], (k, 0), mode='reflect')
else.
x_shuffle[idxs] = F.pad(x[idxs][... , -k:], (0, -k),
mode='reflect')
assert x_shuffle.shape == x.shape, "{}, {}".format(x_shuffle.shape,
x.shape)
return x_shuffle
```

The entire network of discriminators is relatively easy to implement in wavegan.py using the class WaveGANDiscriminator(), where model_size is the scale size of the network; num_channels is the number of sample channels, default is 1; alpha is the negative semiaxis slope of the LeakyReLU activation function. The core code is shown as follows:

```
class WaveGANDiscriminator(nn.Module).
def __init__(self, model_size=64, ngpus=1, num_channels=1,
shift_factor=2.
alpha=0.2, verbose=False).
super(WaveGANDiscriminator, self). __init__()
self.model_size = model_size # d
self.ngpus = ngpus
self.num_channels = num_channels # c
```

```
self.shift_factor = shift_factor # n
self.alpha = alpha
self.verbose = verbose
# Convolutional layers
self.conv1 = nn.Conv1d(num_channels, model_size, 25, stride=4,
padding=11)
self.conv2 = nn.Conv1d(model_size, 2 * model_size, 25, stride=4,
padding=11)
self.conv3 = nn.Conv1d(2 * model_size, 4 * model_size, 25,
stride=4, padding=11)
self.conv4 = nn.Conv1d(4 * model_size, 8 * model_size, 25,
stride=4, padding=11)
self.conv5 = nn.Conv1d(8 * model_size, 16 * model_size, 25,
stride=4, padding=11)
# Phase shift layer
self.ps1 = PhaseShuffle(shift_factor)
self.ps2 = PhaseShuffle(shift_factor)
self.ps3 = PhaseShuffle(shift_factor)
self.ps4 = PhaseShuffle(shift_factor)

self.fc1 = nn.Linear(256 * model_size, 1)
# Weight initialization
for m in self.modules().
if isinstance(m, nn.Conv1d) or isinstance(m, nn.Linear).
nn.init.kaiming_normal(m.weight.data)

def forward(self, x).
x = F.leaky_relu(self.conv1(x), negative_slope=self.alpha)
x = self.ps1(x)
x = F.leaky_relu(self.conv2(x), negative_slope=self.alpha)
x = self.ps2(x)
x = F.leaky_relu(self.conv3(x), negative_slope=self.alpha)
x = self.ps3(x)
x = F.leaky_relu(self.conv4(x), negative_slope=self.alpha)
x = self.ps4(x)
x = F.leaky_relu(self.conv5(x), negative_slope=self.alpha)
x = x.view(-1, 256 * self.model_size)
return self.fc1(x)
```

In training the discriminator, we use the loss function of WGAN-GP, so we need to solve the gradient of the discriminator output to the input in the gradient penalty regular term, and we explain this process. First, we use the linear interpolation method to construct the penalty samples using real and spurious samples and use the autograd function in pytorch to solve the derivatives, where we need to set the retain_graph parameter to True to keep the computational graph of the gradient penalty, and we also need to set only_inputs to True to In addition, the dimension of the gradient should be exactly the same as the dimension of the penalized sample, so the grad_outputs parameter should have an all-1, consistent with the dimension of the penalized sample to accept the result. The gradient penalty regular term is

implemented in the calc_gradient_penalty() function of utils.py, where net_dis is the discriminator and labda is the regular term weight, and its code is shown as follows:

```
def calc_gradient_penalty(net_dis, real_data, fake_data,
batch_size, lmbda, use_cuda=False).
# Initialize interpolation parameters
alpha = torch.rand(batch_size, 1, 1)
alpha = alpha.expand(real_data.size())
alpha = alpha.cuda() if use_cuda else alpha

# Construct penalty samples.
interpolates = alpha * real_data + (1 - alpha) * fake_data
if use_cuda.
interpolates = interpolates.cuda()
interpolates = autograd.Variable(interpolates, requires_grad=True)

disc_interpolates = net_dis(interpolates)

# Calculate the gradient of the output over the input
gradients = autograd.grad(outputs=disc_interpolates,
inputs=interpolates.
grad_outputs=torch.ones(disc_interpolates.size()).cuda()
if use_cuda else torch.ones(disc_interpolates.size()).
create_graph=True, retain_graph=True, only_inputs=True)[0]
gradients = gradients.view(gradients.size(0), -1)

# Compute regular terms.
gradient_penalty = lmbda * ((gradients.norm(2, dim=1) - 1) **
2).mean()
return gradient_penalty
```

12.3.3 waveGAN Training

Place the corresponding dataset in the main directory, set the appropriate parameters in config.py, and you can directly run train.py to start training. As of the author's writing of this part, the project has not yet implemented support for multi-card training, and readers can complete it by themselves.

```
$ python train.py
```

When training the discriminator, the training is done once using data from the training set and then once using samples from the validation set, while the generator is trained only once per round. waveGAN uses the commonly used Adam optimization algorithm, whose learning rate are 0.0001, $(\boldsymbol{\beta_1}, \boldsymbol{\beta_2})$ are (0.5, 0.9).

The related output results can be viewed in the output folder. The generated speech samples cannot be displayed visually, readers need to generate their own samples after completing the training. In addition, for the generated results of the sc09 dataset, readers can find the results at https://soundcloud.com/mazzzystar/sets/dcgan-sc09. For the generation results of the piano dataset, readers can find the results at https://soundcloud.com/mazzzystar/sets/wavegan-piano.

If you use the completed training model, you can directly use the load_state_dict() function, where filepath is the model weight path, as follows code:

```
def load_wavegan_generator(filepath, model_size=64, ngpus=1,
num_channels=1.
latent_dim=100, post_proc_filt_len=512, **kwargs).
model = WaveGANGenerator(model_size=model_size, ngpus=ngpus.
num_channels=num_channels, latent_dim=latent_dim.
post_proc_filt_len=post_proc_filt_len)
model.load_state_dict(torch.load(filepath))
return model

def load_wavegan_discriminator(filepath, model_size=64, ngpus=1,
num_channels=1.
shift_factor=2, alpha=0.2, **kwargs).
model = WaveGANDiscriminator(model_size=model_size, ngpus=ngpus.
num_channels=num_channels.
shift_factor=shift_factor, alpha=alpha)
model.load_state_dict(torch.load(filepath))
return model
```

References

1. Pascual, Santiago, Antonio Bonafonte, and Joan Serrà. SEGAN: Speech Enhancement Generative Adversarial Network. Proc. Interspeech 2017 (2017): 3642-3646.

2. Kaneko, Takuhiro, and Hirokazu Kameoka. Cyclegan-vc: Non-parallel voice conversion using cycle-consistent adversarial networks. 2018 26th European Signal Processing Conference (EUSIPCO). IEEE, 2018.

3. Kaneko, Takuhiro, et al. Cyclegan-vc2: Improved cyclegan-based non-parallel voice conversion. ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2019.

4. Donahue, Chris, Julian McAuley, and Miller Puckette. Synthesizing audio with generative adversarial networks. arXiv preprint arXiv. 1802.04208 1 (2018).