

FROM  
SCRATCH

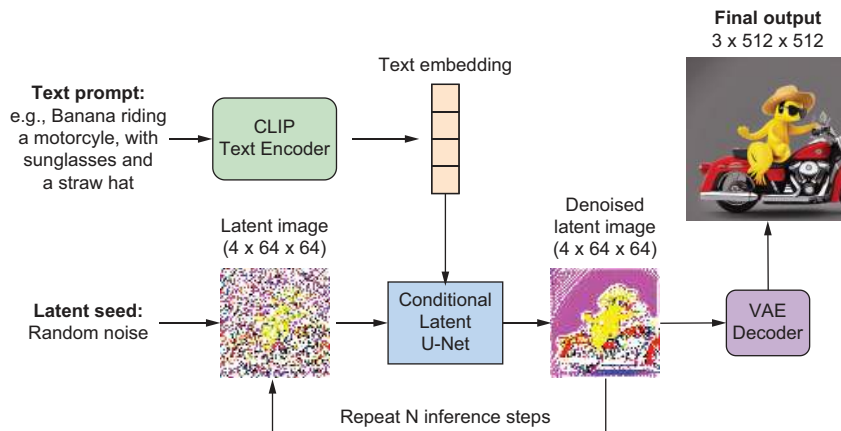
# BUILD A Text-to-Image Generator

with transformers and diffusions

Mark Liu



## How a latent diffusion model (e.g., Stable Diffusion) generates an image based on a text prompt



The text prompt shown at the top-left corner is first encoded into a text embedding. To generate an image in the lower-dimensional space (latent space), we start with an image of pure noise (bottom left). We use the trained U-Net to iteratively denoise the image, conditional on the text embedding so the generated image matches the text prompt, with the guidance of a trained CLIP model. The generated latent image (bottom right) is presented to a trained VAE to convert it to a high-resolution image, which is the final output (top right).

## *Build a Text-to-Image Generator (from Scratch)*



# *Build a Text-to-Image Generator (from Scratch)*

WITH TRANSFORMERS AND DIFFUSIONS

MARK LIU



MANNING  
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit [www.manning.com](http://www.manning.com). The publisher offers discounts on this book when ordered in quantity.

For more information, please contact

Special Sales Department  
Manning Publications Co.  
20 Baldwin Road  
PO Box 761  
Shelter Island, NY 11964  
Email: [orders@manning.com](mailto:orders@manning.com)

© 2026 Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The author and publisher have made every effort to ensure that the information in this book was correct at press time. The author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.



Manning Publications Co.  
20 Baldwin Road  
PO Box 761  
Shelter Island, NY 11964

Development editor: Doug Rudder  
Technical editor: Nathan Crocker  
Review editor: Kishor Rit  
Production editor: Kathy Rossland  
Copy editor: Julie McNamee  
Proofreader: Melody Dolab  
Typesetter: Tamara Švelić Sabljic  
Cover designer: Marija Tudor

ISBN 9781633435421  
Printed in the United States of America

*To all AI enthusiasts!*

# *brief contents*

---

## **PART 1 UNDERSTANDING ATTENTION AND TRANSFORMERS ..... 1**

- 1 ■ A tale of two models: Transformers and diffusions 3
- 2 ■ Build a transformer 22
- 3 ■ Classify images with a vision transformer 52
- 4 ■ Add captions to images 79

## **PART 2 INTRODUCTION TO DIFFUSION MODELS ..... 103**

- 5 ■ Generate images with diffusion models 105
- 6 ■ Control what images to generate in diffusion models 126
- 7 ■ Generate high-resolution images with diffusion models 152

## **PART 3 TEXT-TO-IMAGE GENERATION WITH DIFFUSION MODELS... 173**

- 8 ■ CLIP: A model to measure the similarity between image and text 175
- 9 ■ Text-to-image generation with latent diffusion 201
- 10 ■ A deep dive into Stable Diffusion 225

## **PART 4 TEXT-TO-IMAGE GENERATION WITH TRANSFORMERS.... 243**

- 11 ■ VQGAN: Convert images into sequences of integers 245
- 12 ■ A minimal implementation of DALL-E 268



**PART 5 NEW DEVELOPMENTS AND CHALLENGES..... 287**

- 13 ■ New developments and challenges in text-to-image generation 289
- appendix* ■ Installing PyTorch and enabling GPU training locally and in Colab 315

# contents

---

<i>preface</i>	<i>xv</i>
<i>acknowledgments</i>	<i>xvii</i>
<i>about this book</i>	<i>xix</i>
<i>about the author</i>	<i>xxii</i>
<i>about the cover illustration</i>	<i>xxiii</i>

## PART 1 UNDERSTANDING ATTENTION AND TRANSFORMERS ..... 1

### 1 *A tale of two models: Transformers and diffusions* 3

- 1.1 What is a text-to-image generation model? 4
  - Unimodal vs. multimodal models* 5 ■ *Practical use cases of text-to-image models* 6
- 1.2 Transformer-based text-to-image generation 7
  - Converting an image into a sequence of integers and then back* 8
  - Training and using a transformer-based text-to-image model* 10
- 1.3 Text-to-image generation with diffusion models 11
  - Forward and reverse diffusions* 12 ■ *Latent diffusion models and Stable Diffusion* 13
- 1.4 How to build text-to-image models from scratch 15

- 1.5 Challenges for text-to-image generation models 18
  - Are generative AI models stealing from artists?* 18
  - The geometric inconsistency problem* 19
- 1.6 Social, environmental, and ethical concerns 19

## 2 **Build a transformer** 22

- 2.1 An overview of attention and transformers 24
  - How the attention mechanism works* 24 ■ *How to create a transformer* 28
- 2.2 Word embedding and positional encoding 33
  - Word tokenization with the Spacy library* 34 ■ *A sequence padding function* 39 ■ *Input embedding from word embedding and positional encoding* 41
- 2.3 Creating an encoder–decoder transformer 43
  - Coding the attention mechanism* 43 ■ *Defining the Transformer() class* 44 ■ *Creating a language translator* 46
- 2.4 Training and using the German-to-English translator 47
  - Training the encoder–decoder transformer* 47
  - Translating German to English with the trained model* 48

## 3 **Classify images with a vision transformer** 52

- 3.1 The blueprint to train a ViT 54
  - Converting images to sequences* 54 ■ *Training a ViT for classification* 56
- 3.2 The CIFAR-10 dataset 57
  - Downloading and visualizing CIFAR-10 images* 58
  - Preparing datasets for training and testing* 59
- 3.3 Building a ViT from scratch 60
  - Dividing images into patches* 61 ■ *Modeling the positions of different patches in an image* 63 ■ *Using the multi-head self-attention mechanism* 64 ■ *Building an encoder-only transformer* 65 ■ *Using the ViT to create a classifier* 67
- 3.4 Training and using the ViT to classify images 68
  - Choosing the optimizer and the loss function* 69 ■ *Training the ViT for image classification* 70 ■ *Classifying images using the trained ViT* 72

## 4 *Add captions to images* 79

- 4.1 Training and using a transformer to add captions 81  
*Preparing data and the causal attention mask* 81  
*Creating and training a transformer* 84
- 4.2 Preparing the training dataset 86  
*Downloading and visualizing Flickr 8k images* 87 ■ *Building a vocabulary of tokens* 88 ■ *Preparing the training dataset* 89
- 4.3 Creating a multimodal transformer to add captions 91  
*Defining a ViT as the image encoder* 91 ■ *Creating the decoder to generate text* 94
- 4.4 Training and using the image-to-text transformer 96  
*Training the encoder–decoder transformer* 96 ■ *Adding captions to images with the trained model* 99

## PART 2 INTRODUCTION TO DIFFUSION MODELS..... 103

## 5 *Generate images with diffusion models* 105

- 5.1 The forward diffusion process 107  
*How diffusion models work* 107 ■ *Visualizing the forward diffusion process* 109 ■ *Different diffusion schedules* 111
- 5.2 The reverse diffusion process 115
- 5.3 A blueprint to train the U-Net model 117  
*Steps in training a denoising U-Net model* 117 ■ *Preprocessing the training data* 118
- 5.4 Training and using the diffusion model 120  
*The Denoising Diffusion Probabilistic Model noise scheduler* 120  
*Inference using the U-Net denoising model* 121 ■ *Training and using the denoising U-Net model* 122

## 6 *Control what images to generate in diffusion models* 126

- 6.1 Classifier-free guidance in diffusion models 128  
*An overview of classifier-free guidance* 128 ■ *A blueprint to implement CFG* 129
- 6.2 Different components of a denoising U-Net model 131  
*Time step embedding and label embedding* 132 ■ *The U-Net denoising model architecture* 135 ■ *Down blocks and up blocks in the U-Net* 137

- 6.3 Building and training the denoising U-Net model 142  
*Building the denoising U-Net 142 ■ The Denoising Diffusion Probabilistic Model 144 ■ Training the diffusion model 145*
- 6.4 Generating images with the trained diffusion model 147  
*Visualizing generated images 148 ■ How the guidance parameter affects generated images 149*

## 7 *Generate high-resolution images with diffusion models* 152

- 7.1 Attention in U-Net, DDIM, and image interpolation 154  
*Incorporating the attention mechanism in the U-Net model 154  
Denoising Diffusion Implicit Models 156 ■ Image interpolation in diffusion models 156*
- 7.2 High-resolution flower images as training data 158  
*Visualizing images in the training dataset 159  
Applying forward diffusion on flower images 161*
- 7.3 Building and training a U-Net for high-resolution images 163  
*Building the denoising U-Net model 163 ■ Training the denoising U-Net model 165*
- 7.4 Image generation and interpolation 166  
*Using the trained denoising U-Net to generate images 166  
Transition from one image to another 168*

## PART 3 TEXT-TO-IMAGE GENERATION WITH DIFFUSION MODELS ..... 173

### 8 *CLIP: A model to measure the similarity between image and text* 175

- 8.1 The CLIP model 177  
*How the CLIP model works 177 ■ Selecting an image from Flickr 8k based on a text description 178*
- 8.2 Preparing the training dataset 180  
*Image-caption pairs in Flickr 8k 180 ■ The DistilBERT tokenizer 183 ■ Preprocess captions and images for training 184*
- 8.3 Creating a CLIP model 187  
*Creating a text encoder 187 ■ Creating an image encoder 189  
Building a CLIP model 190*

- 8.4 Training and using the CLIP model 192
  - Training the CLIP model* 193 ■ *Using the trained CLIP model to select images* 195 ■ *Using the OpenAI pretrained CLIP model to select images* 197

## 9 *Text-to-image generation with latent diffusion* 201

- 9.1 What is a latent diffusion model? 203
  - How variational autoencoders work* 203 ■ *Combining a latent diffusion model with a variational autoencoder* 205
- 9.2 Compressing and reconstructing images with VAEs 207
  - Downloading the pretrained VAE* 207 ■ *Encoding and decoding images with the pretrained VAE* 209
- 9.3 Text-to-image generation with latent diffusion 213
  - Guidance by the CLIP model* 213 ■ *Diffusion in the latent space* 215 ■ *Converting latent images to high-resolution ones* 216
- 9.4 Modifying existing images with text prompts 219

## 10 *A deep dive into Stable Diffusion* 225

- 10.1 Generating images with Stable Diffusion 227
- 10.2 The Stable Diffusion architecture 230
  - Generating images from text with Stable Diffusion* 231
  - Text embedding interpolation* 232
- 10.3 Creating text embeddings 234
- 10.4 Image generation in the latent space 236
- 10.5 Converting latent images to high-resolution ones 238

## PART 4 TEXT-TO-IMAGE GENERATION WITH TRANSFORMERS..... 243

### 11 *VQGAN: Convert images into sequences of integers* 245

- 11.1 Converting images into sequences of integers and back 247
- 11.2 Variational autoencoders 249
  - What is an autoencoder?* 249 ■ *The need for VAEs and their training methodology* 251

- 11.3 Vector quantized variational autoencoders 252  
*The need for VQ-VAEs 252 ■ The VQ-VAE model architecture and training process 253 ■*
- 11.4 Vector quantized generative adversarial networks 256  
*Generative adversarial networks 256 ■ VQGAN: A GAN with a VQ-VAE generator 257*
- 11.5 A pretrained VQGAN model 259  
*Reconstructing images with the pretrained VQGAN 259  
 Converting images into sequences of integers 263*

## 12 *A minimal implementation of DALL-E* 268

- 12.1 How min-DALL-E works 269  
*Training min-DALL-E 270 ■ From prompt to pixels: Image generation at inference time 272*
- 12.2 Tokenizing and encoding the text prompt 273  
*Tokenizing the text prompt 273 ■ Encoding the text prompt 277*
- 12.3 Iterative prediction of image tokens 278  
*Loading the pretrained BART decoder 279 ■ Predicting image tokens using the BART decoder 280*
- 12.4 Converting image tokens to high-resolution images 282  
*Loading the pretrained VQGAN detokenizer 282 ■ Visualizing the intermediate and final high-resolution outputs 283*

## PART 5 NEW DEVELOPMENTS AND CHALLENGES ..... 287

### 13 *New developments and challenges in text-to-image generation* 289

- 13.1 State-of-the-art text-to-image generators 290  
*DALL-E series 290 ■ Google's Imagen 291 ■ Latent diffusion models: Stable Diffusion and Midjourney 292*
- 13.2 Challenges and concerns 293
- 13.3 A blueprint to fine-tune ResNet50 295  
*The history and architecture of ResNet50 295 ■ A plan to fine-tune ResNet50 for classification 297 ■ Using ResNet50 to classify images 299*

13.4	Fine-tuning ResNet50 to detect fake images	304
	<i>Downloading and preprocessing real and fake face images</i>	304
	<i>Fine-tuning ResNet50</i>	308
	▪ <i>Detecting deepfakes using the fine-tuned ResNet50</i>	311
appendix	<i>Installing PyTorch and enabling GPU training locally and in Colab</i>	315
	<i>references</i>	323
	<i>index</i>	327



## *preface*

---

This book begins with my curiosity about how machines could create images from nothing more than words. When I first encountered DALL-E and Stable Diffusion, the results seemed magical: type a prompt, and out came a lifelike image that matched the description perfectly. But behind the magic were mathematics, code, and a long line of ideas in machine learning. I wanted to demystify those ideas, not just for myself, but for anyone who learns best by building things from scratch.

Generative AI is advancing at a pace few of us could have predicted, reshaping not only the way we work but also how we create, design, and communicate. Text-to-image models in particular are among the most visible and transformative of these technologies. They embody the leap from unimodal to multimodal AI, systems that reason across different types of data. While the headlines focused on their impressive outputs, I found myself drawn to this question: How do they really work? The only satisfying answer, I decided, was to build one myself.

This book is the result of that journey. It's not a collection of high-level explanations or black box demonstrations. Instead, it's a hands-on guide to re-creating the fundamental building blocks of text-to-image generation: transformers, vision models, diffusion processes, and latent representations. By reconstructing these systems piece by piece, readers like you gain a deeper understanding of both their power and their limitations. As Richard Feynman once put it, "What I cannot create, I do not understand." That spirit guides every chapter.

Writing this book also came from a desire to bridge two communities: the machine learning researchers pushing the frontier of generative models and the developers, designers, and enthusiasts who are eager to harness these tools but uncertain where to begin. My hope is that by working through code, experiments, and projects, you'll see

that these models aren't impenetrable black boxes, but accessible systems built from understandable components.

The examples in this book are intentionally playful (pandas in top hats, bananas on motorcycles) because creativity should be joyful. But the lessons carry serious value—from rapid prototyping in design and marketing, to aiding education, to enabling new forms of artistic expression. By the end of this book, I hope you not only understand how text-to-image models function but also feel comfortable extending, adapting, and imagining new applications of your own.

# acknowledgments

---

Many people have helped bring this book from idea to reality, and I'm deeply grateful for their contributions.

First, I owe special thanks to Jonathan Gennick, my acquisition editor at Manning. Jonathan not only believed in this project from the very beginning but also guided it with a clear vision of what readers most want to learn. His thoughtful feedback helped me shape the structure of the book in a way that makes the learning journey approachable and rewarding.

I'm equally indebted to my development editor, Doug Rudder, whose insistence on clarity pushed me to become a better writer. Doug's ability to spot where explanations could be simplified or examples expanded has left a lasting mark on the book. His encouragement to always aim for clarity, even when discussing the most technical details, has greatly enhanced the accessibility of the book.

My sincere gratitude also goes to my technical editor, Nathan Crocker, author of *AI-Powered Developer* (Manning, 2024) and cofounder and CTO of Checker, the global liquidity network for stablecoins. Nathan's rigorous technical review and insightful comments were invaluable. His careful attention to detail caught subtle issues, and his practical suggestions improved the book tremendously. One memorable example is from chapter 11, where I initially instructed readers to clone and manually update an outdated GitHub repository. Nathan wisely proposed that I instead provide a forked and updated repository so readers could dive directly into experimentation without unnecessary manual updating. It's improvements like these, both large and small, that make this book far more accessible to readers.

I would also like to thank the broader Manning team, including the production editor, copyeditor, and reviewers, who worked behind the scenes to polish the manuscript

and ensure its accuracy. Their efforts often go unseen but are crucial to turning a draft into a finished book. To all of the following reviewers, your suggestions helped make this a better book: Abhilash Babu, Adil Patel, Akinwale Habib, Alejandro Cuevas Rivero, Amaresh Rajasekharan, Arun Lakhera, Atilla Özgür, Curtis Bates, Devavrat Sabnis, Dieter Späth, Donald Bleyl, Eli Richmond Hini, Gautham K, Giovanni Alzetta, Hardev Ranglani, Ifiok Moses, Ijem Ofili, Ishita Verma, Jahred Love, Jan Goyvaerts, José Salavert-Torres, Khashayar Baghizadeh, Mariia Bulycheva, Matheus Antônio Nogueira, Mouhamed Klank, Naga Sai Abhinay Devarinti, Oscar Peña, Pavan Kumar Adepu, Praveen Nair, Ravikumar Sanapala, Raymond Cheung, Sergio Arbeo, Shabie Iqbal, Mohammad Shahnawaz Akhter, Simeon Leyzerzon, Sukanya Konatam, Tathagata Dasgupta, Thomas M. Seeber, Trinawat Charoenpradubsilp, Vikram Shibad, Vishal Gandhi, Xin Hu, and Yuanyuan Chen.

Finally, I want to express my deepest gratitude to my family. To my wife, Ivey Zhang, thank you for your patience, encouragement, and constant belief in this project, even during the long evenings and weekends I spent writing. To my son, Andrew Liu, thank you for reminding me of the joy of curiosity, the same spirit that inspired me to write this book. This journey wouldn't have been possible without your love and support.

## about this book

---

This book was written with one guiding principle: the best way to truly understand how something works is to build it from the ground up. *Build a Text-to-Image Generator (from Scratch)* takes this philosophy and applies it to one of the most exciting areas in AI today: text-to-image generation. Rather than treating modern AI systems as impenetrable black boxes, this book guides you step-by-step through the construction of the core components that make them work: transformers, vision models, diffusion processes, and multi-modal architectures. By the end, you'll not only know how to use state-of-the-art models, such as Stable Diffusion and DALL-E, but also how to re-create simplified versions of them yourself, giving you both practical skills and a deep conceptual foundation.

### **Who should read this book**

This book is written for developers, researchers, students, and curious practitioners who want to move beyond simply running prebuilt AI models and instead learn how they are designed. You should have a solid command of Python and a working knowledge of machine learning, especially neural networks in PyTorch. A background in deep learning fundamentals, such as convolutional networks, embeddings, and training loops, will be helpful, though the book introduces each concept in context. If you're an engineer seeking to deepen your AI skills, a researcher exploring multi-modal learning, or simply an enthusiast who learns best by coding, this book is for you.

### **How this book is organized: A road map**

The book is organized into four parts:

- *Part 1: Understanding attention and transformers*—Introduces transformers, the architecture that revolutionized natural language processing and later computer vision.

You'll build transformers from scratch and apply them to machine language translation (e.g., German to English), classification, and image captioning tasks.

- *Part 2: Introduction to diffusion models*—Explains how diffusion models work by gradually denoising random noise into coherent images. You'll implement diffusion-based image generation and explore methods to control and enhance the quality of generated images.
- *Part 3: Text-to-image generation with diffusion models*—Focuses on multimodal learning. You'll train and experiment with contrastive language-image pretraining (CLIP) for measuring text-image similarity, implement latent diffusion, and take a deep dive into the architecture of Stable Diffusion.
- *Part 4: Text-to-image generation with transformers*—Demonstrates how to generate images using transformer-based approaches. You'll learn about vector quantized generative adversarial network (VQGAN) for converting images into discrete tokens and build a minimal implementation of DALL-E.
- *Part 5: New developments and challenges*—The last chapter surveys recent advances and open challenges in text-to-image generation, including copyright issues, ethical concerns, and future research directions. The second half of the chapter provides a hands-on guide on how to fine-tune ResNet50 to differentiate real images from deepfakes.

Along the way, you'll complete hands-on projects, such as generating “a panda with a top hat reading a book” or “a banana riding a motorcycle,” to make abstract ideas both engaging and concrete.

### About the code

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a fixed-width font like this to separate it from ordinary text. Sometimes code is also **in bold** to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

You can get executable snippets of code from the liveBook (online) version of this book at <https://livebook.manning.com/book/build-a-text-to-image-generator-from-scratch>. The complete code for the examples in the book is available for download from the Manning website at [www.manning.com/books/build-a-text-to-image-generator-from-scratch](http://www.manning.com/books/build-a-text-to-image-generator-from-scratch) and from GitHub at <https://github.com/markhliu/txt2img>.

Each chapter comes with a Jupyter Notebook that replicates the code presented in the text as well as a Google Colab notebook. I encourage you to run the notebooks, modify the code, and experiment with your own prompts and data.

## Software and hardware requirements

This book assumes you're working in Python 3 with PyTorch installed. Installation instructions for PyTorch, as well as guidance on enabling GPU acceleration locally and in Google Colab, are provided in the appendix. While most examples can run on a modern laptop CPU, training models from scratch will be faster and more enjoyable with access to a GPU (e.g., those available through Colab or a personal NVIDIA GPU). You'll also need standard Python libraries such as NumPy, matplotlib, and Hugging Face's Transformers and Datasets.

## How to use this book

This is a hands-on, build-first book. You'll benefit most by typing out the code, running the experiments, and exploring "what if" scenarios on your own. Each chapter is designed to be self-contained, but reading sequentially will provide the smoothest learning curve. If you're new to transformers or diffusion models, start from the beginning. If you're already comfortable with the basics, you can jump directly into the parts that focus on multimodal and text-to-image generation.

Above all, this book is meant to be practical and enjoyable. By the final chapter, you'll not only have the skills to understand how today's leading text-to-image models work but also the confidence to adapt and extend them for your own projects.

## liveBook discussion forum

Purchase of *Build a Text-to-Image Generator (from Scratch)* includes free access to liveBook, Manning's online reading platform. Using liveBook's exclusive discussion features, you can attach comments to the book globally or to specific sections or paragraphs. It's a snap to make notes for yourself, ask and answer technical questions, and receive help from the author and other users. To access the forum, go to <https://livebook.manning.com/book/build-a-text-to-image-generator-from-scratch/discussion>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It's not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

## *about the author*

---



**MARK LIU** is a tenured finance professor and the founding director of the master of science in finance program at the University of Kentucky. He is the author of four books: *Make Python Talk* (No Starch Press, 2021), *Machine Learning, Animated* (CRC Press, 2023), *AlphaGo Simplified* (CRC Press, 2024), and *Learn Generative AI with PyTorch* (Manning, 2024). Mark obtained his PhD in finance from Boston College. Mark has published his research in top finance journals such as *Journal of Financial Economics*, *Journal of Financial and Quantitative Analysis*, and *Management Science*.



## *about the cover illustration*

---

The figure on the cover of *Build a Text-to-Image Generator (from Scratch)*, titled “L’Actrice,” or “The Actress,” is taken from a book by Louis Curmer published in 1841. Each illustration is finely drawn and colored by hand.

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture centuries ago, brought back to life by pictures from collections such as this one.



# *Part 1*

## *Understanding attention and transformers*

**T**his part introduces the foundations of transformer architectures, which have become the backbone of modern generative AI. We begin with a comparison of transformers and diffusion models, showing how each tackles the problem of generating data in very different ways. From there, you'll build a transformer from scratch in chapter 2 to translate German to English, gaining hands-on experience with the attention mechanism that enables these models to capture relationships across sequences.

We then explore practical applications of transformers in computer vision and multimodal tasks. You'll implement a vision transformer (ViT) to classify images in chapter 3 and build a multimodal transformer to generate captions for images in chapter 4, bridging the gap between visual and textual data. By the end of this part, you'll understand how transformers adapt naturally from text to images, as well as why attention has become the most influential concept in modern AI.



# 1

## *A tale of two models: Transformers and diffusions*

---

### ***This chapter covers***

- The distinction between unimodal and multimodal models
- How vision transformers use attention mechanisms from natural language processing to process images
- The inner workings of diffusion models and how they generate images from noise
- The challenges and limitations facing current text-to-image models

Generative artificial intelligence (generative AI) refers to a class of machine learning models designed to create new content—text, images, audio, or even video—that closely resembles real-world data. Unlike traditional AI systems that merely classify, predict, or retrieve information, generative AI models are creative: they “learn” patterns from massive datasets and then generate entirely new outputs based on those patterns. For example, ChatGPT can write essays and code, while DALL-E and Stable Diffusion can produce images from written descriptions.

Text-to-image generation stands out as one of the most captivating advances within generative AI. These models translate natural language prompts into detailed, visually compelling images, often with remarkable creativity and realism. Recent breakthroughs such as OpenAI's DALL-E 2, Google's Imagen, and Stability AI's Stable Diffusion have captured the world's attention by turning abstract descriptions into vivid pictures, sometimes indistinguishable from photographs or human art. Beyond creating images from scratch, these systems can also edit existing photos through text commands (e.g., cropping, removing objects, or changing backgrounds), making them valuable tools for photographers and designers alike. Companies such as Adobe have integrated text-to-image and text-based editing into their design suites, enabling graphic designers to instantly visualize and refine concepts. In healthcare, start-ups use these models to convert doctors' notes into medical diagrams. The real-world impact is profound: generative AI is reshaping industries ranging from design and entertainment to education and medicine.

At the core of these systems are two powerful approaches: (1) transformer architectures that were originally developed for natural language processing (NLP) and (2) diffusion models. Vision transformers (ViT), for instance, adapt the same attention mechanisms that revolutionized language models such as BERT and GPT, applying them to image data by treating an image as a sequence of smaller patches. Meanwhile, diffusion models take a radically different path: they start with pure random noise and iteratively denoise it, gradually shaping it into a coherent image that matches the prompt. Both methods have redefined what's possible in multimodal AI, where systems must reason across multiple types of data (text, images, audio, and more).

This chapter explores the foundations and real-world impact of text-to-image generation. I'll clarify key concepts, from unimodal (dealing with one single data type) to multimodal (dealing with two or more data types) models, and walk through the mechanics of both transformer- and diffusion-based approaches. Along the way, I'll highlight the practical applications and the ways these technologies are already transforming workflows. You'll see, for example, how companies such as Canva let users generate illustrations for marketing with a simple prompt, as well as how platforms such as Midjourney enable artists to explore new creative frontiers.

Additionally, you'll learn about the unique challenges text-to-image generation models face, such as interpreting complex prompts, maintaining geometric consistency, and addressing questions of ethics and copyright. Whether you're an AI researcher, a developer, or simply curious about the future of creativity, understanding these technologies will give you an edge as generative AI continues to reshape how we interact with information and art.

## **1.1** *What is a text-to-image generation model?*

Text-to-image models are multimodal generative models designed to transform a text description into a corresponding image. *Multimodal* means that the model processes different types of data. In the case of text-to-image models, the input is text, while the

output is an image. Multimodal models can handle a variety of input–output combinations, such as text, images, audio, and video. This contrasts with *unimodal* models, where only one type of data is handled by the model.

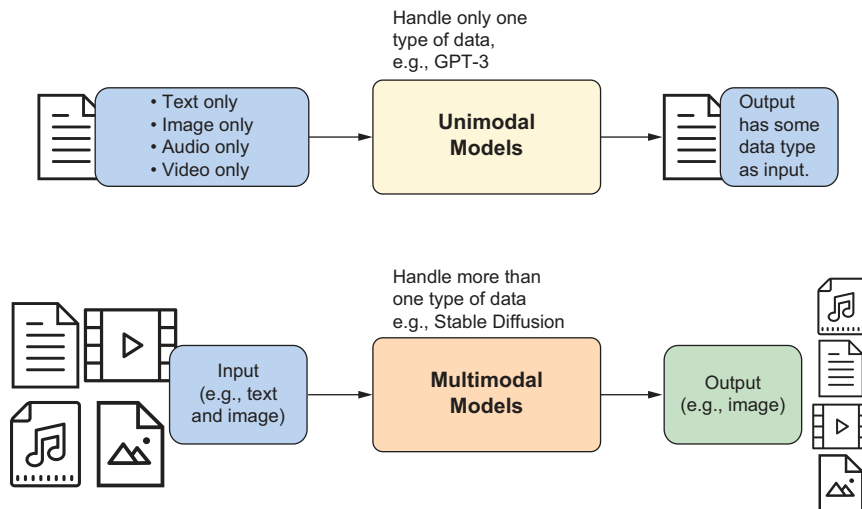
In this section, I'll cover text-to-image generation models, emphasizing the fact that they are multimodal instead of unimodal models. I'll then outline the practical applications of text-to-image generation.

### 1.1.1 Unimodal vs. multimodal models

Unimodal models operate within a single type of data modality, such as text-only or image-only models. For instance, a unimodal text model generates text based solely on text input, similar to traditional language models. As a specific example, GPT-3 is a unimodal model because it processes text as input and generates text as output. Likewise, an image model might generate images from random noise, but it would only process image data.

In contrast, multimodal models connect different data modalities, enabling interactions across text, images, audio, and video. Text-to-image models exemplify this approach, taking input in one modality (e.g., text) and generating output in another (images). These models use a combination of NLP and computer vision techniques to interpret textual descriptions and create corresponding images. A prominent example is text-to-image generation models (e.g., Stable Diffusion), where the input is text and an image (when editing existing images), and the output is an image.

Figure 1.1 shows the difference between unimodal and multimodal models. The top half of the figure illustrates how unimodal models operate. In these models, the input and output are of the same type. An example is GPT-3, which takes text as input and



**Figure 1.1** Comparison of unimodal and multimodal models

generates text as output. In contrast, the bottom half of the figure depicts multimodal models, which handle different data types. An example is Stable Diffusion, a text-to-image model. Here, the input is a text description, and the output is an image that matches the description. Alternatively, as you'll see later in the book, Stable Diffusion can also modify an existing image using text descriptions. In this case, the input is text and an image, while the output is an image.

### 1.1.2 *Practical use cases of text-to-image models*

Text-to-image models have a wide range of practical applications in real-world scenarios, such as content generation, product design, and educational and training tools. While images are their primary output, learning how to build these models from scratch also equips you with skills relevant to related tasks. For example, the ability to align text and image representations can be extended to measuring similarity between text and images or to selecting the most appropriate image for a given description. You'll also learn to build and train an image-to-text model to add captions to images.

These models can rapidly generate high-quality content, making them ideal for producing art, illustrations, and other creative visuals based on textual input. This capability is particularly useful for artists, designers, and writers who need to quickly prototype visual concepts. In advertising and marketing, businesses can use these models to create targeted advertisements, generate engaging marketing content, or quickly produce visuals tailored to specific customer descriptions or product requirements.

In product design, text-to-image models enable rapid prototyping based on design prompts, facilitating faster design cycles and greater flexibility in exploring variations. Game designers, for instance, can create environments, characters, assets, and concept art for games, animations, or films simply by describing their features. Similarly, fashion designers can generate prototypes, sketches, or visual concepts from descriptive prompts, accelerating visual iteration and market testing.

Text-to-image models are also valuable in education and training, helping illustrate complex concepts described in text, such as visualizing historical events, scientific phenomena, or medical conditions. Additionally, these models can assist in data augmentation for machine learning by generating synthetic datasets with realistic images based on textual descriptions, helping train and augment models when data is scarce.

The skills you acquire from building text-to-image models from scratch in this book will enable you to create various practical applications. For example, after completing chapter 4, you'll be able to generate captions for images. Figure 1.2 illustrates three examples of adding captions to images.

As shown in figure 1.2, the generated captions may not be identical to the human-created ones (because there are different ways of describing the same image) but still accurately capture the content of the images. For instance, the original caption for the third image states, "two young girls are running through shallow water at a pool," whereas the model-generated caption reads, "two girls are running away from the water and laughing." Despite the difference in wording, the generated caption accurately reflects the



**\*\*Original caption:**  
the children are playing basketball indoors  
**\*\*Generated caption:**  
the boy is playing basketball in the arena



**\*\*Original caption:**  
two white dogs run through the grass  
**\*\*Generated caption:**  
two white dogs running



**\*\*Original caption:**  
two young girls are running through shallow water at a pool  
**\*\*Generated caption:**  
two girls are running away from the water and laughing



**Figure 1.2** Examples of generating captions for images. The original caption from the training dataset, created by humans, is displayed above each image. The image is then fed into a trained image-to-text model to generate the second caption shown. While the generated captions differ from those created by humans, they accurately describe what's going on in these images.

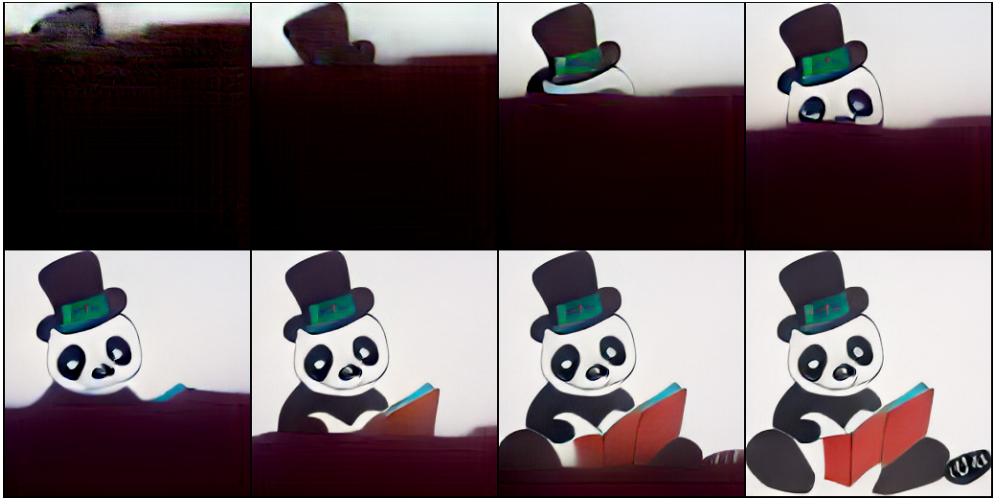
scene depicted in the third image. The captioning technology is especially valuable for the vast number of images that lack human-created captions, making them accessible and searchable in large databases.

Later, you'll learn how to build and train a contrastive language-image pretraining (CLIP) model from scratch. A trained CLIP model enables you to measure the similarity between a text prompt and an image. As a result, you can perform an *image selection* task: input a text description and use the model to identify the image from a large pool that best matches the text. These examples highlight just a few of the many real-world applications of text-to-image generation models and the valuable skill sets they offer.

## 1.2 Transformer-based text-to-image generation

One approach to text-to-image generation involves framing the task as a next-token prediction problem using a transformer. Here, an image is broken down into smaller patches, such as a  $16 \times 16$  grid, resulting in 256 patches in total. These patches are arranged sequentially, starting from the top-left corner and progressing rightward, ending at the bottom-right corner.

For example, consider this text prompt, "panda with a top hat reading a book." Figure 1.3 illustrates the progression of how an image might be formed. The trained transformer first predicts the top-left patch of the image based on the prompt. During each subsequent iteration, the transformer predicts the next patch based on the prompt and the previously generated patches, repeating this process until all 256 patches are generated.



**Figure 1.3** How the min-DALL-E model generates an image based on the prompt “panda with a top hat reading a book.” The top-left subplot shows the output when 32 image patches are generated. The second subplot in the top row shows the output when 64 patches are generated. The remaining images show the outputs for 96, 128, . . . , and 256 patches.

You might be wondering how a model converts a text prompt into meaningful image patches that align with the description. To do this, the training of a transformer-based text-to-image generative model needs three components:

- A submodel (vector quantized generative adversarial network [VQGAN]) to convert an image into a sequence of integers; let’s call the sequence image tokens. The same submodel can convert the image tokens (i.e., a sequence of integers) back to the original image.
- Another submodel (bidirectional and auto-regressive transformer [BART]) to convert text descriptions into a sequence of integers; let’s call the sequence text tokens.
- A method to make the image tokens identical to the text tokens.

To generate an image from a text prompt after the model is trained, we feed the text prompt to BART to produce text tokens. Because the text tokens are trained to be identical to the image tokens, we can feed the text tokens to VQGAN to produce an image as the output that matches the text prompt.

Sounds abstract? I agree. Let’s dive deeper into the transformer-based text-to-image generative model and see how various components work.

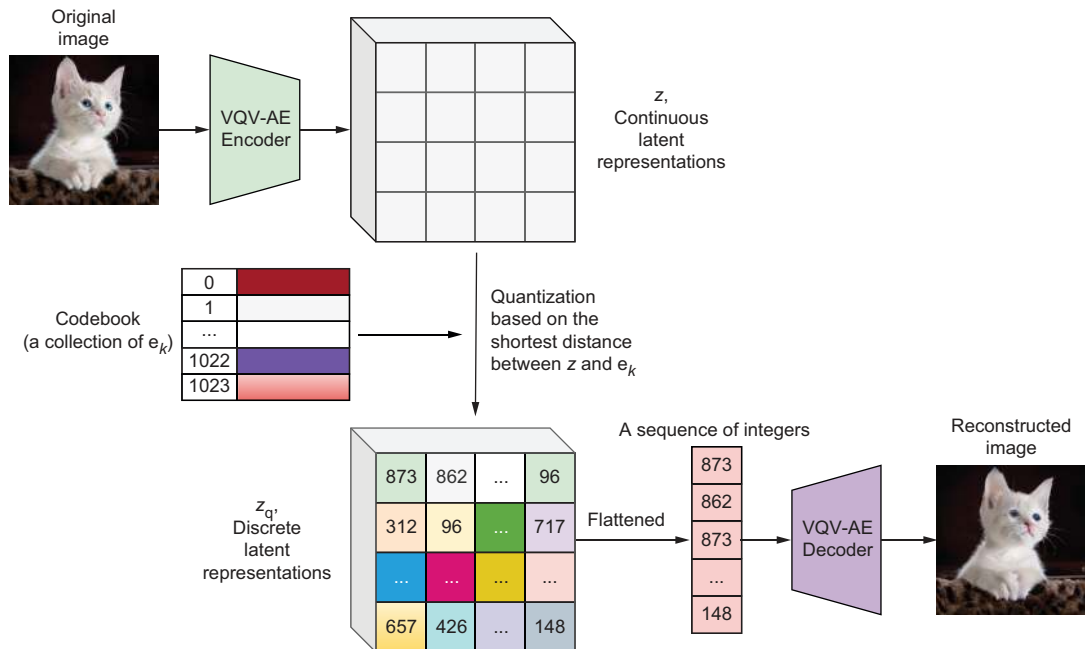
### 1.2.1 *Converting an image into a sequence of integers and then back*

In transformer-based text-to-image generation, a crucial step is converting an image into a sequence of integers, just as language models convert text into sequences of

word tokens. This transformation is enabled by VQGAN, as I'll explain in detail in chapter 11. In a nutshell, VQGAN uses an encoder to compress the input image into a compact, lower-dimensional latent representation. The latent vector is replaced by its closest match from a learned codebook containing a fixed set of discrete vectors. This quantization process maps the image into a sequence of integers where each integer refers to an index in the codebook.

Why is this important? This discrete sequence allows us to treat the image in a way that's analogous to text: just as sentences can be represented by sequences of word indices, images can now be represented as sequences of codebook indices. This enables the use of transformer architectures, originally developed for modeling language, to process and generate images patch by patch, token by token.

Figure 1.4 is a diagram of how VQGAN works. The encoder in VQGAN compresses an image into a lower-dimensional latent space. The latent vector for each image is divided into different patches. The continuous latent vector for each patch is then compared to the discrete vectors in the codebook. The quantized latent vector uses discrete vectors in the codebook to approximate the continuous latent vector for each image patch. The quantized latent vectors are then passed through the decoder in VQGAN to reconstruct the image.



**Figure 1.4** A diagram of VQGAN

The encoder in VQGAN first transforms an image into a grid, where each cell corresponds to a “patch” of the original image. For example, we can divide an image into  $16 \times 16 = 256$  patches. Each patch is represented by a continuous vector in a lower-dimensional latent space. These continuous latent representations are compared to the vectors in a codebook, which has a finite number (e.g., 1,024) of discrete vectors. The quantized latent vector uses discrete vectors in the codebook to approximate the continuous latent vector for the image patches. This step is crucial because it converts an image into a sequence of integers. For example, the image in the figure is converted to the sequence (873, 862, 873, . . . , 148). This means the top-left patch is represented by vector 873 in the codebook, the patch to the right is represented by vector 862, . . . , and the bottom-right patch is represented by vector 148. This step discretizes the image information, effectively turning continuous image data into a sequence of integers with each integer representing an image token.

The quantized latent vectors are then passed through the decoder in VQGAN to reconstruct the image. This step allows a sequence of image tokens (i.e., a sequence of integers) to be converted to realistic images by the trained decoder in VQGAN.

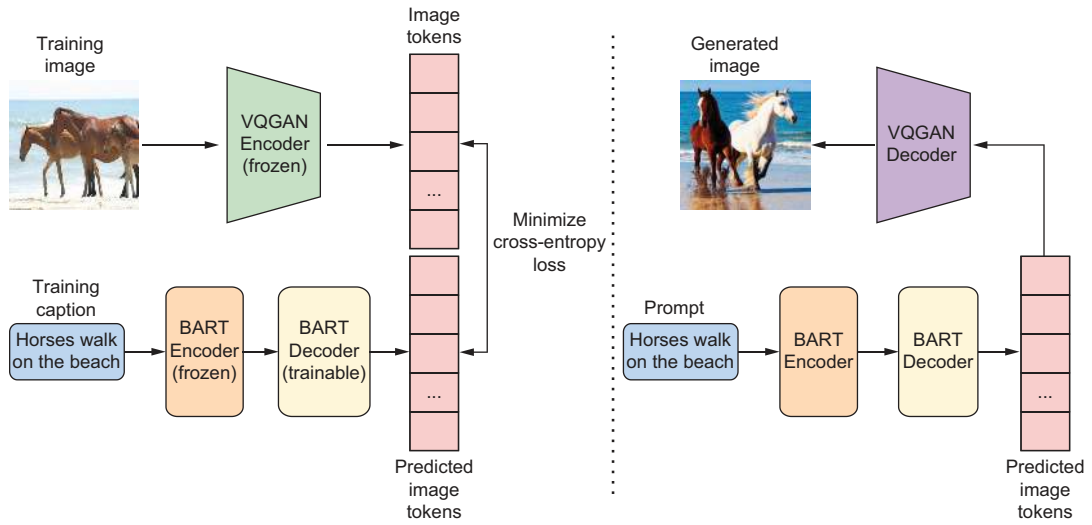
To summarize, the encoder in VQGAN transforms an image into a sequence of integers. The decoder in VQGAN converts a sequence of integers back into an image. These two features have important implications for transformer-based text-to-image models, as we’ll explain in the next subsection.

### 1.2.2 *Training and using a transformer-based text-to-image model*

OpenAI’s DALL-E is a transformer-based text-to-image model. While DALL-E isn’t open source, Boris Dayma et al.’s DALL-E mini project (<https://github.com/borisddayma/dalle-mini>) aims to replicate it. In chapter 12, you’ll learn how to generate images from text prompts using Brett Kuprel’s min-DALL-E project (<https://github.com/kuprel/min-dalle>), which is a PyTorch implementation of the DALL-E mini project.

Figure 1.5 is a diagram of how DALL-E mini is trained and then used to generate an image based on a text prompt. The left side of this figure depicts how a transformer-based text-to-image model is trained, while the right side illustrates the process of generating an image from a text prompt using the trained model. To train the model, images are encoded into image tokens using a VQGAN encoder. Corresponding captions are processed through a BART encoder and then a BART decoder to generate text tokens. The objective is to train the BART decoder to predict text tokens that match the image tokens produced by the VQGAN encoder. To generate an image using the trained model, the text prompt is fed into the BART encoder and then the BART decoder to produce the predicted text tokens, which are passed through the VQGAN decoder to generate the final output, as shown at the top center of the figure.

I’ll cover the DALL-E mini project in detail in chapter 12, but here’s a summary. During training, we start with a large collection of image–caption pairs. Each image is encoded into a sequence of discrete tokens using VQGAN, while the paired caption is transformed into a sequence of text tokens via a BART encoder and then a BART



**Figure 1.5** How DALL-E mini is trained to generate an image based on a text prompt

decoder. The core training objective is to align the BART decoder's output with the image tokens produced by the VQGAN encoder. This is achieved by minimizing the difference between the text tokens and the image tokens.

At inference time, the process works in reverse. Given a text prompt, we encode it using the BART encoder and pass it through the BART decoder to generate a sequence of text tokens. These predicted tokens are then fed into the VQGAN decoder, which reconstructs and outputs the final image that matches the prompt, as shown in the top center of figure 1.5.

### 1.3 Text-to-image generation with diffusion models

The second approach for text-to-image generation is based on diffusion models. To understand how text-to-image diffusion models work, I'll first discuss the idea behind forward diffusion and reverse diffusion. Imagine you want to train a diffusion model to generate high-resolution images (not based on a text prompt, but unconditionally, without any input). You first acquire a set of images for training. You'll gradually introduce small amounts of noise into these images. This process is known as *forward diffusion*. After many steps of adding noise, the training images become pure random noise (much like the snowy static on a TV screen).

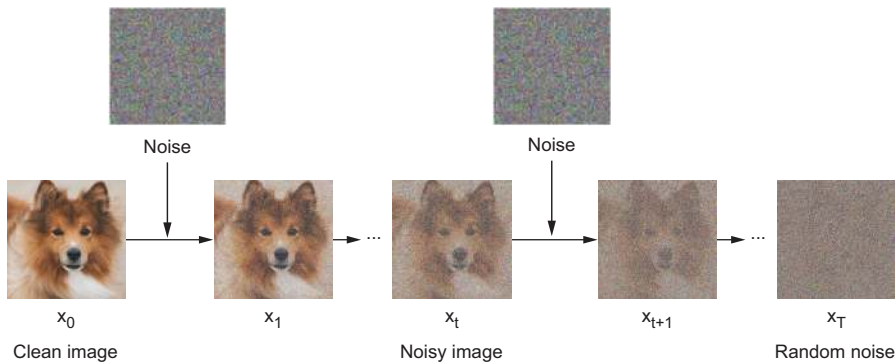
You'll then train a model to reverse this process, which is known as the *reverse diffusion* process (or the *denoising* process). Specifically, you'll start with random noise images and progressively reduce the noise until the images are indistinguishable from those in the original training set. The trained diffusion model can use random noise images as a starting point. The model gradually eliminates noise from the image over many

iterations until a clean image is generated. This is the underlying principle of diffusion-based models.

Diffusion-based text-to-image generation can be viewed as a denoising process that depends on a text prompt. It begins with an image composed entirely of random noise. A trained diffusion model then iteratively removes noise from this image, guided by a text prompt, until a coherent image matching the prompt emerges. Diffusion models have become the preferred choice for generative tasks because of their ability to generate high-quality images by systematically denoising images through progressive steps. They form the foundation of leading text-to-image systems, including Imagen, DALL-E 2, and Stable Diffusion.

### 1.3.1 Forward and reverse diffusions

In forward diffusion, noise is gradually added to a clean image until it becomes pure noise. Figure 1.6 shows how the forward diffusion process works.



**Figure 1.6** In the forward diffusion process, we start with a clean image from the training set,  $x_0$ , and add noise  $\epsilon_0$  to it to form a noisy image  $x_1$ , which is a weighted sum of  $x_0$  and  $\epsilon_0$ . We repeat this process for 1,000 time steps until the image  $x_{1,000}$  becomes random noise.

We start with a clean image,  $x_0$ , which is illustrated with the leftmost image of a dog in figure 1.6. In the forward diffusion process, we'll add small amounts of noise to the image in each of the  $T$  time steps (e.g.,  $T = 1,000$ ; a time step is a discrete stage in the process of gradually adding noise to data). The noise tensor follows a standard normal distribution and has the same shape as the clean image. In time step 1, we add noise  $\epsilon_0$  to the image  $x_0$  so that we obtain a noisy image  $x_1$ , which is a weighted sum of  $x_0$  and  $\epsilon_0$ .

#### Time steps, forward diffusion, and reverse diffusion

A time step is a discrete stage in the process of gradually adding noise to data or removing noise from the data. In the forward diffusion process, we start with a clean



image at time step 0 and gradually add noise to the image to obtain noisy images at time steps 1, 2, . . . , until we reach time step  $T = 1,000$ , when the image becomes pure random noise.

In contrast, in the reverse diffusion process, we start with a pure random noise image at time  $T = 1,000$ . We then gradually remove noise from the image to obtain images at time steps 999, 998, . . . , until we obtain a clean image at time step 0. That is, we go from time step 0 to time step  $T$  in forward diffusion, and we go from time step  $T$  to time step 0 in reverse diffusion.

Let's keep adding noise to the image for the next  $T-1$  time steps so that the noisy image at time step  $t$ ,  $x_t$ , is a weighted sum of  $x_{t-1}$  and  $\epsilon_{t-1}$ . We control the amount of noise injected into the images over time so that at time  $T = 1,000$ , the image,  $x_T$ , becomes complete random noise (shown at the right in figure 1.6).

We'll then construct a deep neural network as the denoising model to reverse the diffusion process, which is the denoising process referred to earlier. If we can train a model to reverse the forward diffusion process, we can feed the model with random noise and ask it to produce a noisy image. We can then feed the noisy image to the trained denoising model again and obtain a clearer, though still noisy, image. We can repeat the process for many time steps, and the output is a clean image. This is the idea behind denoising diffusion models.

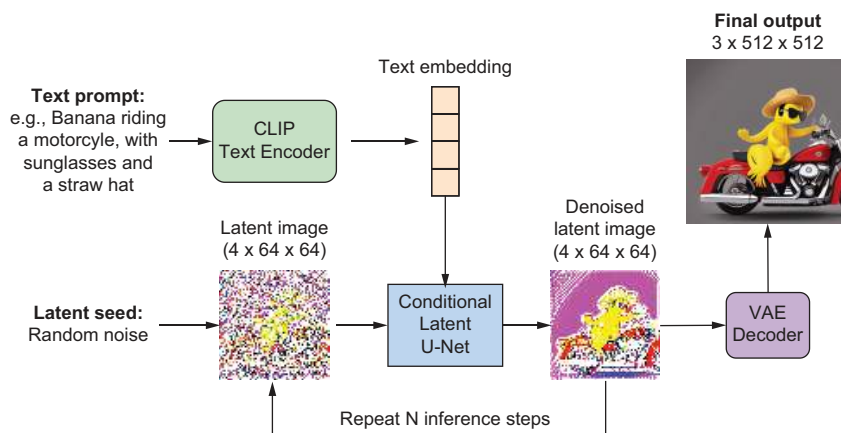
### 1.3.2 Latent diffusion models and Stable Diffusion

To generate an image based on a text prompt, we start with an image composed of pure random noise. We use a trained U-Net denoising model to iteratively remove noise from this image, guided by the text prompt. However, creating a high-resolution image this way can be computationally intense, as images contain many pixels, and the diffusion process involves many time steps (often 1,000 or more). As a result, the generation process is extremely time consuming and inefficient.

Latent diffusion models (LDMs) address this challenge by conducting the forward and reverse diffusion processes on a smaller, compressed version of the image, significantly reducing the pixel count. However, this comes at the cost of lower image quality. To compensate for this, once the LDM is trained, we use a variational autoencoder (VAE) decoder to convert the low-resolution images back into high-resolution images as the final output. A VAE is a generative model composed of an encoder and a decoder. The encoder transforms high-resolution images into compact, lower-dimensional representations, which the decoder then uses to reconstruct the original image.

You'll learn how LDMs work in chapter 9. Instead of conducting forward and reverse diffusion processes on high-resolution images of size  $3 \times 512 \times 512$  (3 color channels with a height and width of 512 pixels), the LDM in our example conducts diffusion processes in a latent space where the images have dimensions of  $4 \times 64 \times 64$ . This reduces the size of the input to the diffusion model by 48 times (since  $(3 \times 512 \times 512) \div (4 \times 64$

$\times 64) = 48$ ), which greatly speeds up the diffusion processes. The images in the latent space are then fed to a trained VAE, which converts them into high-resolution images of size  $3 \times 512 \times 512$ . Figure 1.7 is a diagram of the steps involved in text-to-image generation with a trained LDM.

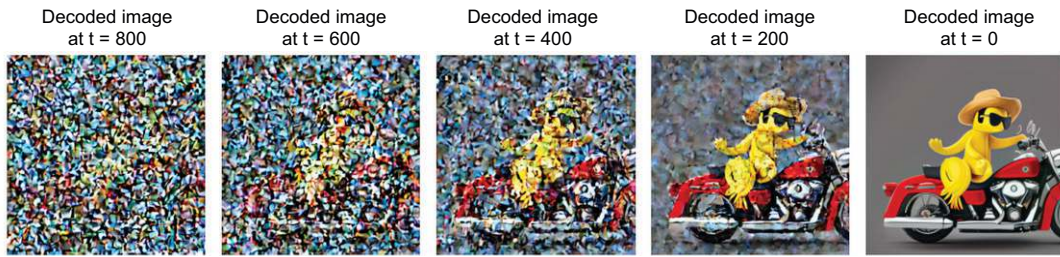


**Figure 1.7** How a trained LDM generates an image based on a text prompt

To generate an image, we first encode the text prompt (e.g., “a banana riding a motorcycle, wearing sunglasses and a straw hat”) into a text embedding (top left of figure 1.7). We then start the reverse diffusion process in the latent space, as illustrated in the bottom half of the figure. We iteratively denoise a noisy image, conditional on the text embedding so the generated image matches the text embedding. This step is guided by a CLIP model, which you’ll learn all about in chapter 8. The CLIP model guides the image generation by maximizing the similarity between the text embedding and the generated image embedding. Once the reverse diffusion process is finished in the lower-dimensional (latent) space, the latent image is presented to a trained VAE decoder to convert it to a high-resolution image, which is the final output of the model (top right of figure 1.7).

Better yet, you can visualize the intermediate outputs from the LDM. Figure 1.8 shows the generated images at different time steps during the image generation process. The left image shows that at time step  $t = 800$ , the image looks close to pure noise. As we move to the right, the image starts to match the text prompt. At time step  $t = 0$ , you can see a clean image of a banana riding a motorcycle. Recall that in reverse diffusion, we start with a pure random noise image at  $T = 1,000$ . We then gradually remove noise to obtain cleaner images at  $t = 999, 998, \dots$ , and eventually  $t = 0$ . This is why you see noisy images at  $t = 800$  and less noisy images at  $t = 600, 400, \dots$ , until you see a clean image at  $t = 0$  in figure 1.8.





**Figure 1.8** Intermediate decoded outputs from a trained LDM at time steps 800, 600, . . . , 200, and 0. The text prompt is “a banana riding a motorcycle, wearing sunglasses and a straw hat.”

Stable Diffusion is based on the LDM discussed earlier. It’s one of the state-of-the-art text-to-image models, along with OpenAI’s DALL-E 2 and Google’s Imagen. However, Stable Diffusion is the only one that is open source and free for anyone to use.

Stable Diffusion was developed by researchers from CompVis, Stability AI, and LAION. It’s largely based on the 2022 paper by Rombach et al. [1]. Stable Diffusion has incorporated several improvements and additional features:

- Stable Diffusion uses improved training techniques and optimizations, which lead to better stability and performance, further enhancing the quality of generated images.
- While the LDM in the 2022 paper was trained on the LAION-400M dataset (with 400 million image–text pairs), Stable Diffusion was trained on a subset of an even larger dataset, the LAION-5B database (with 5 billion image–text pairs).
- Stable Diffusion has developed user-friendly applications and interfaces, democratizing access to high-quality generative models. These tools enable users without deep technical expertise to have access to the power of diffusion models.

In chapter 10, you’ll learn how to use Stable Diffusion to generate high-quality images through text prompts by using Hugging Face’s `Diffusers` library, but I’ll first discuss how to generate images using out-of-the-box tools. The `StableDiffusionPipeline` package in the `Diffusers` library allows you to use Stable Diffusion as an off-the-shelf tool to generate images in just a few lines of code. For example, the prompt “an astronaut in a spacesuit riding a unicorn” leads to the image shown in figure 1.9.

We’ll then take a dive deep into the Stable Diffusion model and look at how various components of Stable Diffusion work. Instead of treating Stable Diffusion as a black box, we’ll look under the hood and see how each component of the model contributes to the text-to-image generation process.

## 1.4 How to build text-to-image models from scratch

The best way to learn something is always by building it from scratch. In this book, you’ll create text-to-image models from the ground up so you can grasp a deep

understanding of various components within these models. Building these models yourself allows you to use them more effectively.

Throughout the book, you'll implement and train key components of text-to-image models using small datasets, making it feasible to complete training in Google Colab or on a standard computer equipped with a CUDA-enabled GPU (NVIDIA GeForce RTX 20 series or better) in just a few hours. For parts of the models that require extensive data and time to train, we'll load publicly available pre-trained parameters so you can witness these models in action.

The models discussed in this book are built on deep neural networks. We'll use Python and PyTorch to create, train, and deploy these models. Python is chosen for its intuitive syntax, cross-platform compatibility, and active community support. Meanwhile, PyTorch stands out among other frameworks, such as TensorFlow, due to its dynamic computational graph, which makes it highly adaptable and easier to debug. This book assumes you have a working knowledge of Python, including basic concepts such as functions, classes, lists, and dictionaries. If you need a refresher, there are numerous free online resources available. The appendix includes instructions for installing Python, creating a virtual environment for this book, and setting up Jupyter Notebook as the primary computing environment. The appendix also covers how to install PyTorch with CUDA or train models with a GPU in Google Colab.

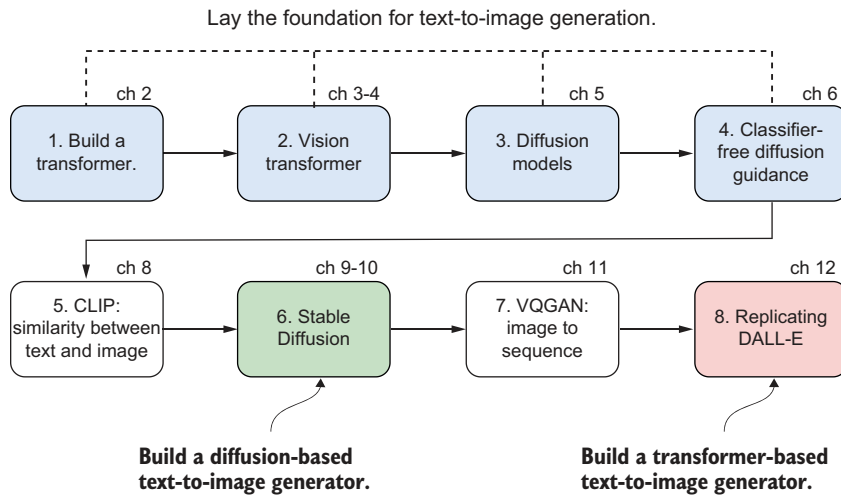
You should also have basic knowledge of machine learning and an understanding of training deep neural networks using PyTorch. If you're not there yet, consider starting with a book on this subject, such as *Deep Learning with PyTorch* by Luca Antiga, et al. (Manning, 2020).

Figure 1.10 outlines the eight-step learning journey you'll follow to build your own text-to-image generator from scratch. Each step builds practical skills and conceptual understanding, guiding you from basic model components to advanced multimodal systems.

Let's walk through what you'll achieve in each step. In step 1, you'll build and train an encoder-decoder transformer for tasks such as translating German to English.



**Figure 1.9** The `StableDiffusionPipeline` package in the `Diffusers` library allows you to use Stable Diffusion as an off-the-shelf tool to generate great images in just a few lines of code. Here, for example, it creates an image of an “astronaut in a spacesuit riding a unicorn.”



**Figure 1.10** The eight steps to building a text-to-image generator from scratch

This step prepares you to code the attention mechanisms and learn the transformer architectures. In step 2, you'll dive into computer vision by constructing a ViT for image classification. You'll see how images are split into patches and processed like sequences, which is a crucial insight for multimodal modeling.

We'll explore the fundamentals of forward and reverse diffusion in step 3 by implementing a basic diffusion model that learns to denoise images, a key building block of today's best image generators. In step 4, you'll learn to steer the generation process with conditioning information. This prepares you for the use of classifier-free guidance, which lets you control how closely images match your text prompts, balancing creativity and accuracy.

In step 5, you'll build a CLIP model from scratch and learn how to connect images and text in a shared embedding space. CLIP is essential for guiding LDMs, as it quantifies the similarity between generated images and text descriptions. In step 6, you'll combine your knowledge to construct a diffusion-based text-to-image generator, using latent spaces and CLIP guidance. In step 7, you'll master the VQGAN framework, which enables you to convert images into discrete codebook sequences, making transformer-based image generation possible. Finally, in step 8, you'll build a min-DALL-E, a transformer-based model that predicts image tokens one at a time from text prompts, echoing the architecture behind OpenAI's DALL-E model.

Each step is carefully crafted to build your intuition, skill, and confidence. By the end of this book, you'll not only understand *how* text-to-image models work but also *why* they work and how to build, extend, and improve them yourself.

Figure 1.10 serves as a road map for the whole book that we'll refer back to as we progress. With this structured, hands-on approach, you'll transform from a consumer

of generative AI into a true creator, capable of building text-to-image models from scratch and understanding every layer of their magic.

## 1.5 **Challenges for text-to-image generation models**

This is an exciting time for AI enthusiasts in general. Along with other generative models, text-to-image generation has made great strides in recent years, transforming from rudimentary image generation into a sophisticated and creative tool capable of generating highly detailed and realistic images from text prompts.

OpenAI's DALL-E series, Google's Imagen, and Stability AI's Stable Diffusion represent major milestones in text-to-image generation. These models take advantage of advancements in deep learning, particularly in transformer architectures and diffusion models, to bridge the gap between linguistic and visual modalities. Chapter 13 will explain how the DALL-E series, Imagen, and Midjourney work.

However, despite these advancements, the state-of-the-art text-to-image generation models still face many challenges. In this section, we'll focus on two of these: (1) the debate on whether text-to-image models "steal" from artists by reproducing training data or whether they create new concepts and (2) the geometric inconsistency problem.

### 1.5.1 **Are generative AI models stealing from artists?**

The first challenge facing state-of-the-art text-to-image models is the debate on whether text-to-image models steal from artists by reproducing training data or whether they create new concepts [2]. The debate is rooted in concerns over intellectual property, creativity, and the nature of generative AI. This debate has two main camps.

The first camp argues that current text-to-image models reproduce and steal from training data because these models are typically trained on vast datasets, which may include copyrighted artworks, photographs, and illustrations without explicit permission from the creators. Therefore, text-to-image models are, in effect, "copying" aspects of these works by generating images that are influenced by the style, composition, or elements of the original pieces in the training data. This is especially true if certain images are overrepresented in the training data. In such cases, the model may unintentionally or intentionally reproduce elements that are highly similar to the original artwork, blurring the line between generation and copying. There have been instances where models seemed to re-create nearly identical versions of known artworks or images when given prompts closely related to them [3].

The second camp argues that text-to-image models create new concepts, rather than reproducing training data. People in this camp are mostly proponents of generative models, and they argue that these models don't memorize or reproduce images from their training data directly, but rather learn patterns and statistical relationships between pixels, objects, and styles. Like human artists, models are capable of "inspiration" from vast sources without copying any single work. They combine concepts in new and unexpected ways to generate original outputs.

People on this side argue that the process of image generation in models such as diffusion models involves adding random noise to an image and then refining that noise based on the patterns learned during training. This process inherently prevents the model from reproducing exact images from the training data. The outputs are always new, even if they are influenced by certain stylistic or compositional elements. Proponents emphasize that text-to-image models are tools that generate new and novel content by learning from patterns in data, much like how human artists learn and create. This debate will likely evolve as legal and ethical standards catch up to technological advancements in generative AI.

### **1.5.2 The geometric inconsistency problem**

The second challenge facing text-to-image generation models is the geometric inconsistency problem. Text-to-image models, whether based on transformers or diffusion models, often struggle with maintaining geometric consistency because they are typically trained on large, diverse datasets of 2D images without a deep understanding of 3D structure or physical rules governing spatial relationships. Therefore, these models don't inherently understand 3D geometry. They learn patterns and associations between pixels and the semantic meaning of text, but they lack an internal representation of the 3D world, which is essential for maintaining consistent spatial relationships across objects and viewpoints.

Transformer-based models use positional encodings to understand spatial relationships, but these are limited when it comes to enforcing strict geometric constraints, especially across complex 3D objects. Current text-to-image models typically lack a robust mechanism for encoding depth and perspective consistently. These models don't understand the rules of perspective, lighting, or physics as humans do. As a result, they might generate objects that defy physical laws or appear distorted when viewed from certain angles, which breaks geometric consistency. Achieving geometric consistency would likely require incorporating models that explicitly consider 3D structure, physics simulations, or other methods that enforce spatial coherence.

## **1.6 Social, environmental, and ethical concerns**

Text-to-image generation, like other generative AI technologies, has rapidly advanced, evolving from basic image generation to a powerful tool capable of creating detailed, realistic images from text prompts. As these state-of-the-art models have advanced, new concerns have emerged. This section explores the social, environmental, and ethical implications associated with these models.

The first concern centers on the computational costs and environmental impact of these models. Developing and deploying leading text-to-image models is computationally intensive, requiring vast resources, including large datasets and powerful hardware. The energy consumption for training these models is significant, contributing to environmental costs. For example, an article in *Computer Weekly* in June 2024 mentioned that "if the rate of AI usage continues on its current trajectory without any form of

intervention, then half of the world's total energy supply will be used on AI by 2040" [4]. Although techniques such as pruning, quantization, and transfer learning have been developed to offset the impact, the scalability of text-to-image models remains an obstacle. Reducing the computational footprint without compromising output quality remains a critical research focus.

A second concern involves the potential misuse of model-generated images. The ability to produce realistic images opens possibilities for misuse in the form of deep-fakes and misinformation. These models can be exploited to create misleading, propagandistic, or manipulative content, posing risks to public trust, democratic processes, and reliable information. For example, in early 2024, thousands of New Hampshire voters received a robocall purporting to be from Joe Biden [5]. Addressing these risks calls for robust content moderation systems, clear guidelines for responsible use, and partnerships between AI developers, policymakers, and media organizations to curb the spread of misinformation.

Finally, text-to-image models often reflect societal biases related to race, gender, or culture. For example, prompts like "a doctor" or "a leader" may yield stereotypical depictions that amplify existing social inequalities. Addressing these biases requires methods for auditing and mitigating biases within models, a challenge that demands collaboration among AI developers, social scientists, and ethicists. However, it's essential to avoid overcorrection, as demonstrated by cases such as Google's Gemini model, which, in seeking to correct stereotypes, inadvertently produced inaccurate historical depictions, for example, including people of color in portrayals of Nazi-era German soldiers [6].

Text-to-image generative AI models have made tremendous progress in the past few years, becoming transformative tools in art, design, and AI. However, as these models become more integrated into society, addressing the challenges of resource consumption, ethical use, and bias will be essential for responsible and sustainable development.

## Summary

- Text-to-image models are a type of multimodal generative model designed to transform a text description into a corresponding image.
- Unimodal models operate within a single type of data modality, such as text-only or image-only models. In contrast, multimodal models connect different data modalities, enabling interactions across text, images, audio, and video.
- Transformer-based text-to-image generation models treat images as sequences by dividing them into patches, each patch acting as an element in the sequence. Image generation is then a sequence prediction problem, where the model predicts patches from top-left to bottom-right based on a text prompt.
- In diffusion-based text-to-image generation models, we start with an image of pure noise. The model iteratively denoises it based on the text prompt, reducing noise with each step until a clear image matching the prompt is produced.

- Instead of conducting forward and reverse diffusion processes on high-resolution images, latent diffusion models (LDMs) conduct diffusion processes in a lower-dimensional latent space, making the process faster and more efficient. After training, a variational autoencoder (VAE) converts the low-resolution latent space images into high-resolution final outputs.
- Despite significant advancements, text-to-image generative models face challenges such as copyright disputes, geometry inconsistencies, and social, ethical, and environmental concerns.



# Build a transformer



## ***This chapter covers***

- How the attention mechanism assigns weights to elements in a sequence
- Building an encoder–decoder transformer from scratch for language translation
- Word embedding and positional encoding
- Training a transformer from scratch to translate German to English

Understanding attention and transformer architectures is foundational for modern generative AI, especially for text-to-image models. This chapter comes at the very beginning of our journey to build a text-to-image generator from scratch for two reasons:

- One of the most powerful approaches to text-to-image generation is based directly on transformers. As you'll see in chapter 12, models such as OpenAI's DALL-E treat image generation as a sequence prediction task. An image is divided into patches (e.g., a  $16 \times 16$  grid, resulting in 256 patches).



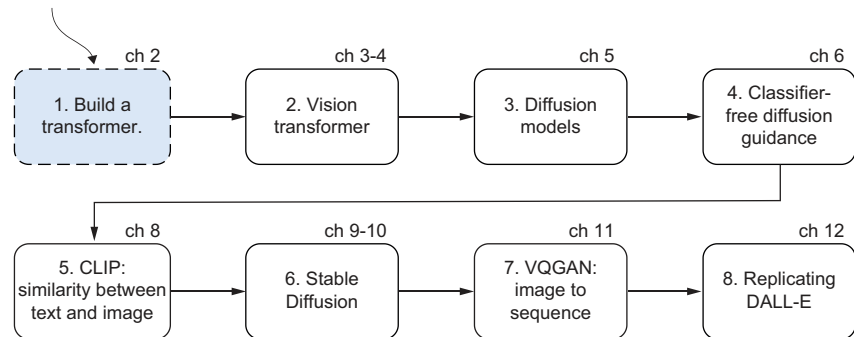
The transformer then generates these patches one by one, predicting the next patch in the sequence based on the text prompt and the patches generated so far. This sequential approach borrows from the same techniques used in language translation. At the core of this process is the attention mechanism, which allows the model to focus on the most relevant parts of the input when making predictions.

- Even when models aren't explicitly based on transformers, such as diffusion models, attention mechanisms are still essential. The denoising U-Net backbone used in diffusion models relies on attention to selectively focus on relevant parts of the input. In addition, when generating images from text prompts, diffusion models use a multimodal transformer such as contrastive language-image pretraining (CLIP) to encode both images and text into a shared latent space, allowing the model to align generated images with textual descriptions accurately.

In short, whether you're building DALL-E-like transformer models or advanced diffusion models, attention and transformers are everywhere. That's why we start here. By implementing the attention mechanism and building a transformer from scratch, you'll gain the intuition and coding skills needed for all subsequent chapters.

Figure 2.1 charts the eight key steps you'll follow to build a complete text-to-image generator from scratch. This chapter is the crucial first step: learning to build and train an encoder-decoder transformer for tasks such as translating German to English. By mastering how attention works at the code level, you'll establish the core skills needed for more advanced architectures, eventually culminating in your own text-to-image generator.

**Build and train an encoder-decoder transformer to translate German to English; code attention from scratch.**



**Figure 2.1** The eight-step road map for building a text-to-image generator from scratch. This chapter launches your journey by guiding you through the construction and training of an encoder-decoder transformer for German-to-English translation, giving you hands-on experience with attention mechanisms and transformer architectures, the essential building blocks for all subsequent steps.

In the rest of this chapter, you'll explore the inner workings of the attention mechanism, including query, key, and value vectors, and how scaled dot-product attention (SDPA) is calculated. You'll conduct a line-by-line implementation of the seminal "Attention Is All You Need" paper by Vaswani et al. [1], which I'll refer to often in this chapter. Specifically, you'll train the transformer on a real dataset with 29,000 German-to-English sentence pairs. This prepares you for building text-to-image models later in the book. By the end of this chapter, you'll have coded a transformer from scratch and demystified the "black box" of attention. This prepares you for building text-to-image models later in the book. Let's dive in and see why attention truly is all you need.

## 2.1 *An overview of attention and transformers*

Transformers are advanced deep-learning models that excel in handling sequence-to-sequence prediction challenges. Their strength lies in effectively understanding the relationships between elements in sequences over long distances, such as two words far apart in the text (e.g., 100 words away from each other).

The revolutionary aspect of the transformer architecture is its attention mechanism. This mechanism assesses the relationship between words in a sequence by assigning weights, determining the degree of relatedness in meaning among words based on the training data. This enables models such as ChatGPT to comprehend relationships between words, thus "understanding" human language more effectively.

In this section, we'll dive deep into the attention mechanism, exploring how it works. This process is crucial for determining the importance, or weights, of various words within a sentence. After that, we'll examine the structure of different transformer models, including one that can translate between any two languages.

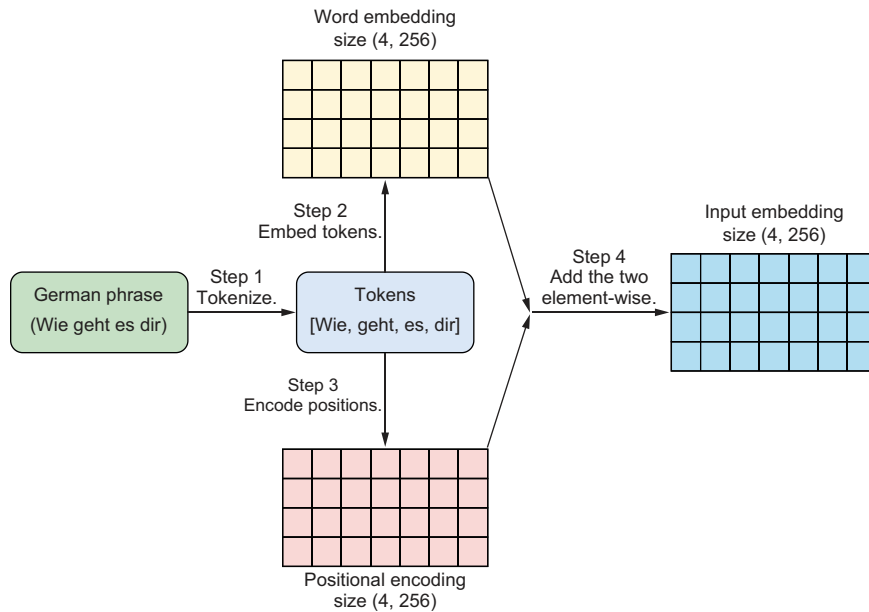
### 2.1.1 *How the attention mechanism works*

The *attention mechanism* is an algorithm designed to compute the relationships between any two elements in a sequence. It assigns scores to indicate how one element is related to all other elements, including itself, within the sequence. A higher score signifies a stronger relationship between the two elements. In natural language processing (NLP), the attention mechanism enables the model to meaningfully relate one word to other words in a sentence.

This chapter will guide you through implementing the attention mechanism for language translation, using German to English translation as an example. We'll construct a transformer composed of an encoder and a decoder: the encoder transforms a German sentence, such as *Wie geht es dir* (How are you), into vector representations that capture its meaning. The decoder then uses these vector representations to generate the English translation.

Like other neural networks, transformers can't take raw text as inputs. Therefore, we'll first convert German and English phrases into input embeddings, which we can feed into our transformer model. Figure 2.2 illustrates how we convert raw text into

input embeddings using our example German phrase of “Wie geht es dir” (How are you?), which is broken down into a sequence of smaller semantic units known as tokens ([Wie, geht, es, dir]). Each token is then transformed into a word embedding that captures its meaning. Positional encodings are also created to retain the order of tokens within the sequence. Finally, the word embeddings and positional encodings are combined element-wise to produce the final input embedding. We perform the same procedures to create word embeddings, positional encodings, and eventually input embedding for English phrases.



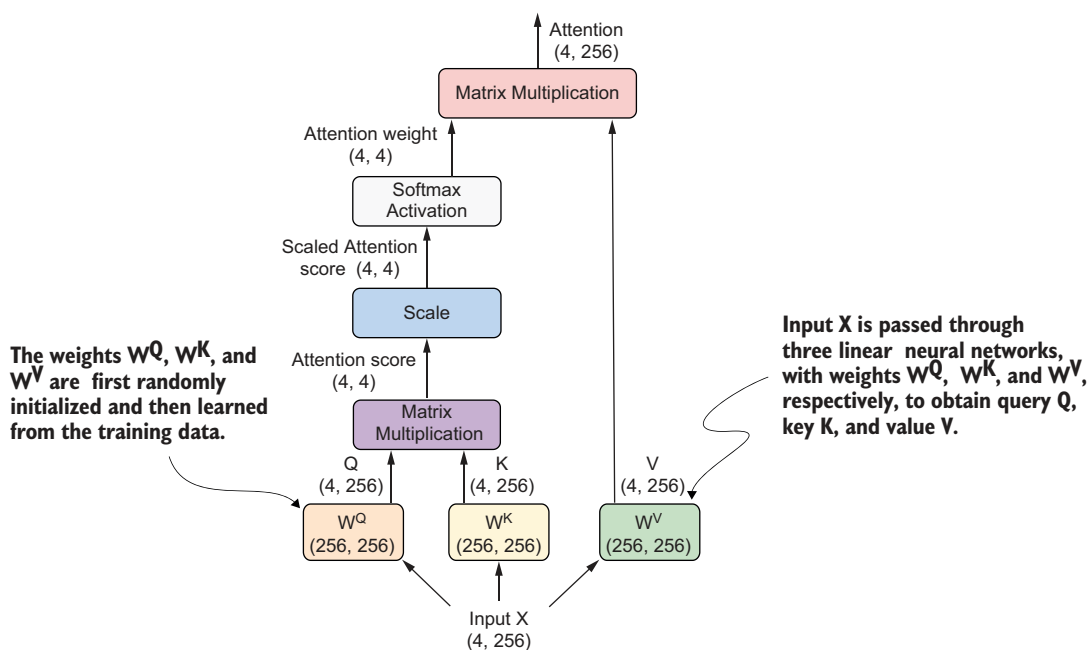
**Figure 2.2** How to convert raw text into input embeddings that we can feed into the transformer model

To transform the phrase “Wie geht es dir” into vector representations, the model first breaks it down into tokens [Wie, geht, es, dir], a process known as *tokenization* (step 1 in figure 2.2). *Tokens* are words or parts of words that act as semantic units in a language. Many words have prefixes or suffixes (e.g., *dis-agree* and *agree-ment*), and subword tokenization (using parts of words) allows such variations to share root language units, improving efficiency and contextual representation. The tokens for the phrase “Wie geht es dir” are each converted to a 256-dimensional vector known as word embedding, which captures the meaning of each token (step 2).

The encoder also employs *positional encoding*, a method to determine the positions of tokens in the sequence (step 3). The positional encodings are added to the word embeddings to create input embeddings (step 4), which are then used to feed into our

transformer model to calculate self-attention. The input embedding for “Wie geht es dir” forms a tensor with dimensions (4, 256), where 4 represents the number of tokens and 256 is the dimensionality of each embedding. If you recall from linear algebra, a *tensor* is a multidimensional array used to represent data. Word embedding, positional encoding, and input embeddings are all filled with floating-point numbers (e.g.,  $-0.192$ ,  $0.347$ ), and we’ll discuss how to calculate them in detail later.

While there are different ways to calculate attention, we’ll use the most common method, scaled dot product attention (SDPA). This mechanism is also called *self-attention* because the algorithm calculates how a word attends to all words in the sequence, including the word itself. Figure 2.3 provides a diagram of how to calculate self-attention.



**Figure 2.3** A diagram of the self-attention mechanism

To calculate SDPA, the input embedding  $X$  is processed through three distinct neural network layers. The corresponding weights for these layers are  $W^Q$ ,  $W^K$ , and  $W^V$ ; each has a dimension of  $256 \times 256$ . These weights are first randomly initialized and then learned from data during training. Thus, we can calculate query  $Q$ , key  $K$ , and value  $V$  as  $Q = X * W^Q$ ,  $K = X * W^K$ , and  $V = X * W^V$ . The dimensions of  $Q$ ,  $K$ , and  $V$  match those of the input embedding  $X$ , which are  $4 \times 256$  for “Wie geht es dir,” as we just explained.

The concepts of query, key, and value are borrowed from retrieval systems. Imagine visiting a public library to find a book. When you enter the phrase “generative AI with PyTorch” into the library’s search engine, that phrase serves as your query. The book titles and descriptions act as keys. The system then compares your query with these keys and returns a list of books, which represent the values. Books featuring “generative AI,” “PyTorch,” or both in their titles or descriptions tend to rank higher, while those less related to these terms score lower and are less likely to be recommended.

We assess the similarities between the query and key vectors using the SDPA approach. SDPA involves calculating the dot product of the query ( $Q$ ) and key ( $K$ ) vectors (the key vector needs to be transposed in the process for dimensions to match). A high dot product indicates a strong similarity between the two vectors and vice versa. The scaled attention score is computed as

$$\text{AttentionScore}(Q, K) = \frac{Q * K^T}{\sqrt{d_k}} \quad (2.1)$$

where  $d_k$  represents the dimension of the key vector  $K$ , which in our case is 256. We scale the dot product of  $Q$  and  $K$  by the square root of  $d_k$  to stabilize training. This scaling is done to prevent the dot product from growing too large in magnitude. The dot product between the query and key vectors can become very large when the dimension of these vectors (i.e., the depth of the embedding) is high. This is because each element of the query vector is multiplied by each element of the key vector, and these products are then summed up.

The next step is to apply the softmax function to these attention scores, converting them into attention weights (recall that the softmax function transforms a vector of raw scores into probabilities). This ensures that the total attention a word gives to all words in the sentence sums to 100%. The final attention is then calculated as the dot product of these attention weights and the value vector  $V$ :

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{Q * K^T}{\sqrt{d_k}}\right) * V \quad (2.2)$$

To summarize, the process begins with the input embedding  $X$  of the sentence “Wie geht es dir,” which has a dimension of  $4 \times 256$  (4 is the number of tokens, and 256 is the number of values in each embedding). This embedding captures the meanings of the four individual tokens but lacks contextualized understanding. The attention mechanism ends with the output  $\text{attention}(Q, K, V)$ , which maintains the same dimension of  $4 \times 256$ . This output can be viewed as a contextually enriched combination of the original four tokens. The weighting of the original tokens varies based on the contextual relevance of each token, granting more significance to words that are more important within the sentence’s context. Through this procedure, the attention mechanism

transforms vectors representing isolated tokens into vectors imbued with contextualized meanings, thereby extracting a richer, more nuanced understanding from the sentence.

Consider the word “bank” as an example of how the attention mechanism assigns meaning based on context. In the sentence “I went fishing by the river in the morning, staying near the bank,” the term “bank” connects with “fishing” because it describes the riverside area. Here, the transformer interprets “bank” as part of the natural river landscape.

Contrast this with the sentence “Alice went to the bank after work and deposited a check,” where “bank” is associated with “check,” leading the transformer to recognize it as a financial institution. This clearly demonstrates how transformers use surrounding context to derive word meanings.

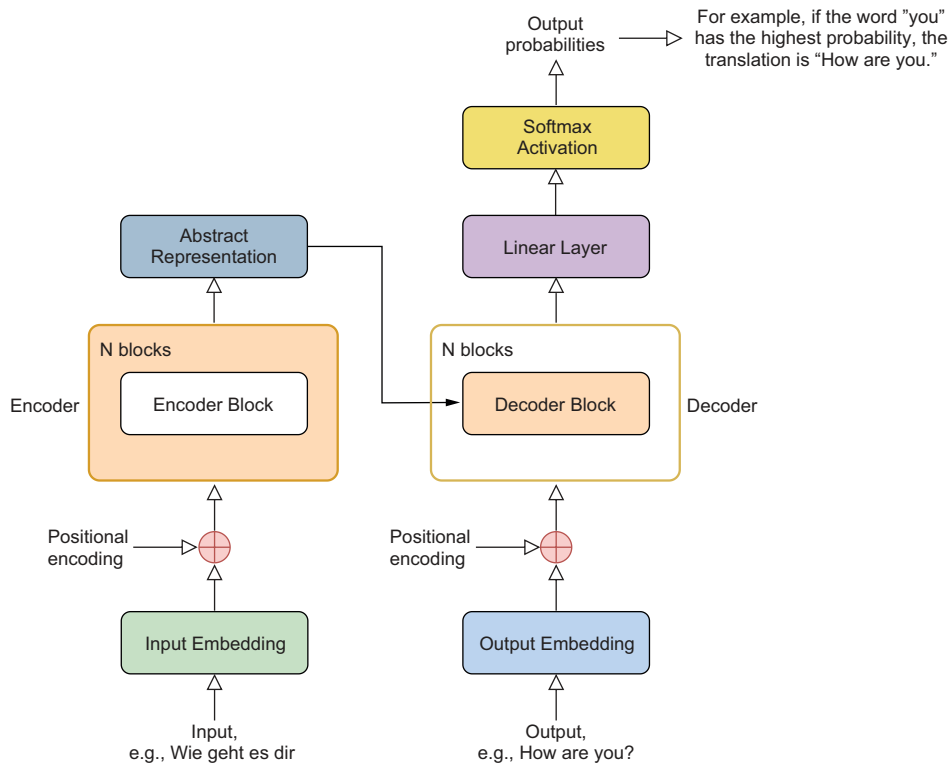
Additionally, instead of using one set of query, key, and value vectors, transformer models use a concept called *multi-head attention*. For example, the 256-dimensional query, key, and value vectors can be split into, say, 8 heads, and each head has a set of query, key, and value vectors with dimensions of 32 (because  $256 \div 8 = 32$ ). Each head focuses on different aspects of the input, allowing the model to capture a wider range of information and develop a more detailed, context-aware understanding. This multi-head approach is particularly beneficial when a word has multiple meanings in a sentence, such as in a pun. Extending the “bank” example, consider this pun: “Why is the river so rich? Because it has two banks.” In this case, multi-head attention can simultaneously grasp the different interpretations of the word “bank.”

### 2.1.2 *How to create a transformer*

The self-attention mechanism and transformers became widely used after the groundbreaking paper “Attention Is All You Need,” which focused on creating a model for machine language translation [1]. The architecture of the transformer is depicted in figure 2.4. It features an encoder–decoder structure that relies heavily on the attention mechanism.

The encoder in the transformer (left side of the diagram), which consists of  $N$  identical encoder layers, learns the meaning of the input sequence and converts it into a tensor that represents this meaning. It then passes this tensor to the decoder (right side of the diagram), which consists of  $N$  identical decoder layers. The decoder constructs the output (e.g., the English translation of a German phrase) by predicting one token at a time based on previous tokens in the sequence and the output from the encoder. The generator on the top right is the head attached to the output from the decoder so that the output is the probability distribution over all tokens in the target language (e.g., the English vocabulary). In this chapter, you’ll build this model from scratch, coding it line by line, and train it to translate German to English.

The encoder in the transformer converts a German phrase such as “Wie geht es dir” into a tensor that holds the meaning of the phrase. The decoder in the transformer then takes the tensor and converts it to the English translation “How are you.” The



**Figure 2.4** An encoder–decoder transformer for machine language translation (e.g., German to English)

encoder’s job is to extract the meaning of the German phrase. The decoder generates an English translation based on the output from the encoder.

Here’s how the encoder in the transformer extracts the meaning of the German phrase. First, the model converts German and English sentences into tokens. For example, “Wie geht es dir” has four tokens, [Wie, geht, es, dir], while “How are you” has three tokens: [How, are, you].

Because machine learning models can’t process text directly, we need to convert text into numbers before feeding the input to the models. Word embedding, which we’ll discuss in section 2.2, converts each token into a continuous vector of a fixed length, say, 256 values (256 is called the embedding dimension in this case). After word embedding, the sentence “Wie geht es dir” is represented by a matrix of  $4 \times 256$ .

Transformers process input data such as sentences in parallel, which enhances their efficiency but doesn’t inherently allow them to recognize the sequence order of the input. To address this, transformers add positional encodings to the word embeddings. These positional encodings are unique vectors assigned to each position in the input sequence and align in dimension with the input embeddings. The vector values are

determined by a specific positional function, particularly involving sine and cosine functions of varying frequencies, defined as

$$\text{PositionalEncoding}(pos, 2i) = \sin\left(\frac{pos}{n^{2i/d}}\right) \quad (2.3)$$

$$\text{PositionalEncoding}(pos, 2i + 1) = \cos\left(\frac{pos}{n^{2i/d}}\right) \quad (2.4)$$

These equations apply the sine function to even indices and the cosine function to odd indices. Here, *pos* indicates a token's position in the sequence, while *i* represents the index within the vector. For instance, the positional encoding for the phrase “Wie geht es dir” is a  $4 \times 256$  matrix, matching the dimensions of the sentence's word embedding. In this example, *pos* ranges from 0 to 3, and the indices  $2i$  and  $2i + 1$  together cover 256 unique values (from 0 to 255). The positional encoding for “Wie geht es dir” is shown here:

```
tensor([[ 0.0000e+00,  1.0000e+00,  0.0000e+00, ...,  1.0000e+00,
          0.0000e+00,  1.0000e+00],
        [ 8.4147e-01,  5.4030e-01,  8.0196e-01, ...,  1.0000e+00,
          1.0746e-04,  1.0000e+00],
        [ 9.0930e-01, -4.1615e-01,  9.5814e-01, ...,  1.0000e+00,
          2.1492e-04,  1.0000e+00],
        [ 1.4112e-01, -9.8999e-01,  3.4278e-01, ...,  1.0000e+00,
          3.2238e-04,  1.0000e+00]], device='cuda:0')
```

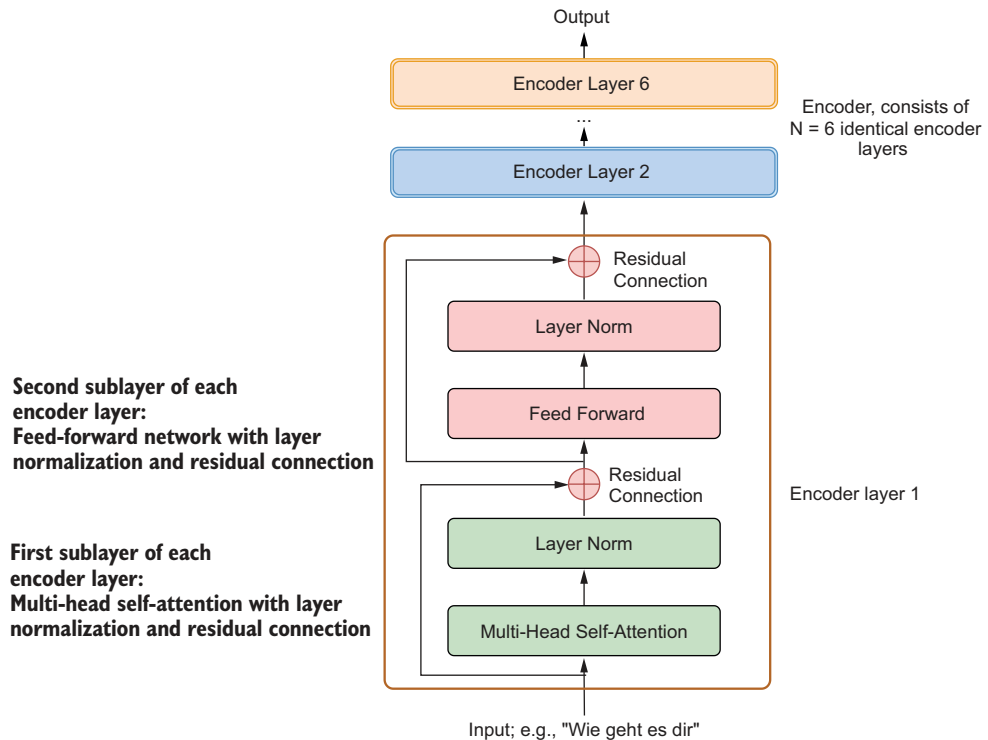
The positional encoding has a shape of  $4 \times 256$ : there are four elements in the sequence, and each element's position is represented by a 256-value tensor.

The transformer's encoder, as depicted in figure 2.5, is made up of six identical layers ( $N = 6$ ). Each of these layers comprises two sublayers. The first sublayer is a multi-head self-attention layer, similar to what was discussed earlier. The second sublayer is a basic, position-wise, fully connected feed-forward network. Each sublayer uses layer normalization and residual connection. This network treats each position in the sequence independently rather than as sequential elements. The feed-forward network further refines the output from the self-attention layer, allowing the model to learn complex patterns and extract deeper semantic information from the input data.

Each sublayer in the encoder incorporates layer normalization and a residual connection. Layer normalization normalizes observations to have zero mean and unit standard deviation. Such normalization helps stabilize the training process. After the normalization layer, we perform the residual connection. This means the input to each sublayer is added to its output, enhancing the flow of information through the network.

The decoder of the transformer model, as shown in figure 2.6, has six identical decoder layers ( $N = 6$ ). Each decoder layer contains three sublayers. The first is a masked multi-head self-attention layer. The second is a multi-head cross-attention layer





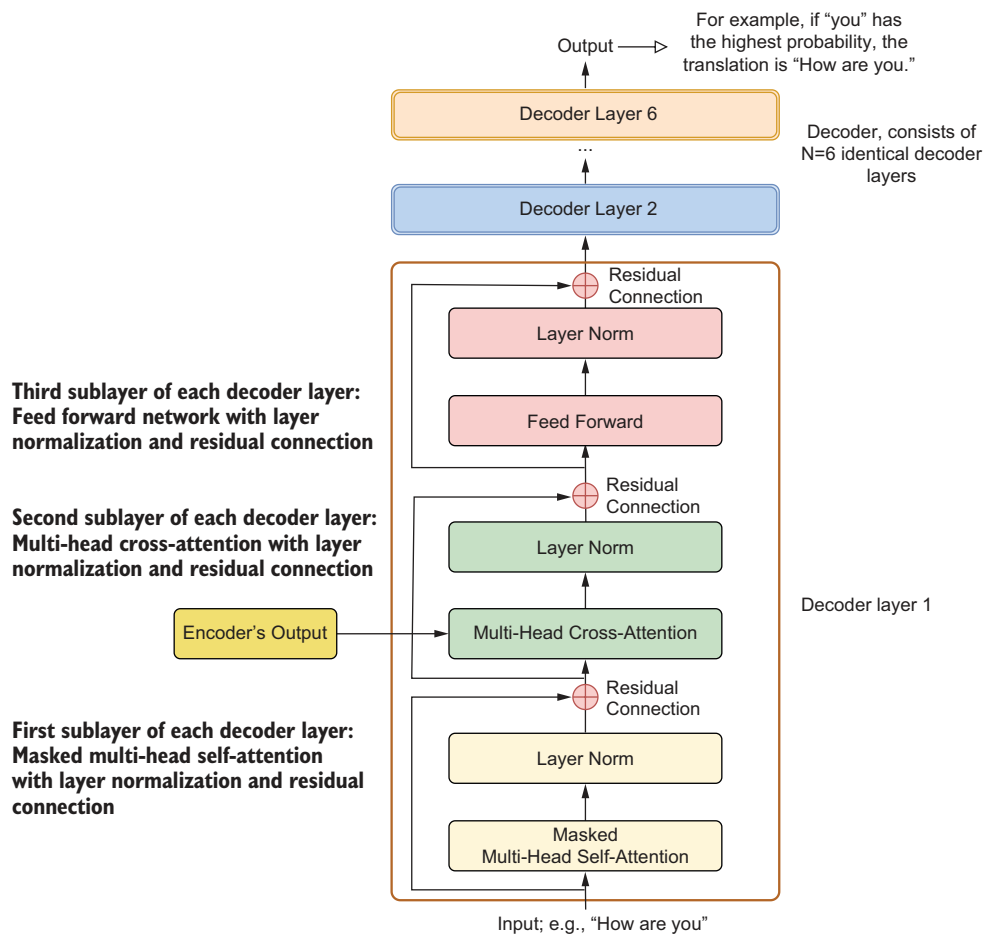
**Figure 2.5** The structure of the encoder in the transformer. The input to the encoder is the embedding of the German phrase (e.g., "Wie geht es dir"), and the output is an abstract representation that captures the meaning of the phrase.

to calculate the cross attention between the output from the first sublayer and the output from the encoder. The third sublayer is a feed-forward network. Each sublayer uses layer normalization and residual connection.

Note that the input to each sublayer is the output from the previous sublayer. The second sublayer in each decoder layer also takes the output from the encoder as input. This is how the decoder generates translations based on the output from the encoder.

The masking mechanism is a key aspect of the decoder's self-attention sublayer. This mask prevents the model from accessing future positions in the sequence, ensuring that predictions for a particular position can only depend on previously known elements. This sequential dependency is vital for tasks such as language translation or text generation. I'll explain the masking mechanism in more detail in section 2.2.2.

The decoding process begins with the encoder receiving an input phrase in German. The encoder transforms the German tokens into word embeddings and positional encodings before combining them into a single embedding. This step ensures that the model not only understands the semantic content of the phrase but also maintains the sequential context, which is crucial for accurate translation or generation tasks.



**Figure 2.6** The structure of the decoder in the transformer

We'll use the decoder to generate the output sequence one token at a time. In the first time step, we start with the BOS token, which indicates the beginning of a sentence. The decoder uses the vector representations of the German phrase "Wie geht es dir" produced by the encoder and attempts to predict the first English token following BOS. Suppose the decoder's first prediction is How. In the next time step, it uses the sequence BOS How as its new input to predict the following token. This process continues iteratively, with the decoder adding each newly predicted token to its input sequence for the subsequent prediction.

Note that the decoder operates as a sequence-to-sequence predictor, meaning it outputs a sequence rather than a single token. In the second iteration of the example we discussed in the previous paragraph, the decoder receives BOS How as its input, which consists of two tokens. Consequently, it produces an output sequence with two tokens:

the first token predicts what should follow BOS, and the second predicts what should follow How. However, because How already follows BOS, we only need to focus on the token that comes after How. Thus, during the second iteration, our goal is to determine the token that should follow BOS How.

The translation process stops when the decoder predicts the EOS token, indicating the end of the sentence. When preparing for the training data, we add EOS to the end of each phrase, so the model has learned that it means “end of sentence.” Upon reaching this token, the decoder recognizes the completion of the translation task and ceases its operation. This autoregressive approach ensures that each step in the decoding process is informed by all previously predicted tokens, allowing for coherent and contextually appropriate translations.

Besides encoder–decoder transformers such as the model we just discussed, there are two other types of transformers: encoder-only and decoder-only transformers. An encoder-only transformer consists of  $N$  identical encoder layers, as shown earlier on the left side of figure 2.4, and is capable of converting a sequence into abstract continuous vector representations. For example, BERT is an encoder-only transformer that contains 12 encoder layers. A decoder-only transformer also consists of  $N$  identical layers, and each layer is a decoder layer similar to the right side of figure 2.4. For example, ChatGPT is a decoder-only transformer that contains many decoder layers. The decoder-only transformer can generate text based on a prompt, for example. It extracts the semantic meaning of the words in the prompt and predicts the most likely next token.

The machine language translation transformer we discussed earlier in the chapter is an example of an encoder–decoder transformer. These transformers are needed for handling complicated tasks, such as text-to-image generation or speech recognition. Encoder–decoder transformers combine the strengths of both encoders and decoders. Encoders are efficient in processing and understanding input data, while decoders excel in generating output. This combination allows the model to effectively understand complex inputs (e.g., text or speech) and generate sophisticated outputs (e.g., images or transcribed text).

## 2.2 *Word embedding and positional encoding*

To make the explanation of self-attention and transformers more relatable, I’ll use German-to-English translation as our example. By working through the example of training a model to translate German phrases to English, you’ll get a deep understanding of various components of a transformer and how the attention mechanism works.

Imagine that you’ve collected 29,000 pairs of German-to-English translations from online sources. Your goal is to create a machine learning model and train the model by using the dataset. I’ll describe how you can go about this task in the following subsections. (The Python programs in this project are adapted from the work of Chris Cui in translating Chinese to English [<https://mng.bz/oZo2>] and Alexander Rush’s German-to-English translation project [<https://mng.bz/64qZ>]).

**NOTE** The Python programs for this chapter can be accessed on the book's GitHub repository (<https://github.com/markhliu/txt2img>) and are also available as a Google Colab notebook (<https://mng.bz/nZ6e>) for interactive exploration. Don't just copy and paste the book's code snippets into a new Colab notebook and run them. Instead, use the Colab notebooks I've provided, as Google Colab has a unique filesystem that requires specific setup. The book's appendix includes guidance on training machine learning models using Google's free GPU resources.

Follow the instructions in the appendix to install PyTorch on your computer. After that, run the following code cell in Jupyter Notebook to install the Spacy library for this chapter:

```
!pip install spacy
```

### 2.2.1 Word tokenization with the Spacy library

We first need to download the training dataset that consists of pairs of German-to-English translations. Run the following code block to download and unzip the training dataset file.

#### Listing 2.1 Downloading the training dataset

```
import requests, os, tarfile

url=("https://raw.githubusercontent.com/neycher/"
     "small_DL_repo/master/datasets/Multi30k/training.tar.gz")
os.makedirs("files", exist_ok=True)
if not os.path.exists("files/training.tar.gz"):
    fb1=requests.get(url)
    with open("files/training.tar.gz","wb") as f:
        f.write(fb1.content)
train=tarfile.open('files/training.tar.gz')
train.extractall('files')
train.close()
```

The URL to download the training dataset

Downloads the dataset to your computer

Unzips the file

Places content in the /files/ folder

Download the file `training.tar.gz` online, and place it in the folder `/files/` on your computer. After execution, two files, `train.en` and `train.de`, will be saved in the `/files/` folder on your computer. We then read the content of the two files into two lists:

```
with open("files/train.de", 'rb') as fb:
    trainde = fb.readlines()
with open("files/train.en", 'rb') as fb:
    trainen = fb.readlines()
trainde=[i.decode("utf-8").strip() for i in trainde]
trainen=[i.decode("utf-8").strip() for i in trainen]
```

The file `train.de` contains 29,001 German phrases, while `train.en` contains the corresponding English translations. The preceding code block reads these phrases and places them in two lists, `trainde` and `trainen`, respectively. We can check the length of each list and print out the first five elements in each list as follows:

```
from pprint import pprint

print(f"the length of the list trainde is {len(trainde)}")
print(f"the length of the list trainen is {len(trainen)}")
print(f"the first five elements of the list trainde are")
pprint(trainde[:5])
print(f"the first five elements of the list trainen are")
pprint(trainen[:5])
```

The output is shown here:

```
the length of the list trainde is 29001
the length of the list trainen is 29001
the first five elements of the list trainde are
[,Zwei junge weiße Männer sind im Freien in der Nähe vieler Büsche.',
 ,Mehrere Männer mit Schutzhelmen bedienen ein Antriebsradsystem.',
 ,Ein kleines Mädchen klettert in ein Spielhaus aus Holz.',
 ,Ein Mann in einem blauen Hemd steht auf einer Leiter und putzt ein
 Fenster.',
 ,Zwei Männer stehen am Herd und bereiten Essen zu.']
the first five elements of the list trainen are
['Two young, White males are outside near many bushes.',
 'Several men in hard hats are operating a giant pulley system.',
 'A little girl climbing into a wooden playhouse.',
 'A man in a blue shirt is standing on a ladder cleaning a window.',
 'Two men are at the stove preparing food.']
```

Both lists have 29,001 phrases. The preceding output shows five German phrases, followed by their English translations.

### Exercise 2.1

Print out the last five elements in the lists `trainde` and `trainen`.

We'll use the tokenizers in the Spacy library to convert both German and English phrases into tokens. First, let's define the two tokenizers as follows:

```
import os, spacy

try:
    de_tokenizer = spacy.load("de_core_news_sm")
except IOError:
    os.system("python -m spacy download de_core_news_sm")
    de_tokenizer = spacy.load("de_core_news_sm")

try:
    en_tokenizer = spacy.load("en_core_web_sm")
except IOError:
    os.system("python -m spacy download en_core_web_sm")
    en_tokenizer = spacy.load("en_core_web_sm")
```

**Tries to load the German model**

**If the German model isn't found, downloads it to your computer**

**Tries to load the English model**

**If the English model isn't found, downloads it to your computer**

The models `de_core_news_sm` and `en_core_web_sm` are language-specific pipelines provided by spaCy, designed for processing German and English text, respectively. The `sm` indicates that it's a small, lightweight model, making it faster and less resource-intensive, though it might be less accurate compared to larger models. We name the English tokenizer `en_tokenizer` and the German tokenizer `de_tokenizer`.

Let's use the two tokenizers to convert the first German phrase and the first English phrase into tokens and print them out:

```
tokenized_de= [tok.text for tok in
                de_tokenizer.tokenizer(trainde[0])]
tokenized_en=[tok.text for tok in
                en_tokenizer.tokenizer(trainen[0])]
print(tokenized_de)
print(tokenized_en)
```

The output is as follows:

```
['Zwei', 'junge', 'weiße', 'Männer', 'sind', 'im', 'Freien', 'in',
 'der', 'Nähe', 'vieler', 'Büsche', '.']
['Two', 'young', ',', 'White', 'males', 'are', 'outside', 'near',
 'many', 'bushes', '.']
```

The tokens in the preceding output are either individual words or punctuation. However, it's possible that some words are broken down into subwords.

### Exercise 2.2

Convert the second German phrase in `trainde` into a list of tokens, and name the list `tokenized_de1`. Then, convert the second English phrase in `trainen` into a list of tokens, and name the list `tokenized_en1`. Print out the lists `tokenized_de1` and `tokenized_en1`.

Because deep neural networks can't take tokens as inputs directly, we need to convert tokens into numbers so we can feed them to the transformer. Next, we build a dictionary for English tokens and map each unique token to a different index.

### Listing 2.2 Creating a dictionary to map English tokens to indices

```
from collections import Counter

en_tokens=[["BOS"]+[tok.text for tok in en_tokenizer.tokenizer(x)]
            +["EOS"] for x in trainen]

PAD=0
UNK=1
word_count=Counter()
for sentence in en_tokens:
    for word in sentence:
```

**Adds BOS and EOS at the beginning and end of each phrase, respectively**

```

word_count[word]+=1
frequency=word_count.most_common(50000)
total_en_words=len(frequency)+2

en_word_dict={w[0]:idx+2 for idx,w in enumerate(frequency)}
en_word_dict["PAD"]=PAD
en_word_dict["UNK"]=UNK

en_idx_dict={v:k for k,v
             in en_word_dict.items()}

```

Assigns an index to each unique token

The padding token and unknown tokens are assigned indices 0 and 1, respectively.

A dictionary to map indices back to tokens

We add tokens BOS (beginning of sentence) and EOS (end of sentence) to the start and end of each phrase. The dictionary `en_word_dict` maps each unique token to a different index. We also create another dictionary, `en_idx_dict`, to map indices back to their corresponding tokens. This reverse mapping allows us to convert a sequence of indices back into a sequence of tokens to reconstruct the original English phrase. The PAD token is used at the end of shorter sequences in a batch so that all sequences are the same length. The UNK token is used to represent unknown tokens. This is useful in case we encounter a prompt with unknown tokens, allowing us to convert them to an index to feed to the model.

Using the dictionary `en_word_dict`, we can transform the first English sentence into its numerical representation:

```

enidx=[en_word_dict.get(i,UNK) for i in tokenized_en]
print(enidx)

```

The output is

```
[19, 25, 15, 1165, 804, 17, 57, 84, 334, 1329, 5]
```

We can also convert the sequence of indices back to the original English phrase:

```

entokens=[en_idx_dict.get(i,"UNK") for i in enidx]
print(entokens)
en_phrase=" ".join(entokens)
for x in '":;.,'("-&)%':
    en_phrase=en_phrase.replace(f" {x}",f"{x}")
print(en_phrase)

```

The output is

```

['Two', 'young', ',', 'White', 'males', 'are', 'outside',
'near', 'many', 'bushes', '.']
Two young, White males are outside near many bushes.

```

**Exercise 2.3**

Use the dictionary `en_word_dict` to convert `tokenized_en1` that you created in exercise 2.2 into a list of indices `enidx1`. Then, convert `enidx1` back into a list of tokens `entokens1` using the dictionary `en_idx_dict`. Finally, join the tokens in `entokens1` into a sentence.

Next, we build a dictionary for German tokens and map each unique token to a different index:

```
de_tokens= ["BOS"]+[tok.text for tok in de_tokenizer.tokenizer(x)]
              +["EOS"] for x in trainde]
de_word_count=Counter()
for sentence in de_tokens:
    for word in sentence:
        de_word_count[word]+=1
defrequency=de_word_count.most_common(50000)
total_de_words=len(defrequency)+2
de_word_dict={w[0]:idx+2 for idx,w
              in enumerate(defrequency)}
de_word_dict["PAD"]=PAD
de_word_dict["UNK"]=UNK
de_idx_dict={v:k for k,v in de_word_dict.items()}
```

← Adds BOS and EOS at the beginning and end of each phrase, respectively

← Assigns an index to each unique token

← The padding token and unknown tokens are assigned indices 0 and 1, respectively.

← A dictionary to map indices back to tokens

The dictionary `de_word_dict` maps each unique German token to an integer, and the dictionary `de_idx_dict` maps integers back to German tokens. Here, we convert the first German phrase to numerical representations:

```
deidx=[de_word_dict.get(i,UNK) for i in tokenized_de]
print(deidx)
```

The output is

```
[21, 85, 257, 31, 87, 22, 94, 7, 16, 112, 5497, 3161, 4]
```

The following code cell converts the preceding numerical representations back to German tokens and restores the original German phrase:

```
detokens=[de_idx_dict.get(i,"UNK") for i in deidx]
print(detokens)
de_phrase=" ".join(detokens)
for x in ' ' '?;.,'(''-!&)%''':
    de_phrase=de_phrase.replace(f" {x}",f"{x}")
print(de_phrase)
```

The output is



```
[,Zwei', ,junge', ,weiße', ,Männer', ,sind', ,im', ,Freien',
'in', 'der', 'Nähe', 'vieler', 'Büsche', '.']
Zwei junge weiße Männer sind im Freien in der Nähe vieler Büsche.
```

### Exercise 2.4

Use the dictionary `de_word_dict` to convert `tokenized_de1` that you created in exercise 2.2 into a list of indices, `deidx1`. Then, convert `deidx1` back into a list of tokens, `detokens1`, using the dictionary `de_idx_dict`. Finally, join the tokens in `detokens1` into a sentence.

Now that you know how to tokenize and index both German and English phrases, you can arrange them into batches to use for training the transformer later.

## 2.2.2 A sequence padding function

The numerical representations that we feed to the transformer must have the same length in each batch. However, the English and German phrases have various lengths. To overcome this, we add 0s at the end of numerical representations of shorter phrases so all inputs to the transformer model have the same lengths in each batch.

First, we convert all English phrases to their numerical representations and do the same for German phrases:

```
out_en_ids=[[en_word_dict.get(w,UNK) for w in s]
            for s in en_tokens]
out_de_ids=[[de_word_dict.get(w,UNK) for w in s]
            for s in de_tokens]
sorted_ids=sorted(range(len(out_de_ids)),
                  key=lambda x:len(out_de_ids[x]))
out_de_ids=[out_de_ids[x] for x in sorted_ids]
out_en_ids=[out_en_ids[x] for x in sorted_ids]
```

Next, we put the numerical representations into batches for training:

```
import numpy as np

batch_size=128
idx_list=np.arange(0,len(de_tokens),batch_size)
np.random.shuffle(idx_list)

batch_indexes=[]
for idx in idx_list:
    batch_indexes.append(np.arange(idx,min(len(de_tokens),
                                          idx+batch_size)))
```

Note that we've sorted observations in the training dataset by the length of the German phrases before placing them into batches. This method ensures that the observations

within each batch are of a comparable length, consequently decreasing the need for padding. As a result, this approach not only reduces the overall size of the training data but also accelerates the training process.

To pad sequences, we define the following `seq_padding()` function:

```
def seq_padding(X, padding=PAD):
    L = [len(x) for x in X]
    ML = max(L)
    padded_seq = np.array([np.concatenate([x,
                                           [padding] * (ML - len(x))])
                          if len(x) < ML else x for x in X])
    return padded_seq
```

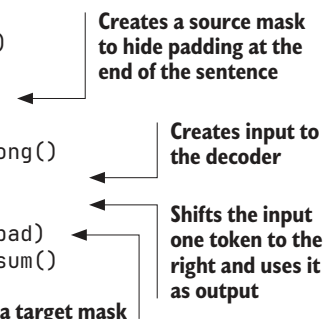
The function first finds the longest sequence in the batch and adds 0s at the end of other sequences so that all sequences in the batch have the same length.

To streamline the code, we define a `Batch` class in the local module `transformer_util`. Download the file `transformer_util.py` from the book's GitHub repository (<https://github.com/markhliu/txt2img>), and save it in the folder `/utils/` on your computer. Make sure that you place a blank file named `__init__.py` in the folder `/utils/` so that Python treats files in the folder as Python modules (or simply clone the repository to your computer). If you're working in Google Colab, you can quickly set up the repository and add it to your working directory by running

```
!git clone https://github.com/markhliu/txt2img
import sys
sys.path.append("/content/txt2img")
```

The `Batch` class is defined as follows in the local module:

```
class Batch:
    def __init__(self, src, trg=None, pad=0):
        src = torch.from_numpy(src).to(DEVICE).long()
        self.src = src
        self.src_mask = (src != pad).unsqueeze(-2)
        if trg is not None:
            trg = torch.from_numpy(trg).to(DEVICE).long()
            self.trg = trg[:, :-1]
            self.trg_y = trg[:, 1:]
            self.trg_mask = make_std_mask(self.trg, pad)
            self.ntokens = (self.trg_y != pad).data.sum()

        
```

The `Batch` class processes a batch of German and English phrases, converting them into a format suitable for training. To make this explanation more tangible, consider the German phrase “Wie geht es dir” and its English equivalent “How are you” as our example. The `Batch` class receives two inputs: `src`, which is the sequence of indices representing the tokens in “Wie geht es dir,” and `trg`, the sequence of indices for the

tokens in “How are you.” This class generates a tensor, `src_mask`, to conceal the padding at the sentence’s end.

The `Batch` class additionally prepares the input and target for the transformer’s decoder. The English phrase “How are you” is transformed into five tokens: `[BOS, how, are, you, EOS]`. The first four tokens serve as the input to the decoder, named `trg`. Next, we shift this input one token to the right to form the target, `trg_y`, which contains tokens `[how, are, you, EOS]`. Each token in the target sequence is the next token in the corresponding position in the input sequence. The input sequence is fed to the decoder to make predictions on what the next token should be. The target sequence serves as the ground truth. We compare the predicted output from the decoder with the target sequence and calculate the cross-entropy loss. During training, we modify model parameters to minimize this loss. This approach compels the model to predict the next token based on the previous ones.

The `Batch` class also generates a mask, `trg_mask`, for the decoder’s input. The purpose of this mask is to conceal the subsequent tokens in the input, ensuring that the model relies solely on previous tokens for making predictions. Text generation (in our case, generating English translation for German phrases) requires the model to predict the next token based only on tokens it has seen so far, not future tokens. If future tokens were visible during training, the model wouldn’t learn the proper dependencies. The mask ensures that the model’s behavior during training matches its behavior during inference, leading to better generalization. We can now import the `Batch` class from the `local` module and put the training data in batches:

```
from utils.transformer_util import Batch

batches=[]
for b in batch_indexes:
    batch_en=[out_en_ids[x] for x in b]
    batch_de=[out_de_ids[x] for x in b]
    batch_en=seq_padding(batch_en)
    batch_de=seq_padding(batch_de)
    batches.append(Batch(batch_de,batch_en))
```

The `batches` list contains data batches for training. Each batch has 128 pairs of numerical representations of German and English phrases, respectively.

### 2.2.3 Input embedding from word embedding and positional encoding

The number of indices in numerical representations of the German and English phrases is fairly large. We can find out how many distinct indices we need for each language by counting the number of elements in the dictionaries `en_word_dict` and `de_word_dict`:

```
src_vocab = len(de_word_dict)
tgt_vocab = len(en_word_dict)
print(f"there are {src_vocab} distinct German tokens")
print(f"there are {tgt_vocab} distinct English tokens")
```

The output is as follows:

```
there are 19214 distinct German tokens
there are 10837 distinct English tokens
```

There are 19,214 distinct German tokens and 10,837 distinct English tokens in our dataset. We'll use a vector of length `d_model=256`, for example, to represent each token. The method is called word embedding. The `Embeddings` class is defined in the local module `transformer_util` as follows:

```
from torch import nn
import math

class Embeddings(nn.Module):
    def __init__(self, d_model, vocab):
        super().__init__()
        self.lut = nn.Embedding(vocab, d_model)
        self.d_model = d_model
    def forward(self, x):
        out = self.lut(x) * math.sqrt(self.d_model)
        return out
```

Word embedding transforms tokens into dense, low-dimensional vectors that capture semantic relationships among them. These embeddings place similar words closer in vector space, making them more efficient for machine learning models to process compared to sparse one-hot encodings. Moreover, the embeddings are learned directly from the training dataset during the model's training process. To model the order of elements in the input and output sequences, we'll first create positional encodings of the sequences.

### Listing 2.3 Designing the `PositionalEncoding()` class

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout, max_len=5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)
        pe = torch.zeros(max_len, d_model, device=DEVICE)
        position = torch.arange(0., max_len,
                                device=DEVICE).unsqueeze(1)
        div_term = torch.exp(torch.arange(
            0., d_model, 2, device=DEVICE)
            * -(math.log(10000.0) / d_model))
        pe_pos = torch.mul(position, div_term)
        pe[:, 0::2] = torch.sin(pe_pos)
        pe[:, 1::2] = torch.cos(pe_pos)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
```

Initiates the class, allowing a maximum of 5,000 positions

Applies the sine function to even indices in the vector

Applies the cosine function to odd indices in the vector

```
x = x + self.pe[:, :x.size(1)].requires_grad_(
    False)
out = self.dropout(x)
return out
```

Adds positional encoding  
to word embedding

The `PositionalEncoding()` class generates vectors for sequence positions using sine functions for even indices and cosine functions for odd indices. One of the benefits of using these trigonometric functions is that their outputs range between  $-1$  and  $1$ . Additionally, it's important to note that `requires_grad(False)` means there's no need to train these values. They remain constant across all inputs, and they don't change during training.

## 2.3 Creating an encoder–decoder transformer

Following the methodology outlined in the “Attention Is All You Need” paper [1], we'll develop an encoder–decoder transformer to translate German to English. This section discusses the process of building the encoder and decoder. The encoder compresses a German sentence (e.g., “Wie geht es dir”) into a vector representation that captures its meaning. The decoder generates the English translation, “How are you,” based on the encoder's output. You'll learn how to construct various sublayers in the encoder and decoder, including how to implement the attention mechanism.

### 2.3.1 Coding the attention mechanism

The SDPA attention mechanism uses query, key, and value to calculate the relationships among elements in a sequence. It assigns scores to show how an element in a sequence is related to all other elements. The following listing defines the `attention()` function in the local module.

**Listing 2.4** Calculating attention based on query, key, and value

```
def attention(query, key, value, mask=None, dropout=None):
    d_k = query.size(-1)
    scores = torch.matmul(query,
                           key.transpose(-2, -1)) / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = nn.functional.softmax(scores, dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

The scaled attention score is the dot product of query and key, scaled by the square root of  $d_k$ .

If there's a mask, hide future elements in the sequence.

Calculates attention weights

Returns both attention and attention weights

The `attention()` function takes query, key, and value as inputs and calculates attention and attention weights as we discussed earlier in this chapter. The scaled attention score is the dot product of query and key, scaled by the square root of the dimension of the

key,  $d_k$ . We apply the softmax function on the scaled attention score to obtain attention weights. Finally, attention is calculated as the dot product of attention weights and value.

Further, instead of using one set of query, key, and value vectors, the transformer model uses a concept called multi-head attention. This way, each head captures a different aspect of the input data so the model can form a more detailed understanding of the text. Our 256-dimensional query, key, and value vectors are split into 8 heads, and each head has a set of query, key, and value vectors with dimensions of 32 (because  $256 \div 8 = 32$ ). This is implemented in the `MultiHeadedAttention()` class defined in the local module. For brevity, the exact definition isn't displayed here. If you're interested, refer to chapters 9 and 10 of my book *Learn Generative AI with PyTorch* (Manning, 2024).

Each encoder layer and decoder layer also contains a feed-forward sublayer. It doesn't treat the sequence of embeddings as a single vector, so we often call it a position-wide feed-forward network (or a 1D convolutional network). The network is defined in the `PositionwiseFeedForward()` class in the local module `transformer_util.py`.

### 2.3.2 Defining the Transformer() class

To create an encoder-decoder transformer, we define a `Transformer()` class in the local module `transformer_util.py`, as shown in the following listing.

**Listing 2.5** Defining the `Transformer()` class

```
class Transformer(nn.Module):
    def __init__(self, encoder, decoder,
                 src_embed, tgt_embed, generator):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator
    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)
    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt),
                             memory, src_mask, tgt_mask)
    def forward(self, src, tgt, src_mask, tgt_mask):
        memory = self.encode(src, src_mask)
        output = self.decode(memory, src_mask,
                             tgt, tgt_mask)
        return output
```

**Defines an encoder in the transformer**

**Defines a decoder in the transformer**

**The source language is encoded into an abstract vector representation by the encoder.**

**The decoder uses the vector representation to generate the translation in the target language.**

The transformer consists of an encoder and a decoder. The encoder converts the numerical representation of a German phrase into an abstract representation (called memory in the preceding `Transformer()` class). The decoder then takes the output from the encoder and generates the translation in an autoregressive fashion: it

generates one element at a time, based on the previously generated elements and the output from the encoder.

The encoder consists of  $N = 6$  identical encoder layers. The `Encoder()` class is defined as follows in the local module:

```
class Encoder(nn.Module):
    def __init__(self, layer, N):
        super().__init__()
        self.layers = nn.ModuleList(
            [deepcopy(layer) for i in range(N)])
        self.norm = LayerNorm(layer.size)
    def forward(self, x, mask):
        for layer in self.layers:
            x = layer(x, mask)
        output = self.norm(x)
        return output
```

Inside each encoder layer, there are two sublayers: a multi-head self-attention layer and a feed-forward layer. We apply layer normalization and residual connection in each sublayer. The normalization layer normalizes each of the inputs in a batch independently across all features so that they all have zero mean and unit standard deviation. This helps stabilize training. Residual connection means the input to each sublayer is added to the output of the sublayer.

The decoder in the transformer consists of  $N = 6$  identical decoder layers. Each decoder layer consists of a multi-head self-attention layer and a basic, position-wise, fully connected feed-forward network with residual connections, as shown in figure 2.6. In addition to these two sublayers, each decoder layer also has a third sublayer that applies multi-head attention over the encoder stack's output. Furthermore, the decoder stack's self-attention sublayer is masked to prevent positions from attending to subsequent positions. The mask forces the model to use previous elements in a sequence to predict later elements. Specifically, the `DecoderLayer()` class is defined in the local module as shown in the following listing.

#### Listing 2.6 Creating a decoder layer

```
class DecoderLayer(nn.Module):
    def __init__(self, size, self_attn, src_attn,
                 feed_forward, dropout):
        super().__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = nn.ModuleList([deepcopy(
            SublayerConnection(size, dropout)) for i in range(3)])
    def forward(self, x, memory, src_mask, tgt_mask):
        x = self.sublayer[0](x, lambda x:
            self.self_attn(x, x, x, tgt_mask))
        x = self.sublayer[1](x, lambda x:
```

**The first sublayer is a masked multi-head attention layer.**

```

self.src_attn(x, memory, memory, src_mask))
output = self.sublayer[2](x, self.feed_forward)
return output

```

The third sublayer is a feed-forward network.

The second sublayer is a cross-attention layer between the two languages.

Each decoder layer has three sublayers: a multi-head self-attention layer, a multi-head cross-attention, and a feed-forward network. To calculate the cross-attention in the second sublayer, we feed the output from the first sublayer, termed *x*, and the output of the encoder stack, termed *memory*, in the preceding code block, to the `attention()` class we defined earlier, where *x* is used as query, and *memory* is used as key and value.

The output from the decoder is the probability distribution over all tokens in the English vocabulary. Because there are 10,837 different English tokens in our dataset, when the input to the decoder contains 4 tokens—for example, [BOS, how, are, you]—the output will have a shape of  $4 \times 10,837$ . Here, 4 indicates the number of predicted tokens, and 10,837 is the number of elements in the probability distribution.

### 2.3.3 Creating a language translator

We define a `Generator()` class in the local module to generate the most likely next token. The output of the `Generator()` class is the probability distribution in the target language vocabulary. This allows the model to predict one token at a time in an autoregressive fashion, based on previously generated tokens and the output from the encoder.

Now we're ready to put all the pieces together and create our transformer model. The `create_model()` function defined in the local module accomplishes that.

**Listing 2.7** Creating a model to translate German to English

```

def create_model(src_vocab, tgt_vocab, N, d_model,
                 d_ff, h, dropout=0.1):
    attn=MultiHeadedAttention(h, d_model).to(DEVICE)
    ff=PositionwiseFeedForward(d_model, d_ff, dropout).to(DEVICE)
    pos=PositionalEncoding(d_model, dropout).to(DEVICE)
    model = Transformer(
        Encoder(EncoderLayer(d_model,deepcopy(attn),deepcopy(ff),
                             dropout).to(DEVICE),N).to(DEVICE),
        Decoder(DecoderLayer(d_model,deepcopy(attn),
                             deepcopy(attn),deepcopy(ff), dropout).to(DEVICE),
                             N).to(DEVICE),
        nn.Sequential(Embeddings(d_model, src_vocab).to(DEVICE),
                       deepcopy(pos)),
        nn.Sequential(Embeddings(d_model, tgt_vocab).to(DEVICE),
                       deepcopy(pos)),
        Generator(d_model, tgt_vocab)).to(DEVICE)
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)
    return model.to(DEVICE)

```

Creates an encoder by instantiating the `Encoder()` class

Creates a decoder by instantiating the `Decoder()` class

Creates `src_embed` by generating input embeddings for the source language

Creates a generator by instantiating the `Generator()` class

Creates `tgt_embed` by generating input embeddings for the target language



The `create_model()` function uses the `Transformer()` class we defined earlier, with five essential elements: `encoder`, `decoder`, `src_embed`, `tgt_embed`, and `generator`. We construct these five components by instantiating `Encoder()`, `Decoder()`, and `Generator()` classes. To generate the source language embedding, we process numerical representations of German phrases using word embedding and positional encoding, combining the results to form the `src_embed` component. We do the same for the target language to create `tgt_embed`.

Finally, we import the `create_model()` function from the local module and construct a transformer so that we can use it to train the German-to-English translator:

```
from utils.transformer_util import create_model

model = create_model(src_vocab, tgt_vocab, N=6,
                    d_model=256, d_ff=1024, h=8, dropout=0.1)
```

The original 2017 paper by Vaswani et al. [1] uses various combinations of hyperparameters when constructing the model. Here, we choose a model dimension of 256 with eight heads.

## 2.4 Training and using the German-to-English translator

Our German-to-English translator is essentially a multiclass classifier. The objective is to predict the next token in the English vocabulary when translating a German sentence.

In this section, we'll first train the encoder-decoder transformer using batches of German-to-English translations, prepared earlier in this chapter, as our training dataset. After the model is trained, we'll translate German phrases to English.

### 2.4.1 Training the encoder-decoder transformer

The original 2017 paper by Vaswani et al. [1] uses label smoothing during training. *Label smoothing* is a technique used to address overconfidence problems (the predicted probability is greater than the true probability) in classifications. Because it's not directly related to the attention mechanism and transformers, I won't discuss it in detail here. We define the optimizer and the loss function as follows:

```
from utils.transformer_util import (NoamOpt, LabelSmoothing,
                                   SimpleLossCompute)
import torch

optimizer = NoamOpt(256, 1, 2000, torch.optim.Adam(
    model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))
criterion = LabelSmoothing(tgt_vocab,
                          padding_idx=0, smoothing=0.0)
loss_func = SimpleLossCompute(
    model.generator, criterion, optimizer)
```

Next, we'll train the transformer by using the data we prepared earlier in the chapter. We'll train the model for 50 epochs. We'll calculate the loss and the number of tokens

from each batch. After each epoch, we calculate the average loss in the epoch as the ratio between the total loss and the total number of tokens.

### Listing 2.8 Training a transformer to translate German to English

```
for epoch in range(50):
    model.train()
    tloss=0
    tokens=0
    for batch in batches:
        out = model(batch.src, batch.trg,
                    batch.src_mask, batch.trg_mask)
        loss = loss_func(out, batch.trg_y, batch.ntokens)
        tloss += loss
        tokens += batch.ntokens
    print(f"Epoch {epoch}, average loss: {tloss/tokens}")
    torch.save(model.state_dict(), "files/de2en.pth")
```

Predicts the next token using the transformer

Calculates loss and adjusts model parameters

Counts the number of tokens in the batch

Saves the weights in the trained model after training

Once the training is done, the model weights are saved as `de2en.pth` on your computer. Alternatively, you can download the weights from my Google Drive at <https://mng.bz/vZM1>. Unzip the file after downloading.

## 2.4.2 Translating German to English with the trained model

Now that you've trained the transformer, you can use it to translate any German sentence to English. We define a function `de2en()` as shown in the following listing.

### Listing 2.9 Defining a function to translate German to English

```
from utils.transformer_util import subsequent_mask
DEVICE="cuda" if torch.cuda.is_available() else "cpu"
def de2en(ger):
    tokenized_ger= [tok.text for tok in de_tokenizer.tokenizer(ger)]
    tokenized_ger=["BOS"]+tokenized_ger+["EOS"]
    geridx=[de_word_dict.get(i,UNK) for i in tokenized_ger]
    src=torch.tensor(geridx).long().to(DEVICE).unsqueeze(0)
    src_mask=(src!=0).unsqueeze(-2)
    memory=model.encode(src,src_mask)
    start_symbol=en_word_dict["BOS"]
    ys = torch.ones(1, 1).fill_(start_symbol).type_as(src.data)
    translation=[]
    for i in range(100):
        out = model.decode(memory,src_mask,ys,
                        subsequent_mask(ys.size(1)).type_as(src.data))
        prob = model.generator(out[:, -1])
        _, next_word = torch.max(prob, dim=1)
        next_word = next_word.data[0]
        ys = torch.cat([ys, torch.ones(1, 1).type_as(
            src.data).fill_(next_word)], dim=1)
        sym = en_idx_dict[ys[0, -1].item()]
        translation.append(sym)
```

Uses the encoder to convert German to vector representations

Predicts the next English token using the decoder

```

if sym != 'EOS':
    translation.append(sym)
else:
    break
trans=" ".join(translation)
for x in '!?,:;.,'("-!&)%'":
    trans=trans.replace(f" {x}",f"{x}")
return trans

```

← Stops translating when the next token is EOS

← Joins the predicted tokens to form an English sentence as the translation

In the `de2en()` function, we first use the tokenizer to convert the German sentence to tokens. We then add BOS and EOS to the beginning and the end of the sentence, respectively. We use the dictionary `de_word_dict` that we created earlier in the chapter to convert tokens to indices. The encoder produces a tensor that captures the meaning of the German sentence and passes it to the decoder.

Based on the output produced by the encoder, the decoder in the trained model starts translation in an autoregressive manner, starting with the beginning token BOS. In each time step, the decoder generates the most likely next token based on previously generated tokens, until the predicted token is EOS, which signals the end of the sentence.

Now let's try to translate five German sentences in the list `trainde`:

```

model.load_state_dict(torch.load("files/de2en.pth",
    weights_only=True, map_location=DEVICE))
model.eval()
for i in range(5):
    print("original Ger:", trainde[100+i])
    print("original Eng:", trainen[100+i])
    print("translated Eng:", de2en(trainde[100+i]))

```

The output is

```

original Ger: Männliches Kleinkind in einem roten Hut, das sich an
einem Geländer festhält.
original Eng: Toddler boy in a red hat holding on to some railings.
translated Eng: Toddler boy in a red hat holding on to some railings.
original Ger: Drei Hunde stehen auf einer Wiese und eine Person kniet
in der Nähe.
original Eng: Three dogs stand in a grassy field while a person
kneels nearby.
translated Eng: Three dogs stand in a grassy field and one person
near them.
original Ger: Ein Mann steht vor einem Hochhaus.
original Eng: A man is standing in front of a skyscraper
translated Eng: A man is standing in front of a skyscraper
original Ger: Eine Frau fährt ihr Baby in einem Sportwagen im
örtlichen Park spazieren.
original Eng: A woman is walking her baby with a stroller at the
local park.
translated Eng: A woman is walking her baby in a stroller at the
local park.

```

original Ger: Ein Mann in einem roten Hemd sitzt neben Obst, das zu verkaufen ist.  
original Eng: A man in a red shirt is sitting by fruit for sale.  
translated Eng: A man in a red shirt is sitting by fruit for sale.

As you can see, the output compares the original English translation and the translation generated by the trained model. The generated English translations are close to the original translations.

### Exercise 2.5

Use the `de2en()` function to translate the 11th and 12th sentences in `trainde` into English. Compare the translations with the original English translations in `trainen`.

While the transformer architecture is originally designed for NLP tasks, it can be used to tackle other challenges in machine learning. In the next chapter, you'll learn to apply transformer models to computer vision tasks. Specifically, you'll learn to convert an image into a sequence of tokens, similar to how text is transformed into a sequence of words. You'll then create a vision transformer and train it to classify images into different categories.

### Summary

- Transformers are advanced deep-learning models that excel in handling sequence-to-sequence prediction challenges. Their strength lies in effectively understanding the relationships between elements in input and output sequences over long distances.
- The key ingredient of the transformer architecture is its attention mechanism. This mechanism assesses the relationship between words in a sequence by assigning weights, determining how closely words are related based on the training data. This enables transformer models such as ChatGPT to comprehend relationships between words, thus understanding human language more effectively.
- A common way to calculate attention is scaled dot product attention (SDPA). This method is also called self-attention because the algorithm calculates how a word attends to all words in the sequence, including the word itself.
- Furthermore, instead of using one set of query, key, and value vectors, transformer models use a concept called multi-head attention. The query, key, and value vectors are split into multiple heads. Each head pays attention to different parts or aspects of the input, enabling the model to capture a broader range of information and form a more detailed and contextual understanding of the input data.

- Word embeddings are vector representations of tokens used in natural language processing (NLP) tasks. They map tokens into continuous vector spaces, allowing the model to process and understand their meanings numerically.
- Transformers lack inherent sequential understanding, as they process tokens in parallel rather than sequentially. Positional encoding introduces information about the token order into the model.

# *Classify images with a vision transformer*

---

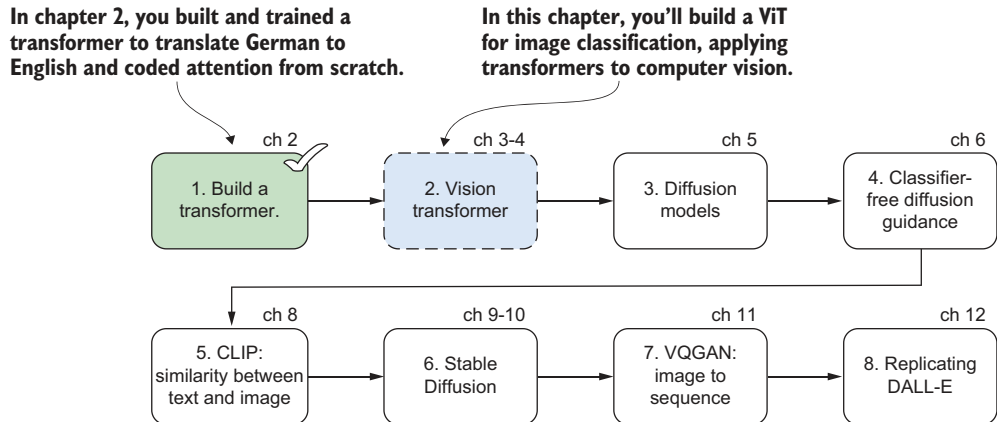
## ***This chapter covers***

- Dividing an image into patches of tokens
- Training a transformer to predict the next image token
- Classifying CIFAR-10 images using a vision transformer (ViT)
- Visualizing how a ViT pays attention to different parts of an image

Building on the ideas from the previous chapter, where we explored how transformers handle sequential data in language, we can now extend this perspective to images. In transformer-based text-to-image generation, a pivotal step is converting an image into a sequence of tokens, much like how we process words in a sentence in natural language. This is where vision transformers (ViTs) come in. ViTs, introduced by Google researchers in their landmark 2020 paper “An Image Is Worth  $16 \times 16$  Words” [1], brought the power of transformer architectures, originally designed for natural language, to the world of computer vision. This innovation allows us to use attention-based mechanisms to connect text and images in a unified framework.

This chapter guides you through the core ideas and practical implementation of ViTs. You'll build a ViT from scratch and train it to classify images from CIFAR-10, a widely used benchmark consisting of 60,000 tiny ( $32 \times 32$ ) color images across 10 object categories such as airplanes, cars, birds, and cats. The process begins by splitting each image into an  $8 \times 8$  grid, resulting in 64 patches. Each patch is treated as a token in the transformer's input sequence. This clever adaptation enables us to bring the strengths of transformers, such as self-attention and long-range dependency modeling, into computer vision tasks.

Figure 3.1 helps you see where this fits in the eight steps of building your own text-to-image models. We'll focus on step 2: mastering how to build a ViT for image classification. You'll apply what you learned about transformers in chapter 2 (for text) to the computer vision domain, discovering how to convert images into sequences and process them with a transformer.



**Figure 3.1** The eight-step road map for building a text-to-image generator from scratch. In this chapter, you'll focus on step 2: building a ViT for image classification. This is your bridge from transformers for text to transformers for images, showing how to represent visual information as sequences of tokens, which is an essential concept for subsequent chapters.

By focusing on image classification, you'll gain hands-on experience with the architecture and training loop of a ViT. Specifically, you'll see how images are divided into patches and encoded as tokens. You'll construct a transformer encoder that can process these tokens, capturing both local and global image features through self-attention. You'll then train your ViT to distinguish between the 10 classes of images in CIFAR-10 and visualize which image regions the model focuses on when making predictions.

Beyond just building a working image classifier, this chapter gives you the intuition and skills to extend transformers to more advanced computer vision and generative

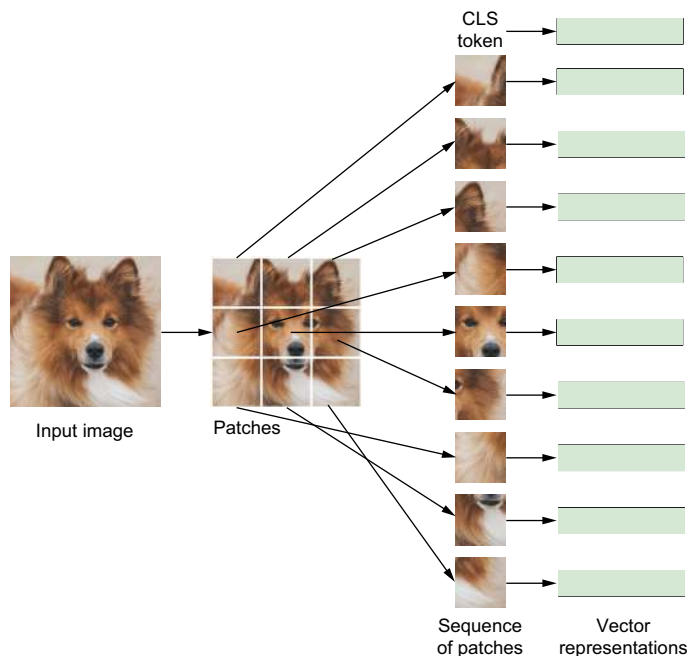
tasks. Understanding how to tokenize images, embed spatial information, and apply self-attention to visual data sets the stage for tasks such as image captioning and text-to-image generation in the chapters ahead. Let's get started on this essential step in your journey: unlocking the power of transformers for images.

### 3.1 *The blueprint to train a ViT*

A ViT applies the transformer architecture to images (first created for handling sequences of text) by turning them into sequences of tokens. This makes it possible to use powerful transformer features, such as self-attention, for computer vision tasks. We'll walk through the main steps for preparing images for a ViT, and then I'll guide you in building and training an encoder-only transformer model for image classification.

#### 3.1.1 *Converting images to sequences*

The first step in constructing a ViT is to convert images into a series of tokens, similar to words in a sentence. Figure 3.2 illustrates the process of converting an image into a sequence. The image is divided into fixed-size patches, which are then ordered sequentially. The sequence starts with the top-left patch, followed by patches to the right in the same row, ending with the bottom-right patch. To perform the image classification task, a special class token is added at the beginning of the sequence to aggregate information in the whole image. Each patch is then transformed into a vector called a token embedding.



**Figure 3.2**  
**Converting an image**  
**into a sequence**



In the example in figure 3.2, the image of a dog is divided into 9 patches for simplicity. These patches form a sequence: the top-left patch is followed by the next patch on the right, continuing row by row until the bottom-right patch. Later in this chapter, we'll work with images from the CIFAR-10 dataset, dividing them into an  $8 \times 8$  grid to yield 64 patches.

For image classification tasks, a class token is added to the beginning of the sequence of image patches. Each token, including the class token and patch tokens, is embedded into a vector through a process known as *image embedding*. For instance, in the CIFAR-10 dataset, each patch is converted into a 48-dimensional vector, capturing its essential features. To summarize, an image in the CIFAR-10 dataset is converted into a tensor of size (65, 48). The number 65 means there are 65 elements in the sequence: one class token (the CLS token) and 64 image patches. Each element is represented by a 48-dimensional vector.

Transformers inherently lack sequence order, as their self-attention mechanism doesn't account for the positions of inputs. To address this, we add *positional encodings* to the token embeddings. These encodings inject information about the relative positions of patches within the image, enabling the model to understand spatial structure. Each positional encoding is also represented by a 48-dimensional vector, forming a tensor of size (65, 48) for the entire sequence.

For the CIFAR-10 dataset, we'll use *learned positional encodings*, which adapt during training, instead of *fixed positional encodings* that we discussed in chapter 2. Images often contain complex and nonuniform spatial relationships. Fixed positional encodings assume a smooth, regular progression, which might not be ideal for capturing the irregularities and varied spatial structures present in visual data.

### Learned positional encoding vs. fixed positional encoding

Fixed positional encodings are predefined, deterministic values added to the input embeddings. These encodings are static and don't change during training. On the other hand, learned positional encodings are model parameters that are updated during training, much like the weights in a neural network. This allows the model to discover the optimal positional encodings for a given task and dataset.

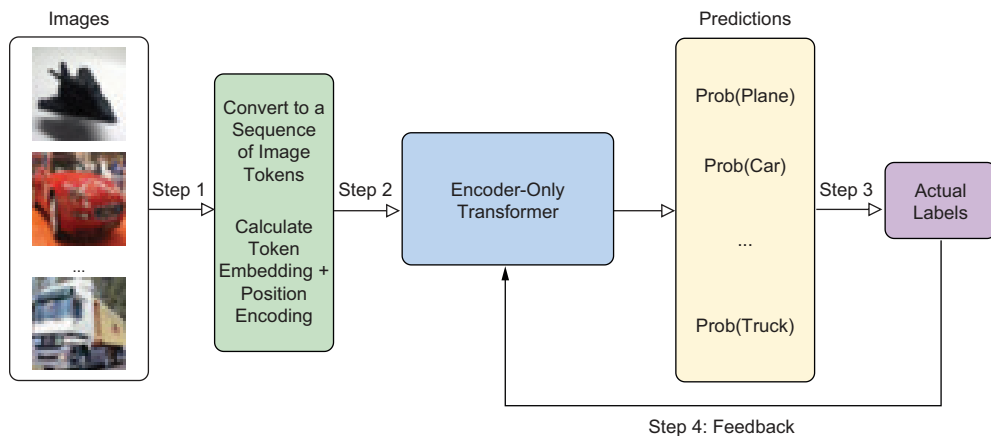
Because fixed positional encodings don't require additional parameters, they reduce the number of learnable parameters in the model, resulting in faster training and lower memory usage. It's also easy to implement and computationally efficient. In addition, fixed positional encoding is extendable: it can be used for sequence lengths beyond those seen in the training data. In contrast, learned position encoding has limited extrapolation capabilities: it struggles with positions not seen in the training data.

Fixed positional encoding may lack flexibility and adaptability, especially for tasks or datasets with unique spatial or sequential characteristics. In contrast, learned positional encoding is adaptable to specific tasks and datasets, which can potentially improve performance.

For images, the *input embedding* is the sum of the image embedding and the positional encoding. Because the image embedding and the positional encoding have the same size, the resulting input embedding also has a size of (65, 48). This combined representation is fed into the transformer model, enabling it to process the image for tasks such as classification.

### 3.1.2 Training a ViT for classification

Our goal in this chapter is to create and train a ViT to classify color images in the CIFAR-10 dataset. Figure 3.3 provides a diagram of the steps involved. We'll use the (image, label) pairs in the CIFAR-10 dataset to train the model. Images are first converted to sequences of image tokens. We then calculate the token embeddings and positional encodings of elements in the sequence. The input embeddings are the sum of token embeddings and positional encodings, and these input embeddings are then fed to an encoder-only transformer. The transformer predicts the label of each image. We then compare the prediction with the actual labels and calculate the cross-entropy loss. We tweak the model parameters to minimize this loss.



**Figure 3.3** The steps required to train a ViT to classify images

First, we'll obtain the training dataset, as shown on the left side of figure 3.3. As we'll explain in detail in the next section, we'll use the CIFAR-10 dataset, which contains 10 different classes of color images such as planes, cars, and trucks. The training data is labeled so we know whether each image is a plane, a car, or something else. The size of each training image is  $3 \times 32 \times 32$ , meaning three color channels, with a height and width of 32 pixels.

We'll follow the steps in the previous subsection to convert each training image into a sequence of image tokens (step 1). Specifically, each image is divided into an  $8 \times 8$

grid, with 64 image patches. Additionally, a special class token is added to the beginning of the patch sequence, which is used to aggregate information for image classification tasks. Therefore, an image is converted into a sequence with 65 elements ( $1 + 8 \times 8 = 65$ ). By processing the image as a sequence, we'll train the transformer model to capture long-range dependencies between different parts of the image, allowing us to apply natural language processing (NLP) techniques to image analysis.

Each image patch in the sequence has a height and width of 4 pixels ( $32 \div 8 = 4$ ). Each image patch is then converted to an image embedding, represented by a 48-value vector. The image embedding captures the image features in each patch.

To maintain spatial information about the relative positions of the patches within the image, we'll add positional encodings to the patch embeddings. We'll use learned positional encodings, which also have a size of (65, 48) for each image. The input embedding is the sum of image embedding and positional encoding.

We'll create a ViT as our classifier, as shown in the center of figure 3.3. Specifically, we'll create an encoder-only transformer by stacking  $N = 4$  identical encoder layers on top of each other. Each encoder layer is composed of two distinct sublayers: a multi-head self-attention layer and a fully connected feed-forward network. (The architecture of an encoder is shown in figure 2.5 of chapter 2, section 2.1.2.) We'll also attach a linear layer at the end to ensure the output of the model has 10 values. These 10 values correspond to the 10 classes of images in the CIFAR-10 dataset.

We'll select a loss function for our multiclass classification problem, and cross-entropy loss is commonly used for this task. Cross-entropy loss measures the difference between the predicted probability distribution and the true distribution of the labels. We'll use the AdamW optimizer (a variant of the gradient descent algorithm, which updates model parameters step-by-step in the direction that reduces the error the most) to update the network's weights during training. We set the learning rate to 0.01. The learning rate controls how much the model's weights are adjusted with respect to the loss gradient during training.

In training, we'll iterate through the training data. During forward passes, we feed the input embeddings to the ViT to obtain predictions (step 2) and compute the loss by comparing the predicted labels with the actual labels (step 3; refer to the right side of figure 3.3). We'll then backpropagate the gradient through the network to update the weights. This is where the learning happens (step 4), as shown earlier at the bottom of figure 3.3. After a fixed number of training epochs, we have a trained ViT. Now that you have a high-level overview of how to train a ViT to classify images, let's dive into the details of the project!

### 3.2 The CIFAR-10 dataset

We'll use the CIFAR-10 dataset to train the ViT in this chapter. CIFAR-10 (CIFAR stands for Canadian Institute for Advanced Research) is a popular dataset for training machine learning, especially computer vision algorithms. The dataset consists of 10 different classes of color images.

In this section, you'll download the dataset, split it into a train and test set, and place them in batches for training later. You'll also visualize some sample images to see what the images in this dataset look like. The Python program in this chapter is adapted from the excellent GitHub repository of Tin Nguyen (<https://mng.bz/4n7D>).

**NOTE** The Python programs for this chapter can be accessed on the book's GitHub repository (<https://github.com/markhliu/txt2img>) and are also available as a Google Colab notebook (<https://mng.bz/5vq7>) for interactive exploration.

### 3.2.1 *Downloading and visualizing CIFAR-10 images*

We'll use the `datasets` package in the `torchvision` library to download the CIFAR-10 dataset directly. We first download the training set as follows:

```
import torchvision

trainset=torchvision.datasets.CIFAR10(root=r'.',
                                     train=True, download=True)
```

Each image in the CIFAR-10 dataset is labeled with a class number ranging from 0 to 9. To convert these numbers into meaningful object names, we define a list as follows:

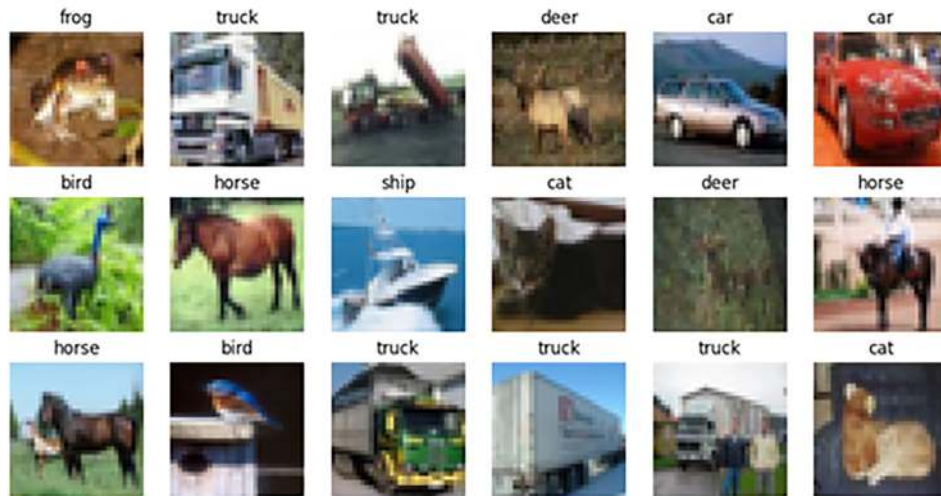
```
names = ['plane', 'car', 'bird', 'cat', 'deer',
         'dog', 'frog', 'horse', 'ship', 'truck']
```

This means an image with a label of 0 is a plane, a label of 1 indicates a car, and so on. Next, we plot the first 18 images in the training set as follows:

```
import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=(12,6),dpi=100)
for i in range(3):
    for j in range(6):
        plt.subplot(3, 6, 6*i+j+1)
        plt.imshow(trainset[6*i+j][0])
        plt.axis('off')
        plt.title(names[trainset[6*i+j][1]], fontsize=12)
plt.subplots_adjust(hspace=0.20)
plt.show()
```

We plot the 18 images in a  $3 \times 6$  grid. The `axis('off')` option in the `matplotlib` library allows us to turn off the axis so we see only the pictures. We place the name of each picture as the title, so that the names are shown on top of each image. For example, the top-left image is a frog, while the bottom-right image is a cat, as shown in figure 3.4.



**Figure 3.4** Sample images from the CIFAR-10 dataset. The dataset contains 10 classes of color images: plane, car, bird, truck, and so on. Each image has a dimension of  $3 \times 32 \times 32$ : three color channels (red, green, blue) with a height and width of 32 pixels. The class label is shown on top of each image.

We also download the test set for out-of-the-sample testing to ensure our trained model isn't overfit. This can be done as follows:

```
testset=torchvision.datasets.CIFAR10(root=r'.',
                                     train=False, download=True)
```

By using the `train=False` argument, we instruct the `torchvision` library to download the test subset instead of the train subset of the CIFAR-10 dataset.

### Exercise 3.1

Plot the first 18 images in the test dataset in a  $3 \times 6$  grid. Use the `axis('off')` option in the `matplotlib` library to turn off the axis so you see only the images, not the x- or y-axis values. Place the name of each picture as the title so that you see a name on top of each image.

### 3.2.2 Preparing datasets for training and testing

The datasets we've downloaded are Python objects. We'll transform them into PyTorch tensors so that we can feed them into the ViT later to train the model. We first transform the training set as follows:

```
import torchvision.transforms as transforms

trainset.transform = transforms.Compose(
```

```
[transforms.ToTensor(),
transforms.Resize((32, 32), antialias=True),
transforms.RandomHorizontalFlip(p=0.5),
transforms.RandomResizedCrop((32, 32), scale=(0.8, 1.0),
                              ratio=(0.75, 1.3333333333333333),
                              interpolation=2, antialias=True),
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

Before transformation, the image data comprises integers ranging from 0 to 255. We convert them to PyTorch float tensors with values between  $-1$  to  $1$ . We also perform a couple of image augmentations such as random horizontal flips and random resize crops. These argumentations increase the variety of images in the training data without extra data, reducing the risk of overfitting. Next, we place the training data in batches for training:

```
import torch

trainloader = torch.utils.data.DataLoader(trainset, batch_size=32,
                                          shuffle=True)
```

We apply similar transformations on the test set and place them in batches as well:

```
testset.transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Resize((32, 32), antialias=True),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
testloader = torch.utils.data.DataLoader(testset, batch_size=32,
                                         shuffle=False)
```

We now have the training and testing datasets ready. In the next section, we'll build a ViT to classify these images.

### 3.3 *Building a ViT from scratch*

To save space, we'll define our ViT in a local module `ViTutil.py`. Download the file `ViTutil.py` from the book's GitHub repository (<https://github.com/markhliu/txt2img>), and place it in the folder `/utils/` on your computer (or clone the repository to your computer). If you're working in Google Colab, you can quickly set up the repository and add it to your working directory by running

```
!git clone https://github.com/markhliu/txt2img
import sys
sys.path.append("/content/txt2img")
```

To convert an image into a sequence, we'll first divide an image (with a size of  $3 \times 32 \times 32$ ) into an  $8 \times 8$  grid, hence 64 different patches. Each patch therefore has a size of  $(3, 4, 4)$  because  $32 \div 8 = 4$ . We'll use a convolutional layer to convert each patch into a

vector with 48 values. These vectors are considered tokens. By doing this, we convert an image into a sequence of 64 tokens.

ViT uses the scaled dot product attention (SDPA) mechanism to capture the relationships among different tokens, which represent different patches in the image. In this section, I'll explain how to code the SDPA attention mechanism and construct a ViT from scratch to classify images.

### 3.3.1 Dividing images into patches

We first define a few hyperparameters that we'll use in our ViT. *Hyperparameters* are parameters that control the learning process in machine learning models. Their values are set before the learning process starts and won't change due to training. In contrast, non-hyperparameters are randomly initialized when we create the model, and their values are adjusted during training. Specifically, we treat all hyperparameters in our model as class variables in the following `Config()` class.

**Listing 3.1** Setting model hyperparameters

```
device = "cuda" if torch.cuda.is_available() else "cpu"

class Config:
    patch_size=4
    hidden_size=48
    num_hidden_layers=4
    num_attention_heads=4
    intermediate_size= 4 * 48
    image_size=32
    num_classes=10
    num_channels=3
config=Config()
```

← Defines the `Config()` class

← Each image patch has a height and width of 4 pixels.

← Each image patch is converted to a 48-value tensor.

← Instantiates the `Config()` class and names it `config`

The class variable `patch_size=4` means each patch of image has a height and width of 4 pixels. Because our training images are  $32 \times 32$  pixels, this means we divided each image into an  $8 \times 8$  grid with 64 patches. The class variable `hidden_size=48` means each patch is converted into a 48-value vector. We'll use multi-head attention to capture different aspects of relationships among tokens in the image, so we set `num_attention_heads=4`. This means we use four attention heads instead of just one when calculating attention. I'll explain other hyperparameters as we go along.

#### Exercise 3.2

You can access the value of the class variables by using the dot notation. For example, `print(config.hidden_size)` leads to an output of 48. What's the output for `print(config.num_classes)`?

To divide an image into 64 patches, we define a `PatchEmbeddings()` class in the local module file `ViTUtil.py` that you just downloaded.

**Listing 3.2** Converting image patches into a sequence

```

from torch import nn
class PatchEmbeddings(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.projection = nn.Conv2d(
            config.num_channels,
            config.hidden_size,
            kernel_size=config.patch_size,
            stride=config.patch_size)
    def forward(self, x):
        x = self.projection(x)
        x = x.flatten(2).transpose(1, 2)
        return x

```

The `PatchEmbeddings()` class takes a batch of images of size  $3 \times 32 \times 32$  and divides each image into an  $8 \times 8$  grid (hence, 64 patches). The class then uses the `Conv2d` layer to convert each image patch into a 48-value vector. We set both the kernel size and stride size to 4, so the kernel slides on each patch just once. This ensures that each image patch is represented by a tensor with a dimension equal to 48 (the number of output channels in the `Conv2d` layer). As a result, a  $3 \times 32 \times 32$  image becomes a tensor of  $48 \times 8 \times 8$  after passing through the `Conv2d` layer. We then rearrange the output so each image has a dimension of  $64 \times 48$ , meaning each of the 64 patches is represented by a 48-value vector. If you need a refresher on how convolutional neural networks operate, see *Deep Learning with PyTorch* by Luca Antiga, et al. (Manning, 2025).

To give you a better understanding of how the `PatchEmbeddings()` class works, we'll create a hypothetical image, pass it through the class, and print out the shape of the output, as follows:

```

from utils.ViTUtil import PatchEmbeddings

patchembed=PatchEmbeddings(config)
img=torch.randn((1,3,32,32))
out=patchembed(img)
print(out.shape)

```

We use `torch.randn()` to generate a tensor with a size of  $(1, 3, 32, 32)$ : there's one image in the batch with a shape of  $3 \times 32 \times 32$ . We then pass the image through the `PatchEmbeddings()` class. The output is as follows:

```
torch.Size([1, 64, 48])
```

The output has a shape of  $(1, 64, 48)$ , meaning there's one image in the batch divided into 64 patches, and each patch is represented by a 48-value vector.



### Exercise 3.3

Use `torch.randn()` to generate a tensor with a size of (5, 3, 32, 32). Pass the tensor through the `PatchEmbeddings()` class. Print out the shape of the output. What's the meaning of each number in the shape of the output?

### 3.3.2 Modeling the positions of different patches in an image

We've transformed each image into 64 patches, effectively converting the images into sequential data. Sequential data refers to datasets where the order of elements is crucial, as the relative positioning of elements often carries critical information necessary for understanding the data. The arrangement of patches within an image is significant because swapping the positions of two patches results in a fundamentally different image. To add the position of each patch within an image to the final output, we define an `Embeddings()` class in the local module.

**Listing 3.3 Adding positional encoding to the image embedding**

```
class Embeddings(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.patch_embeddings = PatchEmbeddings(config)
        self.cls_token = nn.Parameter(torch.randn(1, 1,
                                                    config.hidden_size))

        num_patches = (config.image_size // config.patch_size) ** 2
        self.position_embeddings = \
            nn.Parameter(torch.randn(1, num_patches+1, config.hidden_size))

    def forward(self, x):
        x = self.patch_embeddings(x)
        batch_size, _, _ = x.size()
        cls_tokens = self.cls_token.expand(batch_size, -1, -1)
        x = torch.cat((cls_tokens, x), dim=1)
        x = x + self.position_embeddings
        return x
```

Adds the CLS token to the beginning of the sequence

Uses learned positional encoding, with a dimension of (65, 48)

Passes the image through the PatchEmbeddings() class

Embeds the image, with a dimension of (65, 48)

Adds positional encoding to the image embedding

In the `Embeddings()` class, we first pass an image through the `PatchEmbeddings()` class so that the image is represented by 64 image tokens. We then add the CLS (classification) token to the beginning of the sequence. The CLS token is also a 48-value vector that will be used in the classification layer in our ViT to classify the image. The CLS token isn't associated with any specific part of the image but is used to aggregate information across the entire image for the purpose of classification. This approach allows ViT to use the strength of transformer models for image analysis tasks.

Therefore, the image is now represented by  $64 + 1 = 65$  tokens. To model the positions of these tokens, we create a positional encoding for the sequence, with a size of  $65 \times 48$ , the same as the size of token embeddings of the image. That is, the first position is represented by a 48-value vector, so is the second position, and so on. These 65 positional vectors are learned from the training data. Finally, the positional encoding is added to the token embedding and used as the output of the `Embeddings()` class shown previously in listing 3.3. We again use a hypothetical image and pass it through the `Embeddings()` class to show how positional encoding works:

```
from utils.ViTUtil import Embeddings

embed=Embeddings(config)
img=torch.randn((1,3,32,32))
out=embed(img)
print(out.shape)
print(embed.position_embeddings.shape)
```

Creates a hypothetical image

Passes the image through the class

Prints out the shape of the final output, which is the sum of token embedding and positional encoding

Prints out the shape of the positional encoding only

The output is

```
torch.Size([1, 65, 48])
torch.Size([1, 65, 48])
```

Both the output from the `Embeddings()` class and the positional encoding have a shape of (1, 65, 48).

### 3.3.3 Using the multi-head self-attention mechanism

As we explained in chapter 2, the SDPA uses query, key, and value to calculate the relationships among elements in a sequence. It assigns scores to show how an element in a sequence is related to all other elements. In addition, instead of using one set of query, key, and value vectors, the transformer model uses multi-head attention.

In our example, the embedding dimension is 48 (meaning each token is represented by a 48-value vector), and we'll split the 48-dimensional query, key, and value vectors into four heads. Each head has a set of query, key, and value vectors with dimensions of 12 ( $48 \div 4 = 12$ ). Each head pays attention to different parts or aspects of the input (e.g., texture or edges), enabling the model to capture a broader range of information and form a more detailed and contextual understanding of the image. To implement this, we define the `AttentionHead()` class in the local module `ViTUtil.py` as follows to calculate the attention in each head.

#### Listing 3.4 Calculating self-attention in each attention head

```
class AttentionHead(nn.Module):
    def __init__(self, hidden_size, attention_head_size, bias=True):
        super().__init__()
```

```

self.hidden_size = hidden_size
self.attention_head_size = attention_head_size
self.query = nn.Linear(hidden_size, attention_head_size, bias=bias)
self.key = nn.Linear(hidden_size, attention_head_size, bias=bias)
self.value = nn.Linear(hidden_size, attention_head_size, bias=bias)
def forward(self, x):
    query = self.query(x)
    key = self.key(x)
    value = self.value(x)
    attention_scores = torch.matmul(query, key.transpose(-1, -2))
    attention_scores = attention_scores / math.sqrt(\
        self.attention_head_size)
    attention_probs = nn.functional.softmax(attention_scores,
        dim=-1)
    attention_output = torch.matmul(attention_probs,
        value)
    return (attention_output, attention_probs)

```

**Calculates query, key, and value vectors**

**Calculates attention weights**

**Final attention is the product of attention weights and the value vector.**

The `AttentionHead()` class takes a sequence  $X$  as the input. It passes the input embedding  $X$  through three linear layers to obtain query, key, and value vectors. The corresponding weights for the three linear layers are  $W^Q$ ,  $W^K$ , and  $W^V$ , each having a dimension of  $12 \times 12$ . These weights are learned from data during the training phase. Thus, we calculate query  $Q$ , key  $K$ , and value  $V$  as  $Q = X * W^Q$ ,  $K = X * W^K$ , and  $V = X * W^V$ .

We assess the similarities between the query and key vectors using the SDPA approach. SDPA involves calculating the dot product of the query ( $Q$ ) and key ( $K$ ) vectors. A high dot product indicates a strong similarity between the two vectors, and vice versa. The dot product is scaled by the square root of  $d_k$  (the dimension of the key vector  $K$ , which in our case is 12 in each head) to obtain the scaled attention scores. The next step is to apply the softmax function to these attention scores, converting them into attention weights. This ensures that the total attention a token gives to all tokens in the sequence sums up to 100%. The final attention is then calculated as the dot product of these attention weights and the value vector  $V$ .

After calculating the attention in each head, we use the `MultiHeadAttention()` class defined in the local module `ViTUtil.py` to calculate multi-head attention, similar to what we've done in chapter 2. For brevity, we won't discuss the `MultiHeadAttention()` class in detail, but in brief, the class first takes the input embeddings and creates multiple heads. It then passes the input sequence  $X$  in each head through the `AttentionHead()` class we defined earlier and finally concatenates the attention vectors in the four heads into one large attention vector.

### 3.3.4 Building an encoder-only transformer

To create an encoder-only transformer, we stack  $N = 4$  identical encoder layers together to increase representation capacity and enable hierarchical feature extraction. Each

encoder layer is composed of two distinct sublayers: one is a multi-head self-attention layer, as defined in the `MultiHeadAttention()` class, and the other is a straightforward, position-wise, fully connected feed-forward network, as defined in the `MLP()` class in the local module `ViTUtil.py`:

```
class MLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.dense_1 = nn.Linear(config.hidden_size,
                                  config.intermediate_size)
        self.activation = GELU()
        self.dense_2 = nn.Linear(config.intermediate_size,
                                  config.hidden_size)

    def forward(self, x):
        x = self.dense_1(x)
        x = self.activation(x)
        x = self.dense_2(x)
        return x
```

**The first linear layer, with 48 input channels and  $4 \times 48 = 192$  output channels**

**With GELU activation**

**The second linear layer, with 192 input channels and 48 output channels**

The `MLP()` class is defined with two key parameters: `intermediate_size`, the dimensionality of the feed-forward layer, and `hidden_size`, representing the model's dimension size. Typically, `intermediate_size` is chosen to be four times the size of `hidden_size`. In our example, `hidden_size` is 48, so we set `intermediate_size` to  $4 \times 48 = 192$ . This practice is a standard approach in transformer architectures. It enhances the network's ability to capture and learn complex features in the training dataset.

The input to the `MLP()` class first goes through a linear layer with 48 input channels and 192 output channels. The output then goes through the Gaussian error linear unit (GELU) activation. The most commonly used activation function, rectified linear unit (ReLU), isn't differentiable everywhere. The GELU activation function, in contrast, is differentiable everywhere and provides a better learning process, so we use the GELU activation here. The last layer in the `MLP()` class is a linear layer with 192 input channels and 48 output channels. This way, the output has the same dimension as the input to the `MLP()` class. Next, we create an encoder block using the following `Block()` class in the local module.

### Listing 3.5 Creating an encoder block

```
class Block(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.attention = MultiHeadAttention(config)
        self.layer_norm_1 = nn.LayerNorm(config.hidden_size)
        self.mlp = MLP(config)
        self.layer_norm_2 = nn.LayerNorm(config.hidden_size)

    def forward(self, x, output_attentions=False):
        attention_output, attention_probs = \
            self.attention(self.layer_norm_1(x),
                           output_attentions=output_attentions)
```

**The input first goes through layer normalization and then the multi-head attention layer.**

```

x = x + attention_output
mlp_output = self.mlp(self.layer_norm_2(x))
x = x + mlp_output
if not output_attentions:
    return (x, None)
else:
    return (x, attention_probs)

```

← Applies residual connections

← Applies residual connections again

← Goes through layer normalization again and then the feed-forward network

Each encoder block contains two sublayers: a multi-head self-attention layer and a feed-forward network. Additionally, both of these sublayers incorporate layer normalization and residual connections. A residual connection involves passing the input through a sequence of transformations and then adding the input back to these transformations' output. The method of residual connection is employed to combat the issue of vanishing gradients, which is a common challenge in very deep networks. Layer normalization standardizes the observations in a layer to have a zero mean and a unit standard deviation. We stack four such encoder blocks together to form an encoder-only transformer, as illustrated by the `Encoder()` class in the local module.

### Listing 3.6 Creating an encoder-only transformer

```

class Encoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.blocks = nn.ModuleList([])
        for _ in range(config.num_hidden_layers):
            block = Block(config)
            self.blocks.append(block)
    def forward(self, x, output_attentions=False):
        all_attentions = []
        for block in self.blocks:
            x, attention_probs = block(x,
                                      output_attentions=output_attentions)
            if output_attentions:
                all_attentions.append(attention_probs)
        if not output_attentions:
            return (x, None)
        else:
            return (x, all_attentions)

```

← Stacks four encoder layers together

← The input goes through the four encoder layers sequentially.

← If required, returns attention weights as well

There are four encoder blocks in the encoder-only transformer. The output from the first block is used as the input to the second block. The output from the second block is used as the input to the third block, and so on. The `output_attentions` argument can be set as either `True` or `False`. If it's set as `True`, both the final output and all the intermediate attention weights are returned. Otherwise, only the final output is returned.

### 3.3.5 Using the ViT to create a classifier

Typically, a transformer can be used for various downstream tasks such as text generation, classification, or summarization. In our setting, the goal is to use the ViT to

classify images in the CIFAR-10 dataset. We therefore create a classifier based on the ViT we just built through the `ViTForClassification()` class in the local module.

**Listing 3.7 Building a classifier based on the ViT**

```
class ViTForClassification(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.image_size = config.image_size
        self.hidden_size = config.hidden_size
        self.num_classes = config.num_classes
        self.embedding = Embeddings(config)
        self.encoder = Encoder(config)
        self.classifier = nn.Linear(self.hidden_size,
                                     self.num_classes)

        self.apply(self._init_weights)

    def forward(self, x, output_attentions=False):
        embedding_output = self.embedding(x)
        encoder_output, all_attentions = self.encoder(\
            embedding_output, output_attentions=output_attentions)
        logits = self.classifier(encoder_output[:, 0, :])
        if not output_attentions:
            return (logits, None)
        else:
            return (logits, all_attentions)
```

Attaches a head at the end of the model to create a classifier with 10 output classes

Input images first go through the `Embedding()` class to generate input embeddings.

The input embeddings go through the ViT to create encoder output.

The encoder output goes through the classifier head to obtain the logits corresponding to the 10 classes of labels.

The `ViTForClassification()` class takes a batch of images as input and outputs the logits over the 10 classes of objects. Later, we'll apply the softmax function on these logits to obtain the probability distributions of the images over the 10 classes.

To achieve this, the `ViTForClassification()` class first passes the images through the `Embeddings()` class defined earlier to obtain the input embedding, which is the sum of token embedding and positional encoding. The input embeddings are then passed through the encoder-only ViT. The output from the ViT is passed through a classifier head, with 10 output neurons, corresponding to the 10 classes of labels.

Instead of using all outputs from the ViT to classify images, we use only the output associated with the CLS token. As explained earlier, the CLS token isn't associated with any specific part of the image but is used to aggregate information across the entire image for the purpose of classification.

### 3.4 Training and using the ViT to classify images

Our constructed classifier can be trained to predict which class an image belongs to in the CIFAR-10 dataset. In this section, we'll detail the process of selecting an appropriate loss function and optimizer. We'll train the ViT using batches of data prepared

earlier in this chapter. After the model is trained, we'll use it to classify images in the test set (i.e., unseen images).

### 3.4.1 Choosing the optimizer and the loss function

We first instantiate the `ViTForClassification()` class from the local module to create an image classifier, as follows:

```
from utils.ViTUtil import ViTForClassification

model = ViTForClassification(config).to(device)
```

We'll use the AdamW optimizer, a variant of the Adam optimizer that has been widely used in machine learning. The AdamW optimizer, first proposed by Ilya Loshchilov and Frank Hutter, decouples weight decay (you can learn more about the AdamW optimizer in the original paper by Loshchilov and Hutter [2]). Instead of applying weight decay directly to the gradients, AdamW applies weight decay directly to the parameters (weights) after the optimization step. This modification helps achieve better generalization performance by preventing the decay rate from being adapted along with the learning rates:

```
from torch import nn, optim

optimizer = optim.AdamW(model.parameters(),
                        lr=0.01, weight_decay=1e-2)
```

Because this is a multiclass classification problem, we'll use the categorical cross-entropy loss function, as follows:

```
loss_fn = nn.CrossEntropyLoss()
```

We'll use PyTorch's automatic mixed-precision package `torch.amp` to speed up training. The default data type in PyTorch tensors is `float32`, a 32-bit floating-point number, which takes up twice as much memory as a 16-bit floating number, `float16`. Operations on the former are slower than those on the latter. There is a tradeoff between precision and computational costs. Which data type to use depends on the task at hand. The package `torch.amp` provides an automatic mixed precision, where some operations use `float32` and others `float16`. Mixed precision tries to match each operation to its appropriate data type to speed up training. We therefore instantiate a `GradScaler()` class as follows:

```
scaler = torch.amp.GradScaler()
```

The use of PyTorch's automatic mixed-precision package may reduce the training time.

### 3.4.2 Training the ViT for image classification

When we build and train a deep neural network, there are many hyperparameters that we can choose (e.g., the learning rate and the number of epochs to train). These hyperparameters affect the performance of the model. To find the best hyperparameters, we can use a test set to gauge the performance of the model with different hyperparameters.

To illustrate, we'll use the test dataset to help determine the optimal number of training epochs. It's important to make this decision using data that the model hasn't seen during training, rather than relying solely on the training set. If we use only the training set to choose how many epochs to train for, we risk overfitting: the model may learn to perform very well on the training data but fail to generalize to new, unseen examples. This happens because training for too many epochs can cause the model to memorize patterns specific to the training data, rather than capturing generalizable features. By monitoring performance on the test set, we can identify the point where the model achieves its best out-of-sample performance, helping to avoid overfitting. For this purpose, we define an `EarlyStop()` class and create an instance of the class as follows.

**Listing 3.8** Defining an `EarlyStop()` class

```
class EarlyStop:
    def __init__(self, patience=3):
        self.patience = patience
        self.steps = 0
        self.min_loss = float('inf')
    def stop(self, loss):
        if loss < self.min_loss:
            self.min_loss = loss
            self.steps = 0
            to_save = True
        elif loss >= self.min_loss:
            self.steps += 1
            to_save = False
        if self.steps >= self.patience:
            to_stop = True
        else:
            to_stop = False
        return to_save, to_stop
stopper=EarlyStop()
```

← If the loss in the test set is smaller than the previously achieved minimum, keep training.

← If the loss in the test set is larger than the previously achieved minimum, add 1 to the counter.

← If the performance doesn't improve in three consecutive epochs, stop training.

← Instantiates the `EarlyStop()` class

The `EarlyStop()` class keeps track of the performance of the model, as measured by the loss in the test set. If the loss is smaller than the previously achieved minimum, we keep training the model. If the loss is larger than the previously achieved minimum, the model performance didn't improve in this epoch. We therefore add 1 to the counter `steps`. If the model performance didn't improve for three consecutive epochs, we stop the training loop. Note that you can determine when to stop training by adjusting the `patience` parameter. We set the value of `patience` to 3 in the preceding



example. If you set patience to, say, 10, then you'll stop training only if the model fails to improve in 10 consecutive epochs.

We also define a `train_batch()` function to train the model with a batch of the training data:

```
def train_batch(batch):
    batch = [t.to(device) for t in batch]
    images, labels = batch
    with torch.amp.autocast(device):
        loss = loss_fn(model(images)[0], labels)
        optimizer.zero_grad()
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
    return loss.item()*len(images)/len(
        trainloader.dataset)
```

← Moves training data to CUDA-enabled GPU, if available

← Trains the model with the batch of training data

← Returns the loss in the batch

The function first moves the batch of training data to the CUDA-enabled GPU, if available. It then trains the model using PyTorch's automatic mixed-precision package. Finally, the function returns the training loss in the batch. Note that we've scaled loss by the ratio of the number of images in the batch and the total number of images in the training set. The scaling ensures that later, when we report training loss, it measures the loss in each image, on average.

Next, we train the model for a maximum of 100 epochs. We'll use the `EarlyStop()` class to stop training if the model stops improving in the test set for three consecutive epochs. Further, the best model parameters are saved if an epoch achieves the best performance in the test set.

### Listing 3.9 Training the ViT classifier

```
from tqdm import tqdm
import os
os.makedirs("files", exist_ok=True)
for i in range(100):
    print(f'Epoch {i+1}')
    model.train()
    trainL, testL = 0, 0
    for batch in tqdm(trainloader):
        loss=train_batch(batch)
        trainL+=loss
    model.eval()
    with torch.no_grad():
        for batch in testloader:
            batch = [t.to(device) for t in batch]
            images, labels = batch
            logits, _ = model(images)
            loss = loss_fn(logits, labels)
            testL+=loss.item()*len(images)/len(testloader.dataset)
    print(f'Train and test losses: {trainL:.4f}, {testL:.4f}')
    to_save, to_stop = stopper.stop(testL)
```

← Trains the model for one epoch

← Tests the model in the test set

```

if to_save==True:
    torch.save(model.state_dict(),"files/ViT.pth")
if to_stop==True:
    break

```

← If the model performance improves in this epoch, save the model parameters.

← If the model fails to improve in three consecutive epochs, stop training.

The training stopped after 13 epochs because the model performance didn't improve after 10 epochs. The best model parameters, which are those after 10 epochs of training, are saved in the file `ViT.pth` on your computer in the `/files/` folder. Alternatively, you can download the file from the book's GitHub repository (<https://github.com/markhliu/txt2img>).

### 3.4.3 Classifying images using the trained ViT

To use the trained ViT to classify images, we obtain a batch of 32 images from the test dataset. We make predictions on the images and obtain attention maps for each image.

**Listing 3.10** Using the trained ViT to classify images

```

import math
import torch.nn.functional as F

model.load_state_dict(torch.load('files/ViT.pth'))
model.eval()
with torch.no_grad():
    batch=next(iter(testloader))
    batch = [t.to(device) for t in batch]
    images, labels = batch
    logits, attention_maps = model(images,
        output_attentions=True)
    predictions = torch.argmax(logits, dim=1)
print(predictions)
print([names[i] for i in predictions.tolist()])

```

← Obtains a batch of 32 images from the test dataset

← Obtains logits and attention maps using the trained model

← Makes predictions based on the highest logits

← Prints out the names of the predicted labels

We first load the model with the trained weights. Next, we extract a batch of 32 test images and pass them through the model to obtain both the predicted logits and the attention maps. Logits are the raw, unnormalized outputs produced for each class before being transformed into probabilities via the softmax function. We then make a prediction for each image based on the highest logit value, resulting in a list called `predictions` that contains 32 predicted numerical labels (each ranging from 0 to 9). The corresponding label names are printed out as follows:

```

tensor([8, 8, 8, 0, 6, 6, 1, 6, 5, 1, 0, 9, 6, 7, 9, 8, 5, 7, 8, 6,
        7, 2, 0, 1, 4, 4, 4, 6, 1, 6, 6, 4], device='cuda:0')
['ship', 'ship', 'ship', 'plane', 'frog', 'frog', 'car', 'frog',
 'dog', 'car', 'plane', 'truck', 'frog', 'horse', 'truck', 'ship',
 'dog', 'horse', 'ship', 'frog', 'horse', 'bird', 'plane', 'car',
 'deer', 'deer', 'deer', 'frog', 'car', 'frog', 'frog', 'deer']

```

The output shows that the first three numerical predictions are 8, which corresponds to the word label 'ship'. The fourth numerical prediction is 0, indicating 'plane', and so on. We'll evaluate the overall accuracy of these predictions later.

The `attention_maps` list contains four tensors, each representing the attention map from a different encoder layer. Let's explore these attention maps here:

```
for attn in attention_maps:
    print(attn.shape)
block0_image0_head0=attention_maps[0][0,0,:,:]
print(block0_image0_head0.shape)
probs_sum=torch.sum(block0_image0_head0,dim=1)
print(probs_sum)
```

The shape of the attention map in each encoder layer

The attention map for the first attention head in the first image in the first encoder layer

Adds up the value of attention across columns

We begin by printing the shape of the attention map for each encoder layer. Next, we focus on the attention map from the first attention head of the first image in the first encoder layer, referred to as `block0_image0_head0`, which has a shape of  $65 \times 65$ . Finally, we sum the values in each column and print the results, as shown here:

```
torch.Size([32, 4, 65, 65])
torch.Size([32, 4, 65, 65])
torch.Size([32, 4, 65, 65])
torch.Size([32, 4, 65, 65])
torch.Size([65, 65])
tensor([[1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000], device='cuda:0')
```

The results indicate that the attention map for each encoder layer has a shape of  $[32, 4, 65, 65]$ . This means that there are 32 images in the batch, and for each image, the attention map is generated for four different attention heads. Each head produces a  $65 \times 65$  probability matrix, where each image is divided into 65 tokens, the first being the CLS token and the remaining 64 corresponding to an  $8 \times 8$  grid of patches. This matrix shows the amount of attention each token allocates to every token in the sequence, and each column sums to 1, representing a complete (100%) distribution of attention. Next, we visualize the attention maps for each image.

### Listing 3.11 Visualizing ViT predictions and attention maps

```
with torch.no_grad():
    attention_maps = torch.cat(attention_maps, dim=1)
```

Concatenates the attention maps from the four encoder layers into one

```

print(f"attention map shape: {attention_maps.shape}")
attention_maps = attention_maps[:, :, 0, 1:]
print(f"attention map shape: {attention_maps.shape}")
attention_maps = attention_maps.mean(dim=1)
print(f"attention map shape: {attention_maps.shape}")
num_patches = attention_maps.size(-1)
size = int(math.sqrt(num_patches))
attention_maps = attention_maps.view(
    -1, size, size)
print(f"attention map shape: {attention_maps.shape}")
attention_maps = attention_maps.unsqueeze(1)
attention_maps = F.interpolate(attention_maps, size=(32, 32),
                              mode='bilinear', align_corners=False)
attention_maps = attention_maps.squeeze(1)
print(f"attention map shape: {attention_maps.shape}")

fig = plt.figure(figsize=(8, 8), dpi=100)
for i in range(16):
    ax = fig.add_subplot(4, 4, i+1, xticks=[], yticks=[])
    ax.imshow(attention_maps[i].cpu(), alpha=0.5, cmap='jet')
plt.tight_layout()
plt.show()

```

**Focuses on the attention between the CLP token and the 64 image tokens**

**Takes the average and resizes the attention map into an  $8 \times 8$  image**

**Upscales the attention map to a  $32 \times 32$  image to match the original image size**

**Plots the attention map for each of the first 16 images in the batch**

The output from the preceding code listing is

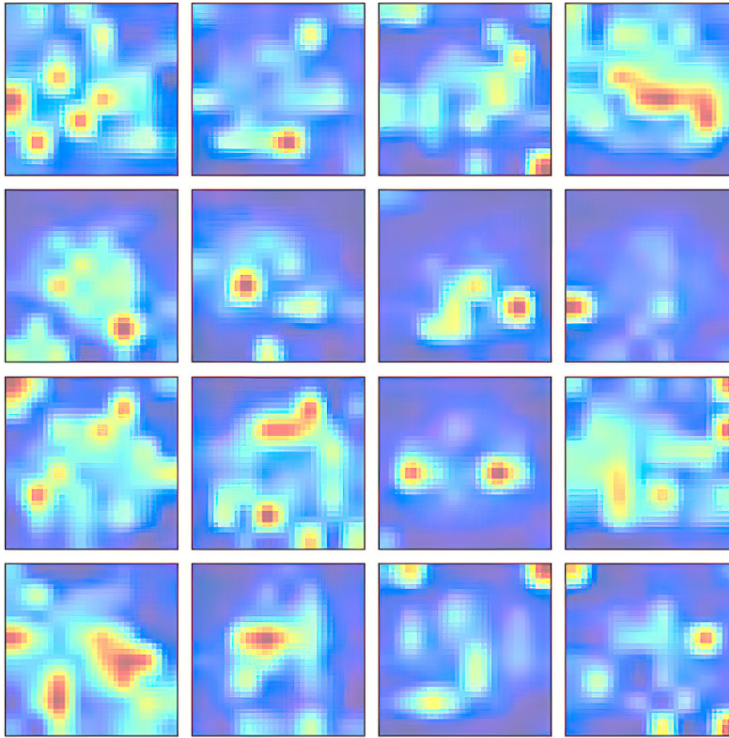
```

attention map shape: torch.Size([32, 16, 65, 65])
attention map shape: torch.Size([32, 16, 64])
attention map shape: torch.Size([32, 64])
attention map shape: torch.Size([32, 8, 8])
attention map shape: torch.Size([32, 32, 32])

```

We start by concatenating the attention maps from all four encoder layers into a single tensor with a shape of  $[32, 16, 65, 65]$ . This shape indicates that there are 32 images in the batch and 16 attention maps per image (four heads from each of the four layers), with each attention probability matrix being  $65 \times 65$ . Next, we extract the attention values corresponding to the interaction between the CLS token and the 64 image patches from the  $8 \times 8$  grid. This results in a tensor of shape  $[32, 16, 64]$ , where each of the 64 values represents the attention the CLS token pays to each image patch. We then average the 16 values to reduce the size of the attention tensor to  $[32, 64]$ . For each image in the batch, we reshape the 64 attention values into an  $8 \times 8$  image and upscale it to a  $32 \times 32$  image to match the original image's size. The attention maps of the first 16 images are shown in figure 3.5.

Figure 3.5 highlights certain regions in the image, but it can be difficult to tell exactly which areas of the image are emphasized. To clarify this, we compare the attention map with the original image by displaying them side by side. Additionally, we overlay both the actual and predicted labels on the images, as shown in the following listing.



**Figure 3.5** The attention maps for 16 test images. For each image, we first extract and concatenate the attention maps from the four encoder layers into one tensor. Then, we isolate the maps representing the attention between the CLS token and the 64 image patches. Averaging the attention maps across the four heads and four encoder layers yields a tensor of 64 values per image. These values are reshaped into an  $8 \times 8$  grid and then upscaled to  $32 \times 32$  to match the original image size. Finally, the attention maps for the 16 images are displayed in a  $4 \times 4$  grid.

### Listing 3.12 Comparing attention maps with original images

```
fig = plt.figure(figsize=(8, 5), dpi=200)
mask = np.concatenate([np.ones((32, 32)), np.zeros((32, 32))],
                      axis=1)
for i in range(16):
    ax = fig.add_subplot(4, 4, i+1, xticks=[], yticks=[])
    img = np.concatenate((images[i].cpu(), images[i].cpu()),
                        axis=-1)
    ax.imshow(img.transpose(1, 2, 0)/2+0.5)
    extended_attention_map = np.concatenate((np.zeros((32, 32)),
        attention_maps[i].cpu()),
        axis=1)
    extended_attention_map = np.ma.masked_where(mask==1,
        extended_attention_map)
    ax.imshow(extended_attention_map, alpha=0.5, cmap='jet')
    gt = names[labels[i]]
```

Shows the original image

Adds the attention map to the right of the original image

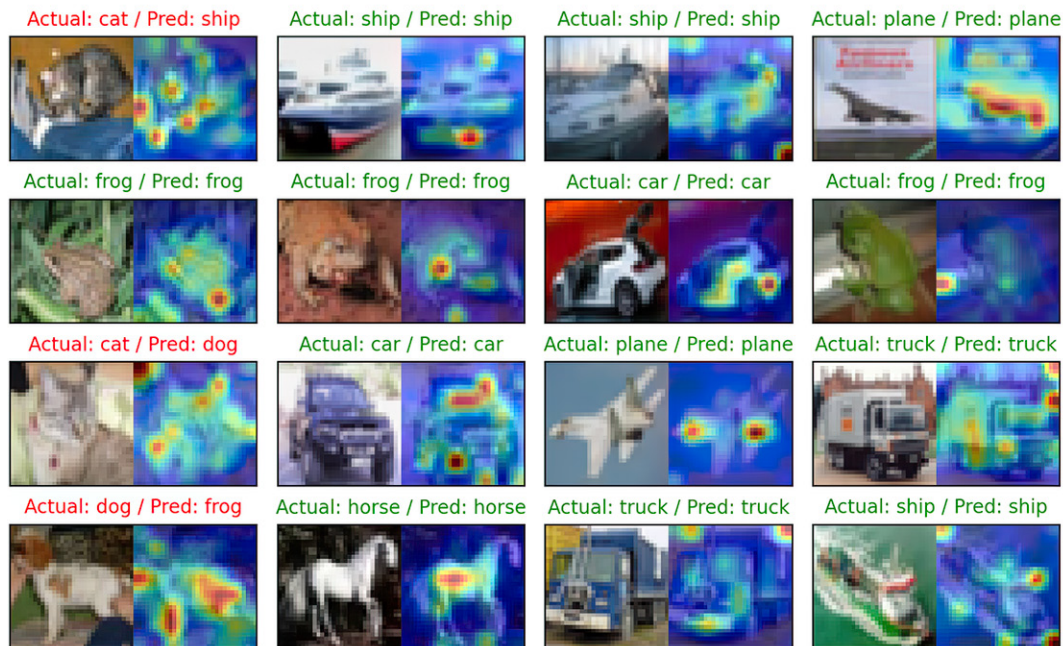
```

pred = names[predictions[i]]
ax.set_title(f"Actual: {gt} / Pred: {pred}",
            color=("green" if gt==pred else "red"),
            fontsize=10)
plt.tight_layout()
plt.show()

```

← Places both the original and the predicted labels on top of the image

After running the preceding code block, you'll see the image shown in figure 3.6. In each image pair, the left side displays the original test image, while the right side shows the corresponding attention map generated by the trained ViT model during prediction. The attention map is overlaid on the original image as its background, clearly indicating which areas are emphasized. For instance, the top-right image highlights the main body of an airplane, and the model correctly classifies it. Similarly, the second image in the bottom row focuses on the torso of a horse, resulting in a correct prediction.



**Figure 3.6** Make predictions on CIFAR-10 images using the trained ViT classifier. In each pair, the left image is the original image from CIFAR-10, and the right image is the attention map generated by the trained ViT. The actual label and the predicted label are printed above each pair of images.

Each subplot displays the actual and predicted labels as titles, green if the prediction is correct and red otherwise. Overall, the trained ViT accurately classified 13 out of the 16 images.



Next, we calculate the overall prediction accuracy of the trained ViT in the test dataset:

```
model.eval()
acc = 0
with torch.no_grad():
    for batch in testloader:
        batch = [t.to(device) for t in batch]
        images, labels = batch
        logits, _ = model(images)
        predictions = torch.argmax(logits, dim=1)
        acc += torch.sum(predictions == \
            labels).item() / len(testloader.dataset)
print(f'the prediction accuracy is {acc:.4f}')
```

← Iterates through all batches in the test dataset

← Makes predictions using the trained ViT classifier

← Prints out the overall accuracy of the predictions

The output is as follows:

```
the prediction accuracy is 0.5621
```

The accuracy is 56.21%. Given that the 10 classes of images are evenly distributed in the test dataset, a random guess will have an accuracy of about 10%. Therefore, the model's prediction accuracy of 56.21% indicates that the model does capture the defining features in each class of images and performs well.

### To improve the prediction accuracy of the ViT classifier

Our goal here is to create and train a ViT classifier from scratch. Achieving the best prediction accuracy isn't the main goal. As a result, we set the patience parameter to 3 in the `EarlyStop()` class to reduce training time.

If desired, you can use the command `stopper=EarlyStop(patience=10)` to set the value of the patience parameter to 10 so that the training stops when the model stops improving in 10 consecutive epochs. Doing so will improve the prediction accuracy.

In the next chapter, you'll build on the skills you learned in this chapter to add captions to images. Specifically, you'll create an encoder–decoder transformer. The encoder is a ViT similar to the one you built in this chapter, which is used to understand visual features in images. The decoder is similar to the one in the German-to-English translator you built in chapter 2. You'll learn how to generate a caption one word at a time, based on the output from the encoder and previous words in the caption.

## Summary

- In transformer-based text-to-image generation, a crucial step is transforming an image into a token sequence, much like converting text into a sequence of words.
- A ViT processes an image by dividing it into fixed-size patches and treating each patch as a token. These patches are then ordered sequentially. The sequence

starts with the top-left patch, followed by the patch to the right in the same row, ending with the bottom-right patch. To facilitate tasks such as image classification, a special class token `CLS` is added at the beginning of the sequence.

- Image tokens, including the class token at the beginning of the sequence, are embedded into a vector through image embedding. Transformers and their self-attention mechanism don't account for the order of inputs. To address this, we add positional encodings to the token embeddings. These encodings inject information about the relative positions of patches within the image, enabling the model to understand spatial structure.
- Fixed positional encodings are predefined, deterministic values added to the input embeddings. These encodings are static and don't change during training. On the other hand, learned positional encodings are model parameters that are updated during training, much like the weights in a neural network. This allows the model to discover the optimal positional encodings for a given task and dataset.
- A trained ViT can be used to classify images. The attention mechanism in ViT can also show the attention the trained model pays to various parts of an image.



# *Add captions to images*



## ***This chapter covers***

- Similarities between image-to-text generation and text-to-image generation
- Building a transformer from scratch to add captions to images
- Training an image-to-text transformer with image-caption pairs
- Adding captions to images with the trained image-to-text transformer

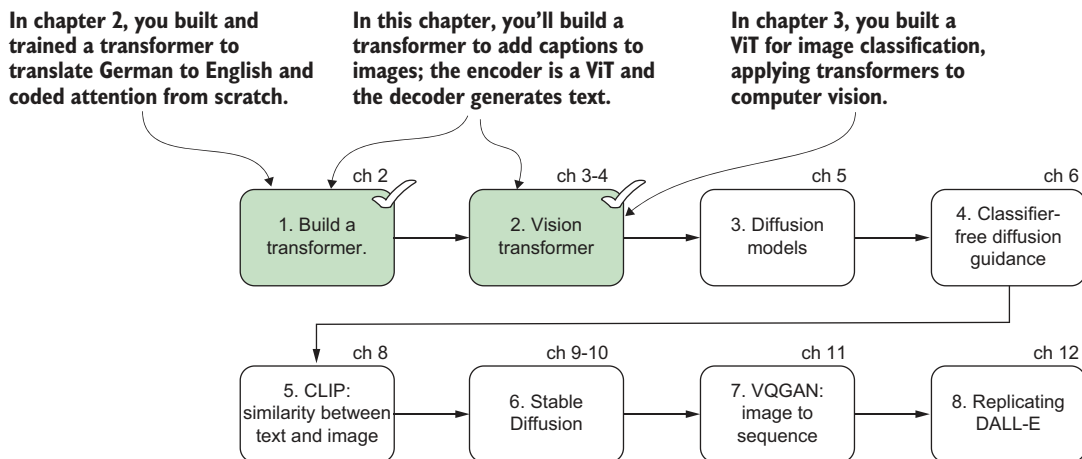
In the previous chapter, we took our first steps in connecting vision and language by learning how models can align these two very different modalities. Now we'll build on that foundation. While our ultimate goal is text-to-image generation, we'll first master the reverse process, image-to-text captioning, because both directions rely on the same underlying principle of learning deep cross-modal relationships.

Training a model to add captions to images is conceptually and practically more accessible than generating a new image from a text prompt. Captioning images forces the model to understand and encode visual features and then map them

coherently to linguistic tokens. This not only builds intuition for handling multimodal data but also lays the groundwork for more ambitious tasks, such as producing realistic images from text prompts, which require the model to reverse the process with added complexity.

In this chapter, you'll build an image-captioning transformer from scratch. This hands-on exercise will not only deepen your understanding of transformer architectures but also demonstrate, step-by-step, how visual and textual information can be integrated into a single model. You'll train the model on a dataset of image-caption pairs, learning how to encode images using a vision transformer (ViT) and decode that information into English captions.

To clarify the big picture, figure 4.1 illustrates how this chapter fits in the eight-step road map for building a text-to-image generator from scratch: it reinforces and applies the skills in steps 1 and 2. By the end of this chapter, you'll be able to encode an image into latent features and use a transformer decoder to translate those features into natural language captions, a skill that is directly transferrable to building text-to-image generators in later chapters.



**Figure 4.1** Eight steps to build a text-to-image generator from scratch. In this chapter, you'll reinforce and apply skills from steps 1 and 2: building a transformer that captions images. The encoder is a ViT, similar to what you created for image classification in chapter 3. The decoder generates text based on image features, much like the decoder you developed for German-to-English translation in chapter 2.

Both image-to-text and text-to-image models share a common architecture: the encoder-decoder transformer. For multimodal tasks, the encoder processes the source data (an image, in this chapter), converting it into a compact embedding. The decoder then generates the target output (a caption), one token at a time, based on the image embedding and previously generated tokens.

In this chapter, the encoder is a ViT. As explained in chapter 3, the ViT breaks an image into a sequence of patches, extracts feature representations from each, and then encodes the entire image as a sequence of patch embeddings.

The decoder is a text generator, much like the one you built for translating German to English in chapter 2. Here, it takes the image embedding and generates an English sentence (caption) that describes the image, token by token, starting from a special <start> token and finishing with an <end> token.

This encoder–decoder structure is foundational for all advanced multimodal generative models. By mastering this structure in the context of image captioning, you’ll be well prepared to handle the more complex challenge of text-to-image generation, which will build directly on the ideas and code you develop in this chapter.

Training a model to add captions to images teaches the model how to see and describe the world, an ability that’s critical when you later ask the model to draw the world from a textual description. Many core techniques, such as patch-based image encoding, cross-modal attention, and autoregressive sequence generation, are shared by both tasks.

In addition, generating realistic images from text (the subject of future chapters) is a far more demanding challenge. The model must not only understand the semantic meaning but also create visually coherent, high-resolution images. By starting with image-to-text, you gain a deep understanding of transformer-based architectures and multimodal learning, getting ready for text-to-image generation in later chapters.

## 4.1 Training and using a transformer to add captions

To train a transformer to add captions to images, we’ll start with data preparation. Specifically, we’ll use a dataset that contains images paired with descriptive captions as our training data. A crucial step in data preparation is to design text input sequences and output sequences so that the transformer learns to predict the next token.

We’ll create an encoder–decoder transformer for this task. The encoder takes the images and encodes them into image embeddings. The decoder takes the output from the encoder and the input sequence from the caption to predict the output sequence. Once the transformer is trained, we’ll feed an image to the trained model and ask it to generate a token after the starting token <start>. We’ll then append the predicted token to <start> and feed the two tokens to the model to predict the next token. We repeat the process until the predicted token is <end>, which signals the end of the captioning process.

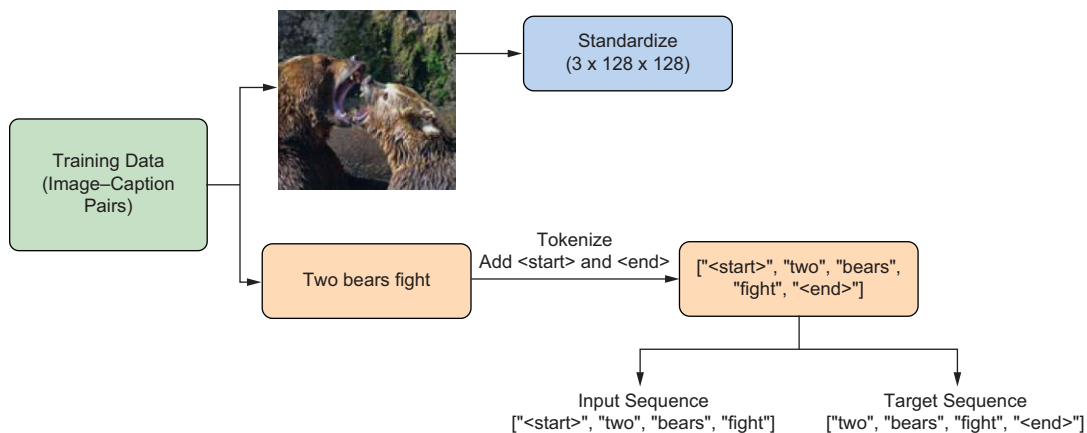
This section provides you with a high-level overview of how to prepare the training dataset so that the model learns to generate captions based on image–caption pairs. We’ll also go over the architecture of the multimodal transformer and the training steps involved.

### 4.1.1 Preparing data and the causal attention mask

To train a transformer to add captions to images, we need a training dataset of images paired with corresponding captions. We’ll use the Flickr 8k Dataset, which consists of about 8,000 images and five captions for each image.

We'll standardize the images to a certain size. In our case, all images will be resized to  $3 \times 128 \times 128$ , meaning three color channels, with a width and height of 128 pixels. We'll tokenize text captions into sequences of tokens. Further, we'll add `<start>` and `<end>` to the beginning and end of each caption so that the model knows when the caption starts and ends.

To force the model to learn to predict the next token based on all tokens before it, but not tokens after it, we'll create input and target sequences. To make the explanation more relatable, figure 4.2 uses an image–caption pair to illustrate how data preparation is performed and how input and target sequences are generated. The image at the top center of figure 4.2 is a picture of two bears fighting. The caption for the image is “two bears fight.”

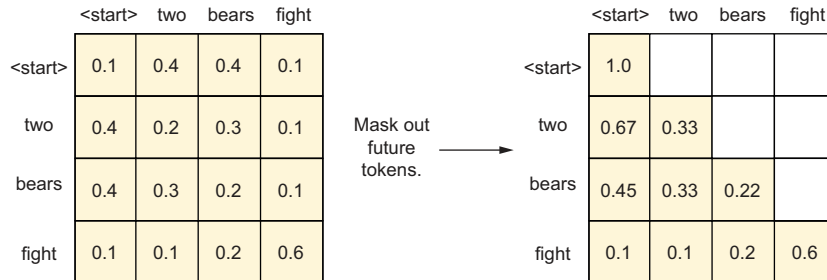


**Figure 4.2** How to prepare data to train a transformer to add captions to images. We first obtain a training dataset of images paired with descriptive captions. The images are standardized to a certain size (e.g.,  $3 \times 128 \times 128$ ) so that we can feed them to the image encoder in the transformer. The captions are tokenized, and we add `<start>` and `<end>` to the beginning and end of the caption. We omit the `<end>` token and use the rest of the tokens as the input sequence (`[<start>, two, bears, fight]`). We slide the input sequence one token to the right and use the result as the target sequence (`[two, bears, fight, <end>]`).

The input sequence for the caption in figure 4.2 is `[<start>, two, bears, fight]`, while the target sequence is `[two, bears, fight, <end>]`. Each token in the target sequence is the next token in the corresponding position in the input sequence. The input sequence is the input to the decoder in the transformer. The target sequence serves as the ground truth. We compare the output from the decoder with the target sequence and calculate the cross-entropy loss. During training, we modify model parameters to minimize this loss.

Our goal is to predict the next token based on all tokens before it, but not future tokens in the sequence (similar to how we generated English translations of German phrases in chapter 2). To hide future tokens during training, we'll introduce the causal

attention mask in the transformer. While we'll explain the transformer structure in detail in the next subsection, we'll explain how the causal attention mask works here, as illustrated in figure 4.3.



**Figure 4.3** How the causal attention mask works

We use the causal attention mask to restrict the model to predict the next token using only tokens before it, not future tokens. This is different from the standard self-attention where each token in a sequence pays attention to all elements in the sequence.

When calculating the standard self-attention weights, each token pays attention to all elements in the sequence, as shown on the left side of figure 4.3. For example, the first token, <start>, pays attention to all four tokens in the sequence [<start>, two, bears, fight]. To calculate masked self-attention, we apply a causal attention mask to the standard self-attention weights. The causal attention mask hides future tokens in the sequence (i.e., attention weights above the main diagonal) so that each token pays attention to only the token itself and all tokens before it. As shown on the right side of figure 4.3, the token <start> pays attention only to itself. This forces the model to predict the token two using the <start> token only in the first iteration. In the second iteration, the model predicts the token bears using only the tokens <start> two, but not future tokens bears fight. In the last iteration, it uses tokens <start> two bears fight to predict the token <end>.

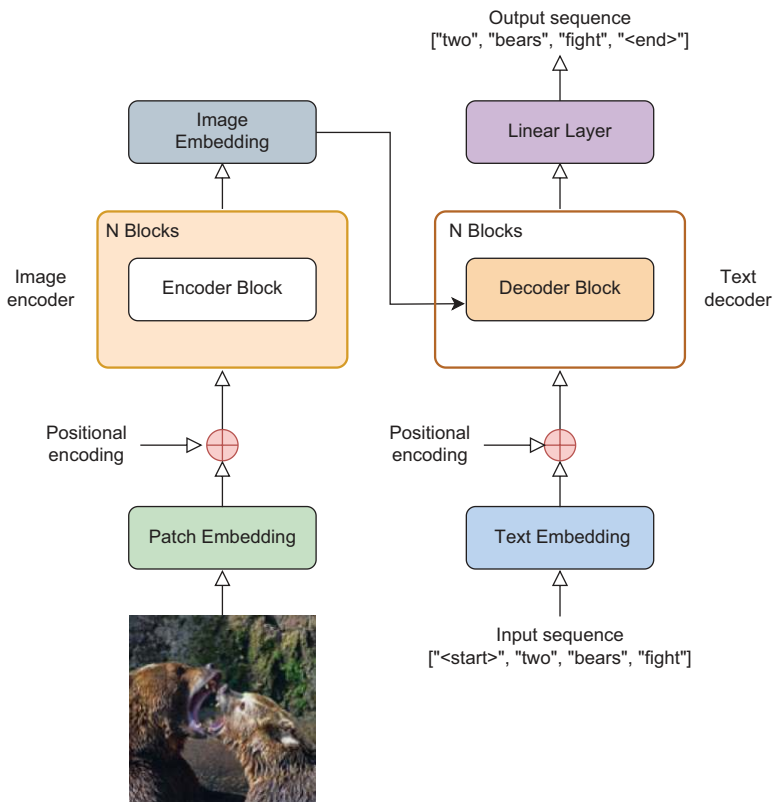
As we mentioned in chapter 2, it's important to note that the decoder operates as a sequence-to-sequence predictor, meaning it outputs a sequence rather than a single token. In the second iteration of the preceding example, the decoder receives <start> two as its input, which consists of two tokens. Consequently, it produces an output sequence with two tokens: the first token predicts what should follow <start>, and the second predicts what should follow two. However, because we already know that two follows <start>, we only need to focus on the token after two.

Now let's go back to data preparation. Because neural networks can take only numerical values as inputs, we'll create a dictionary to map tokens to indices (i.e., integers). The input and target sequences are converted to sequences of indices using this dictionary. Further, to improve the efficiency of training, we'll feed batches of training data to

the model instead of one data point at a time. We'll include 128 input sequences in each batch. Because input sequences have different lengths, we'll pad shorter sequences in each batch with 0s at the end (after the <end> token) so that all sequences in a batch have the same length after padding. Finally, we'll organize the dataset into image–caption pairs and put them in batches for training.

#### 4.1.2 Creating and training a transformer

We'll create an encoder–decoder transformer so that we can train it to add captions to images. Figure 4.4 shows the structure of the model. The encoder in the transformer (left side of the diagram), which consists of  $N$  identical encoder layers, encodes an image into image embeddings that capture the features of the image. It then passes this tensor to the decoder (right side of the diagram), which consists of  $N$  identical decoder layers. The decoder takes the input sequence as the input ([<start>, two, bears, fight]) and predicts the output sequence. We compare the predictions with the ground truth, which is [two, bears, fight, <end>]. We modify the model parameters to minimize the cross-entropy loss. Because a causal attention mask is applied in



**Figure 4.4** The structure of an encoder–decoder transformer to add captions to images

each decoder layer, this is equivalent to training the model to predict the next token based on all tokens before it (but not tokens after it) in the sequence. Note that the target sequence isn't fed to the model; instead, the target sequence is only used to calculate the cross-entropy loss.

The encoder in the transformer first divides the image into a sequence of image patches. It then uses image embedding to extract the features in the image patches. Transformers process input data in parallel, which enhances their efficiency but doesn't inherently allow them to recognize the sequence order of the input. To address this, we add positional encodings to the image embeddings. We'll use fixed positional encoding as we did in chapter 2. The transformer's encoder is made up of six identical layers in this chapter. Each of these layers comprises two sublayers. The first sublayer is a multi-head self-attention layer, and the second sublayer is a feed-forward network.

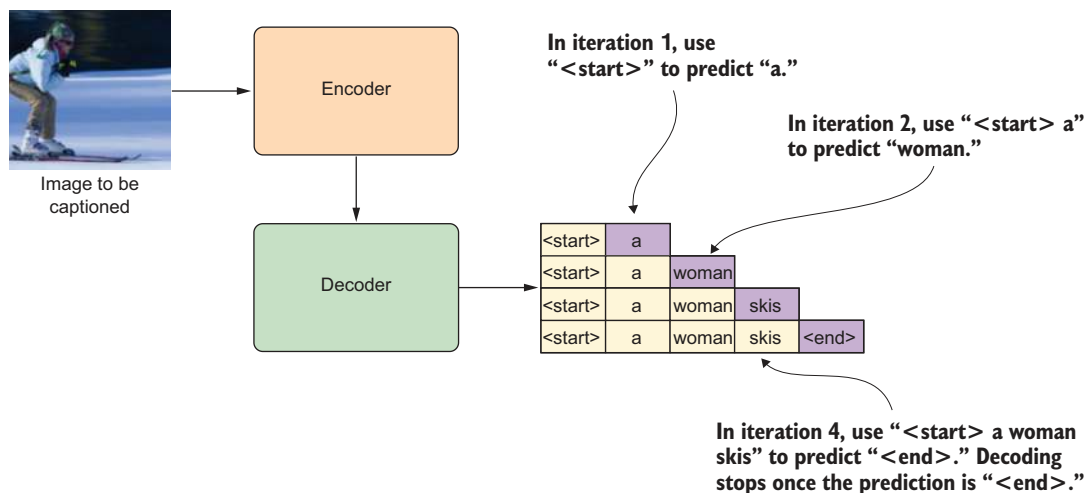
The decoder of the transformer model, as shown on the right in figure 4.4, has six identical decoder layers. Each of these decoder layers contains three sublayers: (1) a multi-head causal self-attention sublayer, (2) a sublayer that performs multi-head cross-attention between the output from the first sublayer and the encoder's output, and (3) a feed-forward sublayer. Note that the input to each sublayer is the output from the previous sublayer. You can refer to chapter 2 on how cross-attention is calculated. The formula to calculate cross-attention is as follows:

$$\text{CrossAttention}(Q, K, V) = \text{softmax}\left(\frac{Q * K^T}{\sqrt{d_k}}\right) * V \quad (4.1)$$

$Q$  is the query vector obtained by passing the input sequence from the caption through a linear layer.  $K$  and  $V$  are the key and value vectors after passing the output from the encoder through two linear layers. In the formula,  $d_k$  represents the dimension of the key vector  $K$ . Here, we're calculating the cross-attention between the text input (represented by  $Q$ ) and the image embedding (represented by  $K$  and  $V$ ).

Once the model is trained, we can use it to add captions to images. Figure 4.5 shows how the captioning process works. The top-left corner of the figure shows an image of a woman skiing. To add a caption to the image, we first feed the image to the encoder in the trained transformer to generate image embeddings. The image embeddings capture the feature maps in the image. We then pass the encoder's output to the decoder. The decoder constructs the caption in an autoregressive fashion. In the context of transformers, autoregressive refers to a sequential approach to generating outputs, where the model generates each token in the sequence one step at a time, conditioned on the previously generated tokens.

In chapter 2, you saw how the decoder generates the English translation to a German phrase autoregressively. In this chapter, the decoder starts with the beginning of the caption token, <start>, and predicts the next token in the caption based on the image embeddings generated by the encoder. It adds the predicted token to the caption and



**Figure 4.5** Adding a caption to an image using the trained transformer

uses the updated caption to predict the next token. The process continues until the predicted token is <end>, which signals the end of the caption.

In figure 4.5, if the first predicted token is a, we append it to the caption and use <start> a to predict the next token. We repeat the process to generate the token woman and then the token skis. In the last iteration, the predicted token is <end>, and we stop the captioning process. Therefore, the trained transformer generates a caption of “a woman skis” for the image shown in figure 4.5.

## 4.2 Preparing the training dataset

You’ll use the Flickr 8k Dataset to train the image-to-text transformer in this chapter. We chose Flickr 8k to reduce the computational resources needed for training the model. The Flickr 8K Dataset is a collection of more than 8,000 photographs sourced from the Flickr website, each annotated with five different descriptive captions provided by human annotators. This dataset is specifically designed for developing and evaluating image-captioning models. The diverse set of images typically depicts people or animals performing various activities in a wide range of settings. The annotations are intended to facilitate the training of machine learning models that can generate descriptive text for images automatically. Other datasets for this purpose include the Flickr 30k Dataset, which has about 30,000 training images, and the MS COCO (Microsoft Common Objects in Context) dataset, which consists of 328,000 images.

In this section, you’ll download the dataset, split the images into a train set and test set, and place them in batches for training later. The Python programs in this chapter are adapted from two great GitHub repositories: one by Senad Kurtisi (<https://mng.bz/yNeo>) and one by Luke Ditria ([https://github.com/LukeDitria/pytorch\\_tutorials](https://github.com/LukeDitria/pytorch_tutorials)).



**NOTE** The Python programs for all chapters in this book can be accessed from the GitHub repository (<https://github.com/markhliu/txt2img>). Ideally, you'll want a CUDA-enabled GPU to run the programs in this chapter. If you don't have one, you can follow along using the Google Colab notebook at <https://mng.bz/MwqD>.

### 4.2.1 Downloading and visualizing Flickr 8k images

You can download the Flickr 8k images and the associated text captions from the Kaggle website ([www.kaggle.com/datasets/adityajn105/flickr8k](http://www.kaggle.com/datasets/adityajn105/flickr8k)). Log in to your Kaggle account, download the zip file, and extract the data from the zip file. Place everything in the `/files/` folder on your computer.

Because we need to split data into train and test, we'll also download a zip file from Andrej Karpathy's website (<https://mng.bz/Qw5R>). Unzip the file, and place the file `dataset_flickr8k.json` in the `/files/` folder on your computer.

We first read the JavaScript Object Notation (JSON) file, which contains information about the captions and corresponding tokens for each caption:

```
import json

with open('files/dataset_flickr8k.json', 'r') as fb:
    data = json.load(fb)
```

All the Flickr 8k images should be stored in the `/files/Images/` folder on your computer. We'll split them into train and test subsets.

#### Listing 4.1 Splitting data into train and test subsets

```
from collections import Counter

train_image_paths = []
train_image_captions = []
test_image_paths = []
test_image_captions = []
word_freq = Counter()

max_len=50
for img in data['images']:
    captions = []
    for c in img['sentences']:
        word_freq.update(c['tokens'])
        if len(c['tokens']) <= max_len:
            captions.append(c['tokens'])
    if len(captions) == 0:
        continue
    path = "files/Images/"+img['filename']
    if img['split'] in {'train', 'val', 'restval'}:
        train_image_paths.append(path)
        train_image_captions.append(captions)
```

← Gathers information about text tokens to create a vocabulary

← Gathers all captions

← Creates a train subset

```
elif img['split'] in {'test'}:
    test_image_paths.append(path)
    test_image_captions.append(captions)
```

← Creates a test subset

The list `train_image_paths` contains the image paths of the train set, while `train_image_captions` contains the corresponding captions. Note that the captions have already been divided into tokens, so we don't need to tokenize them from scratch. Alternatively, you can change `c['tokens']` to `c['raw']` in the preceding code cell, collect the captions, and tokenize them yourself. The counter object `word_freq` counts how often each token is used, and we'll use the information to build a vocabulary later.

Next, we make sure that the number of images matches the number of groups of captions in both the training and test sets (recall that each image has five captions). We also count the number of observations in the two subsets:

```
assert len(train_image_paths)==len(train_image_captions)
assert len(test_image_paths)==len(test_image_captions)
print(f"there are {len(train_image_paths)} training images")
print(f"there are {len(test_image_paths)} test images")
```

The output shows that we have 7,000 training images and 1,000 test images:

```
there are 7000 training images
there are 1000 test images
```

Next, we create a vocabulary for the English language for text-generation purposes.

#### 4.2.2 Building a vocabulary of tokens

Based on the counter object `word_freq`, we'll create a vocabulary of English tokens:

```
min_word_freq=0
words = [w for w in word_freq.keys() if word_freq[w]>min_word_freq]
word2idx = {k:v + 4 for v,k in enumerate(words)}
word2idx['<pad>'] = 0
word2idx['<start>'] = 1
word2idx['<end>'] = 2
word2idx['<unk>'] = 3
```

← Creates a dictionary to map tokens to indices

Hardcodes in the special tokens

The dictionary `word2idx` maps English tokens to indices. We manually insert four special tokens. The padding token `<pad>` (with an index value of 0) pads the captions in a batch to the same length so that we can feed them to the transformer for training later. The token `<start>` (index value 1) signals the start of a caption, and `<end>` (index value 2) signals the end of a caption. Finally, the token `<unk>` (index value 3) is used in case a token isn't found in the dictionary `word2idx`. We can use the dictionary `word2idx` to convert captions to a sequence of integers:

```
indexes=[word2idx.get(token,3) for
          token in test_image_captions[0][0]]
print(indexes)
```

The output is shown here, where tokens in the first caption for the first image in the test set are converted to a sequence of numbers:

```
[12, 18, 318, 11, 12, 13, 11, 45, 30, 4, 234]
```

### Exercise 4.1

Convert the tokens in the third caption for the second image in the test set into a sequence of integers. Name the sequence `indexes2`. Print out the values in `indexes2`.

We create another dictionary, `idx2word`, to map integers indices to tokens. Later, we'll use `idx2word` to convert predicted integers back to tokens and then English sentences in the form of a caption:

```
idx2word={v:k for k, v in word2idx.items()}
tokens=[idx2word.get(idx,"<unk>") for
        idx in indexes]
print(tokens)
print(f"there are {len(idx2word)} unique tokens")
```

Creates a dictionary `idx2word` to map indices to tokens

Converts the sequence of indices to tokens using the dictionary

The output is shown here, where dictionary `idx2word` translates a sequence of integers back to tokens:

```
['the', 'dogs', 'are', 'in', 'the', 'snow', 'in', 'front', 'of', 'a', 'fence']
there are 8387 unique tokens
```

Therefore, the first caption for the first image in the test set must be “the dogs are in the snow in front of a fence.” The English vocabulary we used to create captions has a total of 8,387 unique tokens.

### Exercise 4.2

Use the dictionary `idx2word` to convert the sequence of integers, `indexes2`, that you generated in exercise 4.1 into a sequence of tokens.

## 4.2.3 Preparing the training dataset

To streamline our code, we'll place most helper functions and classes in a local module, `caption_util`. Download the file `caption_util.py` from the book's GitHub repository (<https://github.com/markhliu/txt2img>), and place it in the `/utils/` folder on your computer (or simply clone the repository to your computer). If you're working in

Google Colab, you can quickly set up the repository and add it to your working directory by running

```
!git clone https://github.com/markhliu/txt2img
import sys
sys.path.append("/content/txt2img")
```

Open the file `caption_util.py`, and look at the functions and classes it contains. In particular, we define a `FlickrD()` class in the file to generate image tensors, input tokens, output tokens, and the attention mask. The train and test subsets are generated as follows:

```
from utils.caption_util import FlickrD
trainset=FlickrD(train_image_paths,
                 train_image_captions,word2idx)
testset=FlickrD(test_image_paths,
                 test_image_captions,word2idx)
```

← Imports the `FlickrD()` class from the local module

← Generates the train subset using the `FlickrD()` class

← Generates the test subset using the `FlickrD()` class

We then use the PyTorch `DataLoader()` class to generate data iterators in the train and test subsets:

```
from torch.utils.data import DataLoader
train_loader = DataLoader(trainset,
                          batch_size=128,
                          shuffle=True)
test_loader = DataLoader(testset,
                         batch_size=128,
                         shuffle=True)
```

← Uses the PyTorch `DataLoader()` class to generate a data iterator in the train subset

← Generates a data iterator in the test subset

The generated train and test data loaders can iterate over data during training and evaluation. The `DataLoader()` class provides an efficient way to batch, shuffle, and load data. For example, we can retrieve a batch of testing images and their corresponding input indices, output indices, and attention masks. We save them in a file called `tests.pt` for later use:

```
test_images,test_tokens,\
    test_targets,test_mask=next(iter(test_loader))
import torch
torch.save((test_images,test_tokens),"files/tests.pt")
```

The `.pt` extension designates the file format for PyTorch tensors, enabling you to save tensors directly to files on your computer.

With this, we're done preparing the training dataset. In the next section, we'll create a multimodal transformer to add captions to images.

### 4.3 Creating a multimodal transformer to add captions

Now let's create a multimodal image-to-text transformer. To streamline the code, we'll define the functions and classes related to the construction of the model in the `caption_util.py` file that you downloaded earlier.

#### 4.3.1 Defining a ViT as the image encoder

We'll construct a ViT as the encoder in our model. The purpose of the ViT is to compress an image into an abstract vector representation in the latent space before passing it to the decoder. First, we define the following `extract_patches()` function in the local module to convert an image to patches.

**Listing 4.2 Converting images to patches**

```
def extract_patches(image_tensor, patch_size=8):
    bs, c, h, w = image_tensor.size()
    unfold = torch.nn.Unfold(kernel_size=patch_size,
                             stride=patch_size)
    unfolded = unfold(image_tensor)
    unfolded = unfolded.transpose(1, 2).reshape(bs,
                                                -1, c * patch_size * patch_size)
    return unfolded
```

Obtains the dimensions of the image tensor

Defines the unfold layer

Applies the unfold layer on the image tensor

Reshapes the unfolded tensor to the desired shape

The `extract_patches()` function takes an image and applies an unfold layer to it. Each input image has a shape of (3, 128, 128); the width and height of the image is 128 pixels. Each patch is 8 × 8 pixels tall, so each image is divided into a 16 × 16 grid (128 ÷ 8 = 16). Therefore, the image is divided into 16 × 16 = 256 patches. Each patch in each color channel has 8 × 8 = 64 pixels. Because there are three color channels, the total number of pixels in each patch is 3 × 64 = 192.

Note that the PyTorch `Unfold(kernel_size, stride)` layer extracts sliding local blocks (or patches) from an input tensor. The `kernel_size` argument is the size of the block to extract (an 8 × 8 patch in our example). The `stride` argument is the step size for moving the window across the input, which is 8 in our example.

Given an input of shape (N, C, H, W), the output will be a tensor of shape (N, C × kernel\_height × kernel\_width, L), where L is the total number of sliding blocks. If we apply the `Unfold()` layer on one image with an input shape of (1, 3, 128, 128), the output has a shape of (1, 192, 256): one image in the batch, each image patch has 3 × 8 × 8 = 192 pixels, and there are a total of 256 sliding blocks.

You can verify the results by passing an image through the `extract_patches()` function and printing out the shape of the output:

```
from utils.caption_util import extract_patches
image=test_images[0].unsqueeze(0)
```

Selects an image from the test subset

```
patches=extract_patches(image,patch_size=8)
print(patches.shape)
```

← Applies the `extract_patches()` function on the image

← Prints out the dimensions of the output

The output is

```
torch.Size([1, 256, 192])
```

We pass the first image in the test set through the `extract_patches()` function, and the output has a shape of (1, 256, 192). The unfold layer converts each image tensor into a tensor of (192, 256), where 256 represents the number of patches in each image, and 192 is the number of pixels in each patch. The `extract_patches()` function reshapes the output from the unfold layer as (1, 256, 192); there's one image in the batch. The image is converted to a sequence of 256 image patches. Each image patch is represented by a 192-value vector.

The ViT treats an image as a sequence of patches. These patches have orders. To model the orders of patches in a sequence, we define the following `SinusoidalPosEmb()` class in the local module. Here, we use fixed positional encoding as we did in chapter 2 (you can take a look at the definition in the file `caption_util.py` if interested).

Like other transformers, we use multi-head attention when building the encoder layers. We, therefore, define the `AttentionBlock()` class in the local module.

#### Listing 4.3 Defining the multi-head attention block

```
from torch import nn
class AttentionBlock(nn.Module):
    def __init__(self, hidden_size=128, num_heads=4, masking=True):
        super(AttentionBlock, self).__init__()
        self.masking = masking
        self.multihead_attn=nn.MultiheadAttention(hidden_size,
            num_heads=num_heads,
            batch_first=True,
            dropout=0.0)
    def forward(self, x_in, kv_in, key_mask=None):
        if self.masking:
            bs, l, h = x_in.shape
            mask = torch.triu(torch.ones(l, l,
                device=x_in.device), 1).bool()
        else:
            mask = None
        return self.multihead_attn(x_in, kv_in, kv_in,
            attn_mask=mask, key_padding_mask=key_mask)[0]
```

← Uses the off-the-shelf **MultiheadAttention()** class in PyTorch

← Creates a mask if calculating masked multi-head attention

← Doesn't apply the mask otherwise

The `AttentionBlock()` class will be used several times later. We'll use it to calculate unmasked self-attention when creating the encoder, and we'll set the `masking` argument to `False`. We'll use it to calculate masked self-attention when creating the decoder, and

we'll set the masking argument to True. We'll also use it to calculate cross-attention between the encoder's output and the input in the decoder, and we'll set the masking argument to False then as well.

Next, we define a generic TransformerBlock() class in the local module to create a transformer block, as shown in the following listing. This block can be used as either an encoder or a decoder.

#### Listing 4.4 Defining a generic transformer block

```
class TransformerBlock(nn.Module):
    def __init__(self, hidden_size=128, num_heads=4,
                  decoder=False, masking=True):
        super(TransformerBlock, self).__init__()
        self.decoder = decoder
        self.norm1 = nn.LayerNorm(hidden_size)
        self.attn1 = AttentionBlock(hidden_size=hidden_size,
                                    num_heads=num_heads,
                                    masking=masking)

        if self.decoder:
            self.norm2 = nn.LayerNorm(hidden_size)
            self.attn2 = AttentionBlock(hidden_size=hidden_size,
                                        num_heads=num_heads, masking=False)
        self.norm_mlp = nn.LayerNorm(hidden_size)
        self.mlp = nn.Sequential(nn.Linear(hidden_size, hidden_size * 4),
                                  nn.ELU(),
                                  nn.Linear(hidden_size * 4, hidden_size))

    def forward(self, x, input_key_mask=None,
                cross_key_mask=None, kv_cross=None):
        x = self.attn1(x, x, key_mask=input_key_mask) + x
        x = self.norm1(x)
        if self.decoder:
            x = self.attn2(x, kv_cross, key_mask=cross_key_mask) + x
            x = self.norm2(x)
        x = self.mlp(x) + x
        return self.norm_mlp(x)
```

**Calculates self-attention**

**If it's a decoder layer, calculates cross-attention**

**The output goes through the feed-forward network.**

Both the encoder and decoder of the multimodal image-to-text transformer consist of multiple blocks. The TransformerBlock() class we defined previously serves as the foundation for creating both encoder and decoder blocks. To construct the encoder, six encoder blocks are stacked sequentially. Similarly, the decoder is formed by stacking six decoder blocks. An encoder block processes its input through two main components: a self-attention layer and a feed-forward network. In contrast, a decoder block includes three sublayers: (1) a causal self-attention layer, (2) a cross-attention layer that computes the interaction between the encoder's output and the previous output of the decoder, and (3) a feed-forward network. Now we can go ahead and create an encoder in the local module.

Listing 4.5 Creating a vision encoder

```

class VisionEncoder(nn.Module):
    def __init__(self, image_size, channels_in,
                  patch_size=16, hidden_size=128,
                  num_layers=6, num_heads=4):
        super(VisionEncoder, self).__init__()
        self.patch_size = patch_size
        self.fc_in = nn.Linear(channels_in*patch_size*patch_size,
                                hidden_size)
        seq_length = (image_size // patch_size) ** 2
        self.pos_embedding=nn.Parameter(torch.empty(1,seq_length,
                                                       hidden_size).normal_(std=0.02))
        self.blocks = nn.ModuleList([
            TransformerBlock(hidden_size, num_heads,
                              decoder=False, masking=False) for _ in range(num_layers)
        ])
    def forward(self, image):
        bs = image.shape[0]
        patch_seq = extract_patches(image,
                                     patch_size=self.patch_size)
        patch_emb = self.fc_in(patch_seq)
        # Add a unique embedding to each token embedding
        embs = patch_emb + self.pos_embedding
        # Pass the embeddings through each transformer block
        for block in self.blocks:
            embs = block(embs)
        return embs

```

The encoder consists of multiple encoder blocks.

First, convert images into sequences of patches.

Adds positional encodings to image embeddings

The input embeddings go through multiple encoder blocks.

When a batch of images passes through the encoder, we first use the `extract_patches()` function we defined earlier to convert the images to sequences of image patches. The sequences then go through a linear layer to be converted to image embeddings. The positional encodings of these patches are added to the image embeddings. The sum (which is called input embeddings) then goes through multiple encoder blocks.

### 4.3.2 Creating the decoder to generate text

We define the following `Decoder()` class in the local module. The goal of the decoder is to generate a caption one text token at a time, similar to the decoder in the German-to-English translator we built in chapter 2.

Listing 4.6 Creating a decoder to generate text

```

class Decoder(nn.Module):
    def __init__(self, num_emb, hidden_size=128,
                  num_layers=6, num_heads=4):
        super(Decoder, self).__init__()
        self.embedding = nn.Embedding(num_emb, hidden_size)
        self.embedding.weight.data=0.001*self.embedding.weight.data
        self.pos_emb = SinusoidalPosEmb(hidden_size)
        self.blocks = nn.ModuleList([

```



```

        TransformerBlock(hidden_size, num_heads,
                          decoder=True) for _ in range(num_layers)
    ])
    self.fc_out = nn.Linear(hidden_size, num_emb)
def forward(self, input_seq, encoder_output,
            input_padding_mask=None,
            encoder_padding_mask=None):
    input_embs = self.embedding(input_seq)
    bs, l, h = input_embs.shape
    seq_indx = torch.arange(l, device=input_seq.device)
    pos_emb = self.pos_emb(seq_indx).reshape(1,
                                             l, h).expand(bs, l, h)
    embs = input_embs + pos_emb
    for block in self.blocks:
        embs = block(embs,
                     input_key_mask=input_padding_mask,
                     cross_key_mask=encoder_padding_mask,
                     kv_cross=encoder_output)
    return self.fc_out(embs)

```

**Passes the input sequence through the embedding layer**

**Adds positional encoding to the word embedding**

**The input embedding goes through multiple decoder layers.**

**The output from the decoder layers goes through a linear layer, with the number of outputs equal to the English vocabulary.**

The `TransformerBlock()` class we defined earlier serves as a building block of the `Decoder()` class. During training, the captions are first tokenized and converted to sequences of indices. These sequences go through word embedding and positional encoding, similar to what we've done for the English translation of German phrases in chapter 2. The sum of word embedding and positional encoding is the input embedding to the decoder blocks. Each decoder block has three sublayers: a multi-head masked self-attention layer, a multi-head cross-attention layer, and a feed-forward network.

The masked self-attention layer incorporates a mask to ensure the model predicts the next token in the caption based solely on the preceding tokens, without considering future tokens. The cross-attention layer computes attention between the decoder's current state and the image embeddings provided by the encoder. This mechanism ensures that the generated caption aligns with the content of the image.

Finally, we combine the vision encoder with the text decoder to create the image-to-text transformer. This is reflected in the `VisionEncoderDecoder()` class defined in the local module.

#### Listing 4.7 The encoder–decoder transformer to add captions

```

class VisionEncoderDecoder(nn.Module):
    def __init__(self, image_size, channels_in,
                  num_emb, patch_size=16,
                  hidden_size=128, num_layers=(6,6),
                  num_heads=4):
        super(VisionEncoderDecoder, self).__init__()
        self.encoder = VisionEncoder(

```

```

        image_size=image_size, channels_in=channels_in,
        patch_size=patch_size, hidden_size=hidden_size,
        num_layers=num_layers[0], num_heads=num_heads)
    self.decoder = Decoder(num_emb=num_emb,
        hidden_size=hidden_size,
        num_layers=num_layers[1], num_heads=num_heads)
    def forward(self, input_image, target_seq, padding_mask):
        bool_padding_mask = padding_mask == 0
        encoded_seq = self.encoder(image=input_image)
        decoded_seq = self.decoder(input_seq=target_seq,
            encoder_output=encoded_seq,
            input_padding_mask=bool_padding_mask)
        return decoded_seq

```

Generates padding masks for the text in the captions to ensure all sequences in the same batch have the same length

Encodes the images and passes image embeddings to the decoder

Predicts the output sequence

The `VisionEncoderDecoder()` class takes two inputs in the forward pass: an image and an input sequence. The image is passed through the image encoder to produce the image embedding. The input sequence is the `<start>` token plus tokens in the caption associated with the image, as we explained earlier in this chapter. The decoder tries to predict the next token after each token in the input sequence. For example, if the caption associated with the image is “a dog runs,” we’ll add `<start>` and `<end>` to the beginning and the end of the caption. We’ll use `<start>` a dog runs as the input sequence. We move the input sequence one token to the right and use it as the target sequence (i.e., ground truth). During training, we feed the input sequence `<start>` a dog runs to the decoder to predict the output sequence. We then compare the output sequence with the target sequence (a dog runs `<end>`) and calculate the cross-entropy loss. During training, the model parameters are updated to minimize the loss. Now that we have both the model and the training data ready, we’ll train the model and use it to generate captions for images in the next section.

## 4.4 *Training and using the image-to-text transformer*

We’ll use the Flickr 8k Dataset to train the multimodal image-to-text transformer. We’ll first formally set up the model by using the encoder and decoder defined in the preceding section. We’ll then define the loss function, optimizer, and the training loop.

Once the model is trained, we’ll use it to generate captions for images in the test set. We’ll first define a `caption()` function to add captions to images and then apply the function on images in the test set. Finally, we’ll place both the original caption and the generated caption above the images to compare how good the trained model is.

### 4.4.1 *Training the encoder-decoder transformer*

The `VisionEncoderDecoder()` class we defined in the previous section takes several hyperparameters as arguments. Generally speaking, by passing hyperparameters as arguments, we allow for experimentation with different configurations without modifying the class code. This is particularly important in tasks where the performance

depends on finding the optimal combination of hyperparameters. To save space, we won't perform hyperparameter tuning here. Instead, we'll define a few hyperparameters for the model based on our training dataset and the task at hand:

```
device = "cuda" if torch.cuda.is_available() else "cpu"
hidden_size = 192
num_layers = (6, 6)
num_heads = 8
patch_size = 8
```

Each image token is represented by a 192-value vector.

Number of transformer blocks for the encoder and decoder, respectively

Number of attention heads in multi-head attention

Each image patch is  $8 \times 8$  pixels.

The hyperparameter `hidden_size=192` means that each image patch is represented by a 192-value vector. Both the encoder and the decoder have six transformer blocks. Multiple blocks in both the encoder and decoder are crucial for enhancing the model's capacity to learn complex representations and capture long-range dependencies in the data. We use eight attention heads when calculating multi-head attention. Finally, each image patch has a size of 8, meaning both the width and height of each patch are 8 pixels.

Next, we create an image-to-text transformer by instantiating the `VisionEncoderDecoder()` class we defined in the previous section:

```
from utils.caption_util import VisionEncoderDecoder

caption_model = VisionEncoderDecoder(
    image_size=128, channels_in=3,
    num_emb=len(word2idx), patch_size=patch_size,
    num_layers=num_layers, hidden_size=hidden_size,
    num_heads=num_heads).to(device)
```

The `num_emb` argument in the `VisionEncoderDecoder()` class equals the number of unique tokens in the English vocabulary. The output of the model is the probability distribution over the vocabulary.

We choose the Adam optimizer with a learning rate of 0.0001. Because the model is essentially a multicategory classification problem (which token to choose from the vocabulary), we'll use the cross-entropy loss function:

```
optimizer = torch.optim.Adam(caption_model.parameters(),
                              lr=0.0001)
scaler = torch.amp.GradScaler("cuda")
loss_fn = nn.CrossEntropyLoss(reduction="none")
```

We also use PyTorch's automatic mixed-precision package `torch.amp` to speed up training. The package provides an automatic mixed precision, where some operations use float32 and others float16. Doing so reduces the training time. Let's count the number of parameters in the model:

```

num_model_params = 0
for param in caption_model.parameters():
    num_model_params += param.flatten().shape[0]
print(f"This model has {num_model_params} parameters")

```

The output is as follows, showing that the model has close to 10 million parameters:

This model has 9545219 parameters

We train the model using batches of training data we prepared earlier (see listing 4.8). The weights in the image-to-text transformer we just created are randomly initialized, as is typical for an untrained neural network. During training, the model processes the data to adjust its weights, enabling it to learn associations between images and their corresponding captions.

#### Listing 4.8 Training the image-to-text transformer

```

from tqdm import tqdm

for epoch in range(0, 50):
    caption_model.train()
    eloss=0
    for images,inputs,outputs,masks in tqdm(train_loader):
        images = images.to(device)
        tokens_in = inputs.to(device)
        padding_mask = masks.to(device)
        target_ids = outputs.to(device)
        with torch.amp.autocast("cuda"):
            pred = caption_model(images, tokens_in,
                                padding_mask=padding_mask)
            loss = (loss_fn(pred.transpose(1, 2),
                            target_ids) * padding_mask).mean()
        optimizer.zero_grad()
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
        eloss+=loss.item()
    print(f"epoch {epoch}, loss is {eloss/len(train_loader)}")
    torch.save(caption_model.state_dict(),"files/caption.pth")

```

← **Trains the model for 50 epochs**

**Passes the image and input sequence of the caption through the model to generate predictions**

**Calculates the loss by comparing the predictions with the actual output sequence**

← **Backpropagation: modify model parameters to minimize the cross-entropy loss**

For simplicity, we train the model for 50 epochs. If you like, you can use early stopping to determine how many epochs to train, as we did in chapter 3, based on the performance of the model on a validation set. In each epoch, we iterate through batches of training data, which consist of images, input sequences, target sequences, and the padding mask. The input sequences are indices corresponding to <start> plus the tokens in the captions. For example, if the caption for an image is “a dog runs,” the input sequence is <start> a dog runs while the target sequence is a dog runs <end>. We train the model to predict the next token in the sequence based on all previous tokens.

You can download the pretrained model from my Google drive (<https://mng.bz/X7X1>). Unzip the file after downloading.

#### 4.4.2 Adding captions to images with the trained model

Now that the model is trained, we'll use it to add captions to images. We first define the `caption()` function, as shown in listing 4.9. The function uses the encoder in the trained model to convert an image into the image embedding and passes it to the decoder. The decoder then generates a caption based on the image embedding one token at a time.

**Listing 4.9** Defining a `caption()` function to generate captions

```
def caption(image, temp=1.0):
    sos_token = 1 * torch.ones(1, 1).long()
    log_tokens = [sos_token]
    caption_model.eval()
    with torch.no_grad():
        image_embedding = caption_model.encoder(
            image.to(device))
        for i in range(50):
            input_tokens = torch.cat(log_tokens, 1)
            data_pred = caption_model.decoder(
                input_tokens.to(device), image_embedding)
            dist = Categorical(logits=data_pred[:, -1] / temp)
            next_tokens = dist.sample().reshape(1, 1)
            log_tokens.append(next_tokens.cpu())
            if next_tokens.item() == 2:
                break
        pred_text = torch.cat(log_tokens, 1)
        pred_text_strings = [idx2word.get(i, "<unk>") for
            i in pred_text[0].tolist() if i>3]
        pred_text = " ".join(pred_text_strings)
    return pred_text
```

← Adds the start-of-sentence token <start> to the prompt

← Uses the trained model to encode the image

← Generates a caption one token at a time

← Stops generating when the next token is the end-of-sentence token <end>

The function takes an image as input. It first uses the trained model to encode the image into image embeddings. It then starts the captioning process using the <start> token as the first element in the caption. The model predicts the next token based on the image embeddings and the <start> token. The predicted token is then added to the caption and used as the new caption. In the second iteration, the function predicts the next token based on the updated caption. The process continues until the next predicted token is <end>, which signals the end of the captioning process.

There's an optional argument, `temp`, in the `caption()` function. The default value of `temp` is 1. The creativity of the generated text can be controlled by using the temperature argument, `temp`. Temperature adjusts the distribution of probabilities assigned to each potential token before selecting the next one. It effectively scales the logits, which are the inputs to the softmax function calculating these probabilities, by the value of the temperature. When the temperature is greater than 1, the logits are scaled down. This

makes the softmax probabilities more uniform, meaning the model is more likely to sample less likely tokens. When the temperature is less than 1, the logits are scaled up. This sharpens the softmax probabilities, making the model more confident in its predictions and more likely to choose the most likely tokens. To compare the generated captions with the original captions in the test set, we define the `compare()` function.

#### Listing 4.10 Comparing generated captions with original ones

```
def compare(images, captions, index, temp=1.0):
    image = images[index].unsqueeze(0)
    capi=captions[index]
    capt=[idx2word.get(i,"UNK") for i in capi.tolist() if i>3]
    cap=" ".join(capt)
    pred=caption(image,temp=temp)
    out=torchvision.utils.make_grid(image, 1, normalize=True)
    plt.figure(figsize=(5,10),dpi=100)
    out = torchvision.utils.make_grid(image, 1, normalize=True)
    plt.imshow(out.numpy().transpose((1, 2, 0)))
    plt.title(
        f"*0Original caption:\n"+cap+"\n**Generated caption:\n"+pred,
        wrap=True, loc="left", fontsize=18)
    plt.axis("off")
    plt.show()
```

Retrieves the image from the test set

Retrieves the caption from the test set

Generates a caption using the trained model

Plots the image with both the generated caption and the original caption

The `compare()` function retrieves the image and caption from the test set, based on the `index` argument. It then uses the `caption()` function we defined earlier to generate a caption. We then plot the image with both the generated caption and the original caption on top of the image.

You can apply the `caption()` function on any of the images in the test set. For example, by setting the `index` argument to 0, you can apply the function on the first image:

```
from torch.distributions import Categorical
import torchvision
import matplotlib.pyplot as plt

caption_model.load_state_dict(torch.load("files/caption.pth",
    weights_only=True,
    map_location=device))
compare(test_images, test_tokens, 0, temp=0.75)
```

The first two arguments in the `compare()` function are the images and sequence of indices for the caption tokens. We set these two arguments to `test_images` and `test_tokens`, which means we use examples in the test set. The `index` value is set to 0, and this means we apply the function on the very first image in the test set. We set the `temp` argument to 0.75, so the generated caption is predictable and coherent.

### Exercise 4.3

Apply the `compare()` function on the 10th image in the test set, and set the `temp` argument to 0.95.

I've applied the `compare()` function on all images in the test set. Most generated captions are very accurate: they are coherent and reflective of what's going on in the images. Figure 4.6 gives three such examples.

**\*\*Original caption:**  
the children are playing basketball indoors  
**\*\*Generated caption:**  
the boy is playing basketball in the arena



**\*\*Original caption:**  
two white dogs run through the grass  
**\*\*Generated caption:**  
two white dogs running



**\*\*Original caption:**  
two young girls are running through shallow water at a pool  
**\*\*Generated caption:**  
two girls are running away from the water and laughing



**Figure 4.6 Three examples of generated captions with the trained encoder–decoder transformer in which both the generated captions and the original captions appear above each image for comparison**

We display both the generated captions and the original captions above the images. For example, the generated caption for the first image is “the boy is playing basketball in the arena.” Even though it’s different from the original caption, “the children are playing basketball indoors,” the generated caption does reflect what’s going on in the first image.

Starting from the next chapter, we’ll discuss another method of text-to-image generation, diffusion models. We’ll start with simple diffusion models so that you have a solid foundation in how diffusion models work before embarking on text-to-image generation in later chapters.

## Summary

- Training a multimodal transformer for text-to-image generation (i.e., adding captions to images) and training one for image-to-text generation (i.e., generating images based on textual descriptions) share some similarities. Both tasks involve learning complex mappings between textual and visual modalities.

- Understanding and training multimodal transformers for image-to-text generation first can provide valuable insights and foundational knowledge, making the complex task of text-to-image generation more manageable.
- We create an encoder–decoder transformer for image-to-text generation. The encoder converts an image into a sequence of image patches. It then uses image embedding to convert image patches to vector representations that capture the feature maps in these patches. The decoder generates captions one token at a time, based on the output from the encoder and previously generated tokens in the caption.
- When preparing the training data, we generate input sequences and target sequences based on captions associated with the training images. The decoder tries to predict the next token after the input sequence:
  - For example, if the caption associated with an image is “a dog runs,” we’ll add <start> and <end> to the beginning and the end of the caption. We’ll use <start> a dog runs as the input sequence. We move the input sequence one token to the right and use it as the target sequence (i.e., the ground truth).
  - During training, we feed the input sequence to the model, compare the model output with the target sequence, a dog runs <end>, and calculate the cross-entropy loss. We modify model parameters to minimize this loss.



## *Part 2*

# *Introduction to diffusion models*

**D**iffusion models have rapidly emerged as one of the most powerful techniques for generative tasks, especially in image generation. We walk you through the core idea in chapter 5: generating an image by reversing a noise-adding process. You'll implement a diffusion model step-by-step, starting from the basics of forward and reverse diffusion processes.

Once the fundamentals are clear, we move into conditioning and scaling. In chapter 6, you'll learn how to guide diffusion models with conditioning information so that the outputs aren't random but aligned with your intent. Chapter 7 tackles the challenge of generating high-resolution images and exploring techniques that allow diffusion models to scale up in quality and fidelity. These chapters give you the conceptual and coding foundation for understanding how today's state-of-the-art text-to-image models are built.



# 5

## Generate images with diffusion models

---

### ***This chapter covers***

- How the forward diffusion process gradually adds noise to images
- How the reverse diffusion process iteratively removes noise
- Training a denoising U-Net model from scratch
- Using the trained model to generate new clothing-item images

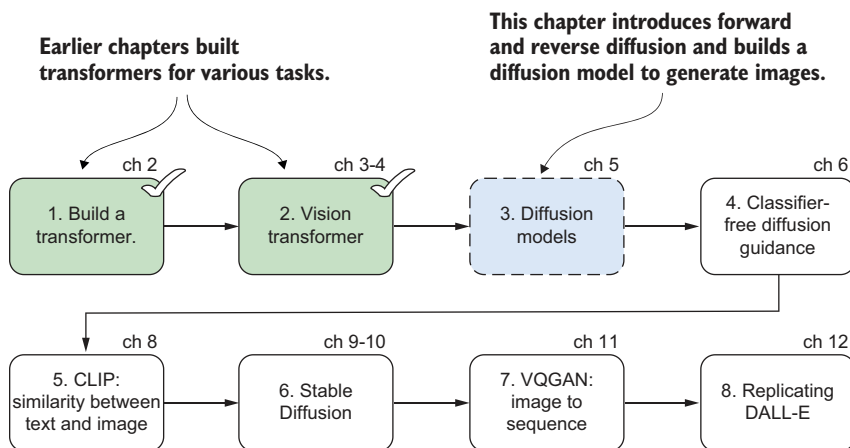
Text-to-image generation has seen remarkable progress in recent years, largely thanks to two classes of models: vision transformers (ViTs) and diffusion models. Diffusion models create images through a two-step process. First, they learn to gradually add random noise to clean images, step-by-step, until the images become pure noise. This is called the *forward diffusion process*. Then, the models are trained to remove noise from images in the *reverse diffusion process*: starting with pure noise, a diffusion model learns to iteratively remove noise, guided by learned patterns, until a new, clean image emerges. By controlling each small denoising step, diffusion

models can generate high-resolution images that surpass the quality of images generated by other approaches (variational autoencoders or generative adversarial networks).

Understanding how diffusion works is essential because it forms the foundation for state-of-the-art text-to-image generation, including popular models such as DALL-E 2, Imagen, and Stable Diffusion. That's why this chapter takes a hands-on, practical approach in which you'll build a simple diffusion model to generate grayscale images of clothing items using the Fashion Modified National Institute of Standards and Technology (Fashion MNIST) dataset, a collection of 70,000 grayscale images of clothing items (e.g., shirts, shoes, and bags). By training your own model from scratch, you'll gain an intuitive, working knowledge of both the forward and reverse diffusion processes.

We'll also explore the role of the noise schedule, the mathematical rule governing how much noise is added at each step. Different schedules (e.g., linear or cosine) have different effects on the learning dynamics and the final image quality. Here, you'll experiment with these choices and see their effects firsthand.

Figure 5.1 illustrates how this chapter fits into your journey of building a text-to-image generator from scratch. We'll focus on step 3: understanding forward and reverse diffusion and training a diffusion model from scratch.



**Figure 5.1** The eight-step road map for building a text-to-image generator from scratch. This chapter is dedicated to step 3: diving deep into the mechanics of forward and reverse diffusion and developing a diffusion model capable of generating images from noise.

By building your own diffusion model and visualizing each step, you'll not only demystify a central technique in modern generative AI but also lay the groundwork for more advanced methods, including conditional diffusion for text-to-image generation, which is covered in later chapters. The knowledge and intuition you gain here will

enable you to confidently create, experiment with, and understand diffusion models at both a conceptual and implementation level.

## 5.1 The forward diffusion process

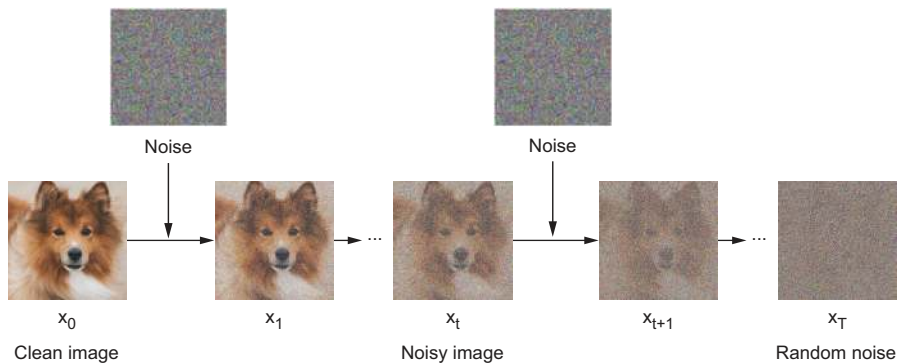
To understand the idea of diffusion-based models, imagine you want to train a model to generate high-resolution images. You first acquire a set of images for training and then gradually introduce small amounts of random noise into these images. This process is known as forward diffusion. After many steps of adding noise, the training images become pure random noise.

Then, using reverse diffusion (or denoising), you'll train a model to reverse this process. Specifically, you'll start with random noise images and progressively reduce the noise until the images are indistinguishable from those in the original training set. The trained diffusion model can use random noise images as a starting point. It gradually eliminates noise from the image over many iterations until the output is a clean image. This is the underlying principle of diffusion-based models.

In this section, we'll first explore the theoretical foundations of diffusion-based models. Then, we'll examine two different ways of noise scheduling: the linear noise schedule and the cosine noise schedule. We'll compare them side by side to see how noise is added to images until they become random noise images, similar to snow on a TV screen.

### 5.1.1 How diffusion models work

The idea behind diffusion models is proposed by several papers with similar underlying mechanisms [1]. In forward diffusion, noise is gradually added to a clean image until it becomes pure noise. The step-by-step addition of noise is essential to create a controllable and learnable process that bridges the gap between structured data and random noise. It enables the model to learn incremental denoising steps during training so that the trained model provides an effective and stable reverse process for generating images. Figure 5.2 is a diagram of how the forward diffusion process works.



**Figure 5.2** The forward diffusion process

We start with a clean image,  $x_0$ , which is illustrated in the left image of a dog in figure 5.2. The image  $x_0$  follows a distribution of  $q(x)$ . We can normalize the value of  $x$  so that it has zero mean and unit standard deviation. In the forward diffusion process, we'll add small amounts of noise to the image in each of the  $T$  time steps (e.g.,  $T = 1000$ ). The noise tensor follows a standard normal distribution and has the same shape as the clean image. Eventually,  $x_T$  approximates a standard normal distribution.

In time step 1, we add noise  $\epsilon_0$  to the image  $x_0$  so that we obtain a noisy image  $x_1$ , as follows:

$$x_1 = \sqrt{1 - \beta_1}x_0 + \sqrt{\beta_1}\epsilon_0 \quad (5.1)$$

Here,  $x_1$  is a weighted sum of  $x_0$  and  $\epsilon_0$ , and  $\sqrt{\beta_1}$  measures the weight placed on the noise. The value of  $\beta$  changes in different time steps, hence the subscript in  $\beta_1$ . Because we assume  $x_0$  and noise  $\epsilon_0$  both have a mean of 0 and variance of 1, the noisy image  $x_1$  will also have zero mean and unit variance. This is easy to prove as

$$\text{mean}(x_1) = \sqrt{1 - \beta_1}\text{mean}(x_0) + \sqrt{\beta_1}\text{mean}(\epsilon_0) = 0$$

and

$$\begin{aligned} \text{var}(x_1) &= \text{var}(\sqrt{1 - \beta_1}x_0) + \text{var}(\sqrt{\beta_1}\epsilon_0) = \\ &= (1 - \beta_1)\text{var}(x_0) + \beta_1\text{var}(\epsilon_0) = 1 - \beta_1 + \beta_1 = 1 \end{aligned} \quad (5.2)$$

Let's keep adding noise to the image for the next  $T-1$  time steps so that

$$x_{t+1} = \sqrt{1 - \beta_{t+1}}x_t + \sqrt{\beta_{t+1}}\epsilon_t \quad (5.3)$$

If we use a reparameterization trick and define  $\bar{\alpha}_t = 1 - \beta_t$  and  $\bar{\alpha}_t = \prod_{k=1}^t \alpha_k$ , we can sample  $x_t$  at any arbitrary time step  $t$ , where  $t$  can take any value in  $[1, 2, \dots, T-1, T]$ :

$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon \quad (5.4)$$

Here,  $\epsilon$  is a combination of  $\epsilon_0, \epsilon_1, \dots$ , and  $\epsilon_{t-1}$ , using the fact that we can add two normal distributions to obtain a new normal distribution. (For proof of this, you can see, for example, the blog of Lilian Weng at <https://mng.bz/a9G9>).

We'll then construct a deep neural network as the denoising model to reverse the diffusion process, which is called the denoising process. If we can train a model to reverse the forward diffusion process, we can feed the model with random noise and ask it to produce a noisy image. We can then feed the noisy image to the trained model again and obtain a clearer, though still noisy, image. We can iteratively repeat the process for many time steps until the output is a clean image. This is the idea behind diffusion

models. In section 5.2, we'll discuss in detail the reverse diffusion process, and in section 5.4, we'll cover how to train the diffusion model.

**NOTE** The Python programs for this chapter can be accessed on the book's GitHub repository (<https://github.com/markhliu/txt2img>) and are also available as a Google Colab notebook (<https://mng.bz/gmW8>) for interactive exploration.

### 5.1.2 Visualizing the forward diffusion process

The value of  $\beta_t$  changes in different time steps, hence the subscript in  $\beta_t$ . There are different ways of modeling the relation between  $\beta_t$  and the time step  $t$ , for example, using a linear diffusion schedule [2]. That is, the value of  $\beta_t$  increases linearly with the time step  $t$ , ranging from  $\beta_1 = 0.0001$  to  $\beta_T = 0.02$ . Let's code that in Python.

**Listing 5.1** The linear diffusion schedule

```
import torch

T=1000
t=torch.arange(0, T + 1, dtype=torch.float32)/T
def linear_scheduler():
    beta1, beta2 = 0.0001, 0.02
    beta_t = (beta2 - beta1) * t + beta1
    alpha_t = 1 - beta_t
    log_alpha_t = torch.log(alpha_t)
    alphabar_t = torch.cumsum(log_alpha_t, dim=0).exp()
    sqrtab = torch.sqrt(alphabar_t)
    sqrtmab = torch.sqrt(1 - alphabar_t)
    return {"sqrtab": sqrtab, "sqrtmab": sqrtmab}
```

$\beta_t$  increases linearly with time step  $t$ .

The weight placed on the clean image  $x_0$

The weight placed on noise  $\epsilon_0$

We assume a total of  $T = 1,000$  time steps. For simplicity, we define a list of 1,001 relative time steps  $t = [0/1000, 1/1000, 2/1000, \dots, 1000/1000]$ . We then define a function `linear_scheduler()`. The value of  $\beta_t$  increases linearly with the time step  $t$ . The function returns a dictionary with two key-value pairs: the first one is the weight on the clean image  $x_0$ ,  $\sqrt{\alpha_t}$ , and the second is the weight on noise  $\epsilon_0$ ,  $\sqrt{1 - \alpha_t}$ .

To visualize how the forward diffusion process works, let's apply the process to a few images. Go to the book's GitHub repository (<https://github.com/markhliu/txt2img>), and download the following four images from the `/files/` folder: `bird.png`, `dog.png`, `mountains.png`, and `horse.png`. Place them in the `/files/` folder on your computer (or simply clone the repository to your computer). If you're working in Google Colab, you can quickly set up the repository and add it to your working directory by running

```
!git clone https://github.com/markhliu/txt2img
import sys
sys.path.append("/content/txt2img")
```

Run the following code block to visualize the forward diffusion process.

**Listing 5.2 Forward diffusion with a linear diffusion schedule**

```

import matplotlib.pyplot as plt
import PIL
import numpy as np
def linear_noisy_image(image,timestep):
    alphabar_t=linear_scheduler()["sqrtab"][timestep]
    noisy=image*torch.sqrt(alphabar_t)+\
        torch.randn_like(image)*torch.sqrt(1 - alphabar_t)
    return torch.clip(noisy,min=-1,max=1)
imgs=[]
for name in ["bird","dog","mountains","horse"]:
    img=np.array(PIL.Image.open(f"files/{name}.png"))
    img=torch.tensor(2*(img/255)-1)
    for timestep in [0,200,400,600,800,1000]:
        imgs.append(linear_noisy_image(img,timestep)/2+0.5)
plt.figure(figsize=(12,8),dpi=100)
for i in range(24):
    plt.subplot(4,6,i+1)
    plt.imshow(imgs[i])
    if i<6:
        plt.title(f't={200*(i%6)}', fontsize=12, c="r")
    plt.axis("off")
plt.tight_layout()
plt.show()

```

**Defines a function to generate noisy images from clean images with a linear diffusion schedule**

**Generates noisy images at time steps 0, 200, 400, ..., 1,000**

**Plots the noisy images in four rows and six columns**

We first define a function `linear_noisy_image()` to add noise to an image. The function takes two arguments: `image`, which is a clean image that you want to add noise to, and `timestep`, which is an integer that can take a value between 0 and 1,000. The result is a noisy image based on the formula in equation 5.4.

We then iterate through the 4 images and the time steps 0, 200, 400, ..., 1,000 and use the function `linear_noisy_image()` to generate 24 images. The 24 images are plotted in four rows and six columns, as shown in figure 5.3. The 4 images in the first column are clean images downloaded from the book's GitHub repository. We then gradually add noise to these images. As the time step increases, more and more noise is injected into the images. The 4 images in the second column are images after 200 time steps. The third column contains images after 400 time steps, showing more noise than those in the second column. The last column contains images after 1,000 time steps, showing pure random noise.

**Exercise 5.1**

Go to the book's GitHub repository (<https://github.com/markhliu/txt2img>), and download the following two images from the `/files/` folder: `bunny.png` and `cat.png`. Place them in the `/files/` folder on your computer. Iterate through the 2 images and the time steps 0, 250, 500, 750, and 1,000. Use the function `linear_noisy_image()` to generate 10 images. Plot the 10 images in a  $2 \times 5$  grid.



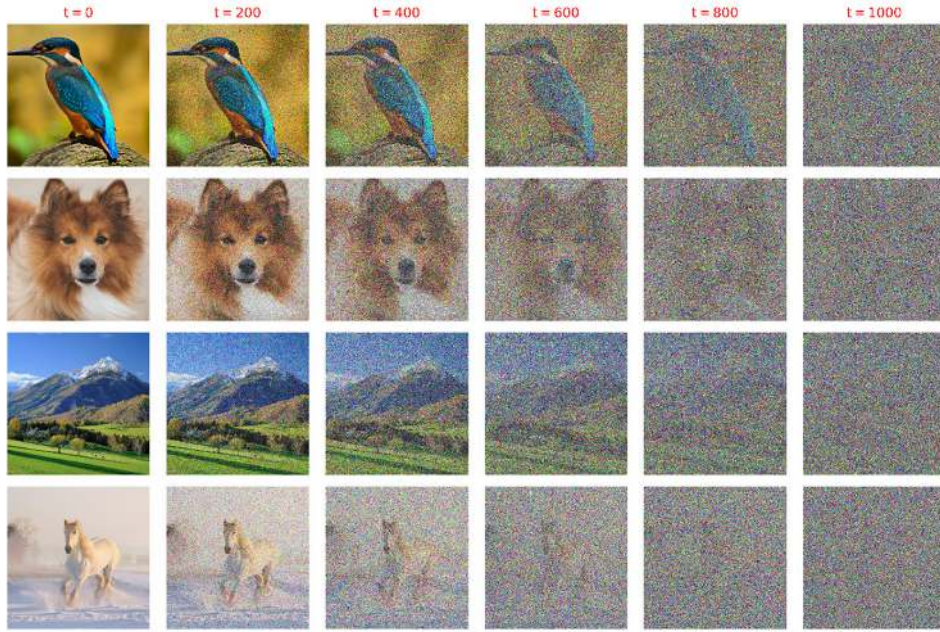


Figure 5.3 The forward diffusion process

### 5.1.3 Different diffusion schedules

In 2021, Alex Nichol and Prafulla Dhariwal proposed a new noise schedule [3]—a cosine diffusion schedule. Because noisy images are used to train the denoising diffusion models, the noise schedule affects how noise is added to images, which, in turn, affects the effectiveness of the trained diffusion model. The authors found that the cosine noise schedule outperformed (in image quality produced by trained models) the linear diffusion schedule proposed in the paper by Ho et al. in 2020 [2].

Specifically, in the proposed cosine diffusion schedule, the weights placed on the clean images and noise are

$$\sqrt{\bar{\alpha}_t} = \cos\left(\frac{\pi t}{2T}\right), \sqrt{1 - \bar{\alpha}_t} = \sin\left(\frac{\pi t}{2T}\right)$$

That is, we can sample  $x_t$  like this:

$$x_t = \cos\left(\frac{\pi t}{2T}\right)x_0 + \sin\left(\frac{\pi t}{2T}\right)\epsilon \quad (5.5)$$

Compared to the linear diffusion schedule, the cosine diffusion schedule adds noise to the image more uniformly from time step 1 to time step T. To see this, run the following code block.

## Listing 5.3 Comparing linear and cosine diffusion schedules

```

import math
def cosine_scheduler():
    sqrttab = torch.cos(0.5*math.pi*t)
    sqrtmab = torch.sin(0.5*math.pi*t)
    return {"sqrttab": sqrttab, "sqrtmab": sqrtmab}

plt.figure(figsize=(10,6),dpi=100)
plt.plot(t, linear_scheduler()["sqrtmab"], linewidth=3,
         linestyle="solid",c="r", label="linear schedule")
plt.plot(t, cosine_scheduler()["sqrtmab"], linewidth=3,
         linestyle="dotted",c="g", label="cosine schedule")
plt.xlabel(r"relative time, t/T", fontsize=12)
plt.ylabel(r"weight on $\epsilon$: $\sqrt{1-\bar{\alpha}_t}$",
          fontsize=12)
plt.legend()
plt.show()

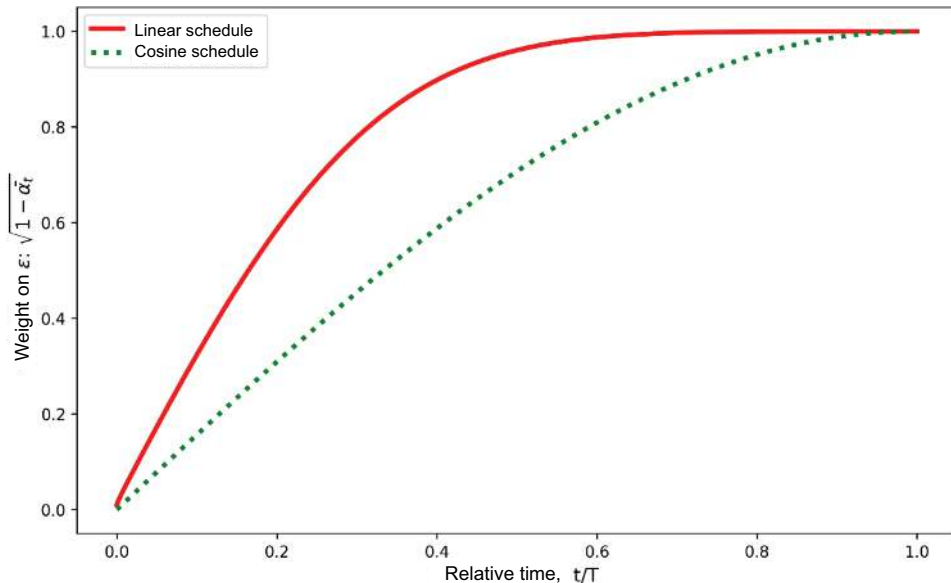
```

Defines the cosine diffusion scheduler

Plots the weight on the noise over time under the linear scheduler

Plots the weight on the noise over time under the cosine scheduler

We first define a function `cosine_scheduler()` to define the weight placed on the clean image and that on the noise over time based on the cosine diffusion schedule. We then plot the weights on the noise under both the linear diffusion schedule and the cosine schedule in the same graph. The output from listing 5.3 is shown in figure 5.4.



**Figure 5.4** The weight placed on noise under linear and cosine diffusion schedules. The x-axis is the relative time step,  $t/T$ , which ranges from 0 to 1. The y-axis is the weight placed on the noise in the forward diffusion process. The solid line represents the linear diffusion schedule, while the dotted line represents the cosine diffusion schedule. The weight on the noise under the cosine diffusion schedule increases over time more gradually compared to that under the linear diffusion schedule.

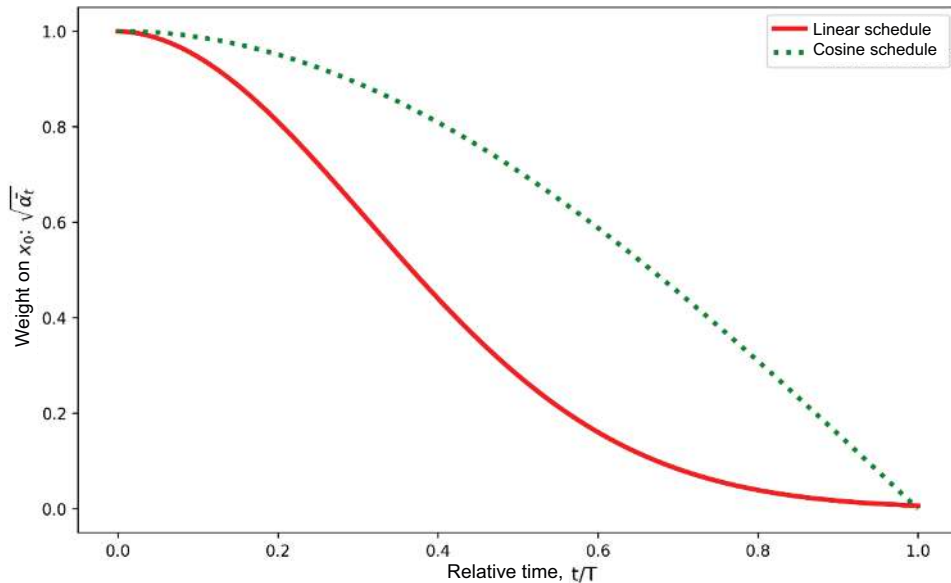
The solid line in figure 5.4 is the weight on the noise over time under the linear diffusion schedule. The weight is flat once the relative time step  $t/T$  exceeds 0.6. In contrast, the dotted line, which represents the weight on the noise under the cosine diffusion schedule, increases over time gradually, leading to better learning in diffusion models.

We can also plot the weight on the clean image over time under the two diffusion schedules. Run the following code block to produce the output shown in figure 5.5:

```
plt.figure(figsize=(10,6),dpi=100)
plt.plot(t, linear_scheduler()["sqrtab"], linewidth=3,
         linestyle="solid",c="r", label="linear schedule")
plt.plot(t, cosine_scheduler()["sqrtab"], linewidth=3,
         linestyle="dotted",c="g", label="cosine schedule")
plt.xlabel(r"relative time,  $t/T$ ", fontsize=12)
plt.ylabel(r"weight on  $x_0$ :  $\sqrt{\alpha_t}$ ",
           fontsize=12)
plt.legend()
plt.show()
```

Plots the weight  
on the clean image  
over time under the  
linear scheduler

Plots the weight  
on the clean image  
over time under the  
cosine scheduler



**Figure 5.5** The weight placed on the clean image under linear and cosine diffusion schedules. The x-axis is the relative time step,  $t/T$ , which ranges from 0 to 1. The y-axis is the weight placed on the clean image in the forward diffusion process. The solid line represents the linear diffusion schedule, and the dotted line represents the cosine diffusion schedule. The weight on the clean image under the cosine diffusion schedule decreases over time more gradually compared to that under the linear diffusion schedule.

The solid line shows the weight on the clean image over time under the linear diffusion schedule. The weight is flat once the relative time step  $t/T$  exceeds 0.8. In contrast, the dotted line shows that the weight on the clean image under the cosine diffusion schedule decreases over time more gradually.

To see how this affects the actual diffusion process, download the image `bunny.png` from the book's GitHub repository, and place it under the `/files/` folder on your computer. Then run the following code block, which creates the output shown in figure 5.6.

#### Listing 5.4 Noisy images in linear and cosine diffusion schedules

```
def cosine_noisy_image(image,timestep):
    alphabar_t=cosine_scheduler()["sqrtab"][timestep]
    noisy=image*torch.sqrt(alphabar_t)+\
        torch.randn_like(image)*torch.sqrt(1 - alphabar_t)
    return torch.clip(noisy,min=-1,max=1)

img=np.array(PIL.Image.open("files/bunny.png"))
img=torch.tensor(2*(img/255)-1)
imgs=[]

for timestep in [0,166,332,498,664,830,996]:
    imgs.append(linear_noisy_image(img,timestep)/2+0.5)
for timestep in [0,166,332,498,664,830,996]:
    imgs.append(cosine_noisy_image(img,timestep)/2+0.5)

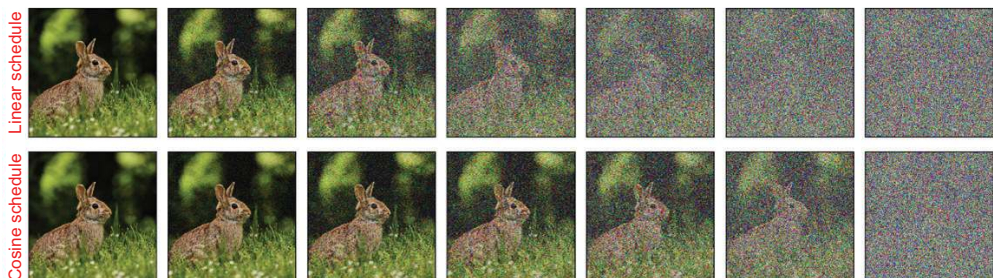
plt.figure(figsize=(13,4),dpi=100)
for i in range(14):
    plt.subplot(2,7,i+1)
    plt.imshow(imgs[i])
    plt.xticks([])
    plt.yticks([])
    if i==0:
        plt.ylabel("linear schedule",fontsize=15,c="r")
    if i==7:
        plt.ylabel("cosine schedule",fontsize=15,c="r")
plt.tight_layout()
plt.show()
```

Defines a function to generate noisy images using the cosine diffusion schedule

Generates noisy images using the linear diffusion schedule

Generates noisy images using the cosine diffusion schedule

Plots the noisy images in two rows and seven columns



**Figure 5.6** How noise is gradually added to images in the forward diffusion process under linear and cosine diffusion schedules

In figure 5.6, gradually add noise to a clean image under a linear diffusion schedule (top row) and a cosine diffusion schedule (bottom row). The first column contains clean images (i.e.,  $t = 0$ ). As we move to the right, the time steps are 166, 332, 498, 664, 830, and 996, respectively. The linear diffusion schedule adds noise to the image more quickly, while the cosine diffusion schedule adds noise more gradually over time.

The top row of images shows how noise is added to the bunny image under the linear diffusion schedule. The bottom row shows how noise is added to the same bunny image under the cosine diffusion schedule. As you can see, noise is added to the image more gradually and more evenly over time under the cosine diffusion schedule. This is why the cosine diffusion schedule leads to better performance. However, the linear diffusion schedule is easier to code and implement. Which diffusion schedule to choose depends on the specific task and the balance between performance and implementation complexity.

### Exercise 5.2

Download the image `cat.png` from the book's GitHub repository, and place it under the `/files/` folder on your computer. Gradually add noise to the image, and save the images in time steps 0, 199, 399, 599, 799, and 999 under the linear diffusion schedule first and then the cosine diffusion schedule. Plot the images in a  $2 \times 6$  grid.

## 5.2 The reverse diffusion process

Now that you understand the forward diffusion process, let's discuss the reverse diffusion process (i.e., the denoising process). If we can train a model to reverse the forward diffusion process, we can feed the model with random noise and ask it to produce a less noisy image. We can iteratively repeat the process for many time steps until we obtain a clean image. A U-Net is the key building block of a denoising diffusion model, which we'll elaborate on in this chapter and the next.

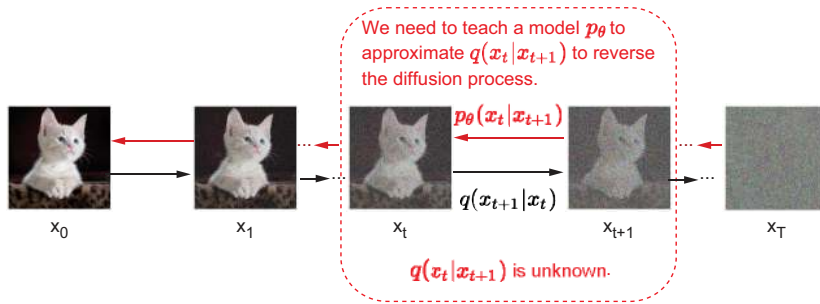
A *U-Net* is a type of neural network commonly used in denoising diffusion models due to its ability to process and generate high-quality images. It typically uses convolutional neural networks (CNNs) to extract spatial features from the input images. U-Nets have an encoder-decoder structure with skip connections that directly link layers of the same resolution in the encoder and decoder. This design allows the U-Net to capture both global context and fine-grained details, making it well-suited for tasks such as denoising and image generation.

In the context of diffusion models, the U-Net learns to reverse the diffusion process by predicting the noise at each step, enabling the gradual reconstruction of clean images from noisy inputs. In the next chapter, you'll learn how to create a U-Net from scratch. In this chapter, we'll use the U-Net from the `diffusers` library. Run the following code cell to install the `diffusers` library on your computer:

```
!pip install diffusers
```



In the forward diffusion process, we add noise to the image in time step  $t$ ,  $x_t$ , to obtain the image in time step  $t + 1$ ,  $x_{t+1}$ , based on a function  $q(x_{t+1}|x_t)$ . However, upon observing  $x_{t+1}$ , we can't obtain  $x_t$  because the function  $q(x_t|x_{t+1})$  is unknown. Therefore, we need to use the training data to teach a model to approximate the conditional probability  $p_\theta(x_t|x_{t+1})$ , where  $\theta$  represents the model parameters. The idea is shown in figure 5.7.



**Figure 5.7** How the reverse diffusion process works. Upon observing  $x_{t+1}$ , we can't obtain  $x_t$  because the function  $q(x_t|x_{t+1})$  is unknown. Therefore, we need to use the training data to teach a model to approximate the conditional probability  $p_\theta(x_t|x_{t+1})$ , where  $\theta$  represents the model parameters.

In this chapter, we'll import the U-Net model from the `diffusers` library (see listing 5.5) and treat it as a black box. Doing so allows us to focus on the training process of the U-Net (instead of creating one from scratch). The denoising U-Net takes a noisy image,  $x_t$ , and the time step it's in,  $t$ , as inputs. The output is the noise injected into the noisy image,  $\epsilon$ . Knowing the noise  $\epsilon$  allows us to back out the original clean image  $x_0$  based on equation 5.4. U-Net can capture both local and global context, where local context involves information from a small, immediate neighborhood around each pixel, while the global context refers to the broader, overall structure of the image. This makes it effective for removing noise while preserving important details such as edges and textures. We'll discuss more on the U-Net architecture in the next chapter.

#### Listing 5.5 Importing a U-Net model from the `diffusers` library

```
import diffusers, torch

device="cuda" if torch.cuda.is_available() else "cpu"
model=diffusers.UNet2DModel(sample_size=32,
                             in_channels=1,
                             out_channels=1, layers_per_block=2,
                             block_out_channels=(128,128,256,512),
                             down_block_types=("DownBlock2D", "DownBlock2D",
                                                "AttnDownBlock2D", "DownBlock2D"),
```

Specifies the width and height of the input and output images

Defines the number of feature maps each block will output

Specifies the types of blocks used in the encoder in the U-Net

```
up_block_types=("UpBlock2D", "AttnUpBlock2D",
               "UpBlock2D", "UpBlock2D",),).to(device)
```

Specifies the types of blocks  
used in the decoder in the U-Net

The `UNet2DModel()` class in the `diffusers` library creates a U-Net model based on several arguments. The `sample_size` argument specified the width and height of the input and output images. We set the value to 32 because we'll resize the clothing-item images from  $1 \times 28 \times 28$  to  $1 \times 32 \times 32$  to fit the `UNet2DModel()` class specification.

The argument `block_out_channels` defines the number of feature maps (channels) that each block in the U-Net will output. As you'll learn in the next couple of chapters, U-Net is built with an encoder-decoder architecture. The encoder progressively reduces the spatial dimensions of the input through downsampling while increasing the depth (number of channels). The decoder progressively increases the spatial dimensions through upsampling while refining the depth.

The `down_block_types` and `up_block_types` parameters define the types of blocks used in the encoder (downsampling) and decoder (upsampling) parts of the U-Net architecture, respectively. These parameters allow you to customize the architecture of the U-Net by choosing different block types, including those with attention mechanisms. The attention mechanism allows the model to focus on specific parts of the input or output features.

The input to the model is a noisy image,  $x_t$ , along with the time step the image is in,  $t$ . The model will predict the noise in the image. Once we know the noise in the image, we can back out the original clean image  $x_0$ .

## 5.3 A blueprint to train the U-Net model

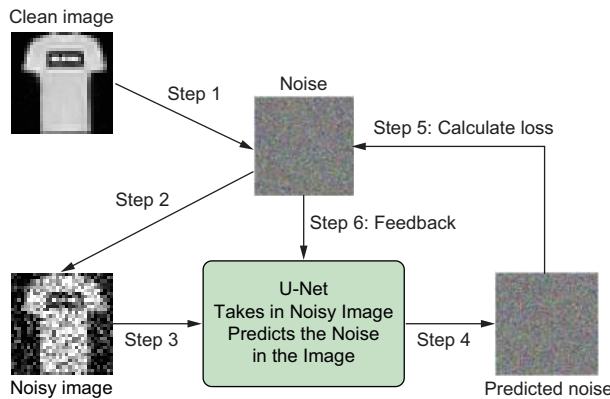
In this section, we'll discuss the steps in training a denoising U-Net model. We'll then preprocess the training data to get it ready for the next section when we perform the actual training.

### 5.3.1 Steps in training a denoising U-Net model

The denoising U-Net takes a noisy image and the time step it's in as inputs. The output is the noise injected into the noisy image. The model is trained to minimize the difference between the output (predicted noise) and the ground truth (actual noise).

The denoising U-Net model uses the U-Net architecture's ability to capture both local and global contexts. These models are widely used in various applications, including medical image denoising and photographic image restoration. Figure 5.8 shows the steps in training a denoising U-Net model.

We first obtain clean clothing-item images as our training set (step 1). Each batch of training data contains 128 images. We randomly choose a time step  $t$  for each of these 128 images where the value of  $t$  ranges from 1 to 1,000. We add noise to these clean images depending on the value of  $t$  for each image based on the formula in equation 5.4. We present the 128 noisy images (step 2), along with the time steps these images are



**Figure 5.8** Steps to train a denoising U-Net model to remove noise from images

in, to the U-Net model (step 3). The model predicts the noise in the noisy images (step 4). We compare the predicted noise with the ground truth (the actual noise injected into the images) and calculate the loss, which is measured by the L1 loss (mean absolute error; step 5). We adjust the model parameters to minimize the loss (step 6).

### 5.3.2 Preprocessing the training data

We'll use the Fashion MNIST dataset in this project. We'll use the datasets and transforms packages in the TorchVision library, as well as the DataLoader packages in PyTorch to help preprocess data.

In the following code block, we first import needed libraries and instantiate a `Compose()` class in the transforms packages to transform raw images to PyTorch tensors:

```
from torchvision import transforms

torch.manual_seed(42)  # Fixes the random state so results are reproducible

tf = transforms.Compose(
    [transforms.Resize((32,32)), transforms.ToTensor(),
     transforms.Lambda(lambda x: 2*(x-0.5)),]  # Converts images to PyTorch tensors with values in the range [-1,1]
)
```

The `manual_seed()` method in PyTorch is used to fix the random state so that results are reproducible. The transforms package in TorchVision can help create a series of transformations to preprocess images. We resize the images from (1, 28, 28) to (1, 32, 32) so that the images fit the input dimensions of the U-Net model we created in the preceding section. The upscaling is a cost of using an off-the-shelf U-Net model here. In the next chapter, we'll create our own U-Net from scratch with no need to upscale the input image. The `ToTensor()` class converts image data (in either PIL image formats or NumPy arrays) into PyTorch tensors. In particular, the image data are integers ranging from 0 to 255, and the `ToTensor()` class converts them to float tensors with



values between 0.0 and 1.0. We then normalize the input data to the range  $[-1, 1]$  for faster convergence during training, which we'll do often in this book.

Next, we use the `datasets` package in `TorchVision` to download the dataset to a folder on your computer and perform the transformation:

```
import torchvision

dataset = torchvision.datasets.FashionMNIST(
    root=".",
    train=True,
    download=True,
    transform=tf,
)
```

There are 10 different types of clothing items. The labels in the dataset are numbered from 0 to 9. The list `text_labels` contains the 10 text labels corresponding to the numerical labels 0 to 9. For example, if an item has a numerical label of 0 in the dataset, the corresponding text label is "t-shirt." The list `text_labels` is defined as follows:

```
text_labels=['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
            'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
```

We can plot the data to visualize the clothes items in the dataset as follows, with the output shown in figure 5.9:

```
import matplotlib.pyplot as plt
plt.figure(dpi=200,figsize=(8,4))
for i in range(24):
    ax=plt.subplot(3, 8, i + 1)
    img=dataset[i+888][0]
    img=img/2+0.5
    img=img.reshape(32,32)
    plt.imshow(img,
               cmap="binary")
    plt.axis('off')
    plt.title(text_labels[dataset[i+888][1]],fontsize=8)
plt.show()
```

We group the training data into batches to maximize computational efficiency. Batch processing enables parallel computation on GPUs and CPUs, allowing the model to update its weights more efficiently and significantly speeding up training:

```
from torch.utils.data import DataLoader

dataloader = DataLoader(dataset, batch_size=128, shuffle=True)
```

Now we're ready to train the denoising U-Net model we just created. Let's tackle that next in the following section.



**Figure 5.9** Grayscale images of clothing items in the Fashion MNIST dataset. There are 10 different types of clothing items, labeled from 0 to 9. The 10 text labels corresponding to the numerical labels 0 to 9 are t-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot, respectively. The text label is on top of each image in this figure.

## 5.4 *Training and using the diffusion model*

In the preceding section, we created a denoising U-Net model and prepared the training data. In this section, we'll train the model and use it to generate clean clothing-item images.

To that end, we'll first create a noise scheduler using the `diffusers` library to add noise to clean images to obtain noisy images. We'll then use the noisy images, along with the time steps the images are in, to train the denoising U-Net.

Once the model is trained, we'll feed random noise to the model to generate a noisy image. We'll then feed the noisy image to the trained model to produce a less noisy image. We repeat the denoising process over many (say, 1,000) time steps. As a result, we obtain a clean image that's indistinguishable from those in the training dataset.

### 5.4.1 *The Denoising Diffusion Probabilistic Model noise scheduler*

Earlier in the chapter, we discussed the idea behind noise schedulers. In the next chapter, we'll code from scratch a full-fledged noise scheduler that can handle both the forward and the reverse diffusion processes. In this chapter, we'll rely on one of the noise schedulers in the `diffusers` library. Specifically, we define the Denoising Diffusion Probabilistic Model (DDPM) noise scheduler in our model as follows:

```
noise_scheduler = diffusers.DDPScheduler(
    num_train_timesteps=1000,
    beta_schedule="squaredcos_cap_v2")
```

The `DDPMScheduler()` class from the `diffusers` library is used here. The argument `num_train_timesteps=1000` specifies the number of time steps used in the forward diffusion process. This choice of 1,000 steps is the standard practice in most diffusion models because it strikes a practical balance between image quality and computational efficiency. Too few steps can make the denoising process too abrupt, resulting in poorer image quality, while too many steps significantly increase computation time with only marginal gains in sample quality.

The `squaredcos_cap_v2` diffusion scheduler is a component of the `diffusers` library, specifically used in the DDPM framework. Here, *probabilistic* means that the entire diffusion and denoising process is governed by probability distributions, capturing the inherent randomness and uncertainty in image generation rather than following a single deterministic path. This scheduler is responsible for managing the diffusion process, which involves gradually adding noise to an image and then learning to denoise it to generate new samples. It uses a cosine-based schedule with specific adjustments to improve the training stability and sample quality. `squaredcos_cap_v2` modifies the noise schedule to better match the denoising process, potentially leading to better sample quality. The `cap` in the name indicates that there is a capping mechanism to ensure that noise levels don't exceed certain limits, preventing excessive noise addition and helping maintain stability.

#### 5.4.2 Inference using the U-Net denoising model

During training, we'll periodically ask the model to generate clean images so that we can monitor the performance of the model. For that purpose, we define the following `sample()` function.

**Listing 5.6 Generating images using the trained model**

```
@torch.no_grad()
def sample(n_sample, model, noise_scheduler, seed=None):
    if seed is not None:
        torch.manual_seed(seed)
    noise_scheduler.set_timesteps(1000)
    image=torch.randn((n_sample,1,32,32)).to(device)
    for t in noise_scheduler.timesteps:
        model_output=model(image,t)['sample']
        image=noise_scheduler.step(model_output,int(t),
                                  image,generator=None)['prev_sample']
    return image
```

**Starts with pure random noise**

**Iterates over many time steps**

**Predicts the noise in the image using the trained model**

**Backs out what the image looks like in the previous time step**

The `@torch.no_grad()` decorator in PyTorch is used to disable gradient computation during the execution of a block of code or a function. This is useful in scenarios where you don't need to calculate gradients, such as during model inference or evaluation, as it improves performance and reduces memory consumption. The `sample()` function

we defined previously in listing 5.6 is used to generate images using the trained model, so we use the `@torch.no_grad()` decorator to disable gradient computation.

The `sample()` function takes several arguments. The first argument `n_sample` specifies how many images you want to generate. The argument `model` is the denoising U-Net model you want to use to generate images, and the `noise_scheduler` argument specifies the noise scheduler to use. Finally, you can choose a random seed if you want to reproduce the results.

### 5.4.3 Training and using the denoising U-Net model

To train the model, we'll adjust model parameters to minimize the difference (measured by mean absolute error) between the predicted noise and the actual noise. We'll use the AdamW optimizer with a learning rate of 0.0002 to update model parameters during training:

```
optim = torch.optim.AdamW(model.parameters(), lr=2e-4)
```

We'll use PyTorch's automatic mixed-precision package `torch.amp` to speed up training, as we did in previous chapters. The following code block trains the denoising U-Net model.

**Listing 5.7 Training the denoising U-Net model**

```
from tqdm import tqdm
from torchvision.utils import save_image, make_grid

scaler = torch.amp.GradScaler("cuda")
for i in range(10):
    pbar = tqdm(data_loader)
    loss_ema = None
    for x, _ in pbar:
        x = x.to(device)
        with torch.amp.autocast("cuda"):
            noise = torch.randn_like(x).to(device)
            timesteps = torch.randint(0,
                                     noise_scheduler.config.num_train_timesteps,
                                     (x.shape[0],), device=device)
            noisy = noise_scheduler.add_noise(x, noise, timesteps)
            noise_pred = model(noisy, timesteps)["sample"]
            loss = torch.nn.functional.l1_loss(noise_pred, noise)
        optim.zero_grad()
        scaler.scale(loss).backward()
        scaler.step(optim)
        scaler.update()
        if loss_ema is None:
            loss_ema = loss.item()
        else:
            loss_ema = 0.9 * loss_ema + 0.1 * loss.item()
        pbar.set_description(f"loss: {loss_ema:.4f}")
    xh = sample(32, model, noise_scheduler)
```

← Trains the model for 10 epochs

← Predicts the amount of noise injected into the images

← Adjusts model parameters to minimize L1 loss

```
grid = make_grid(0.5-xh/2, nrow=8)
save_image(grid, f"files/diffusion_fashion{i}.png")
torch.save(model.state_dict(), "files/diffusion_fashion.pth")
```

← Visually checks the generated images by the model after each epoch

For simplicity, we train the model for 10 epochs. You can use early stopping to determine how many epochs to train, as we did in chapter 3. We iterate through all batches in the training dataset and inject noise into images to obtain noisy images. We then use the U-Net model to predict the amount of noise in these images. We compare the predictions with the actual amount of noise to calculate the L1 loss. We then tweak the model parameters to minimize this loss.

After each epoch, we use the `sample()` function we defined before to produce 32 clean images using the U-Net model. These images are saved on your computer so that you can visually inspect the results throughout training. Figure 5.10 shows the generated images by the model after 1, 3, 5, 7, and 9 epochs of training, respectively.



**Figure 5.10** Clothing-item images generated by the denoising U-Net model after 1, 3, 5, 7, and 9 epochs of training

Figure 5.10 illustrates clothing-item images produced by the denoising U-Net model at different stages of training. The model is trained for a total of 10 epochs, and images are generated at the end of each epoch. The top row shows the images after the first epoch, the second row after three epochs, and so on, with the bottom row displaying

the images after nine epochs. As training progresses, the quality of the generated images consistently improves.

After training, the trained model is saved on your computer. Alternatively, you can download the trained model from my Google drive at <https://mng.bz/Bze2>. Unzip the file after downloading. Let's load up the trained model parameters and generate some images. The output is shown in figure 5.11.

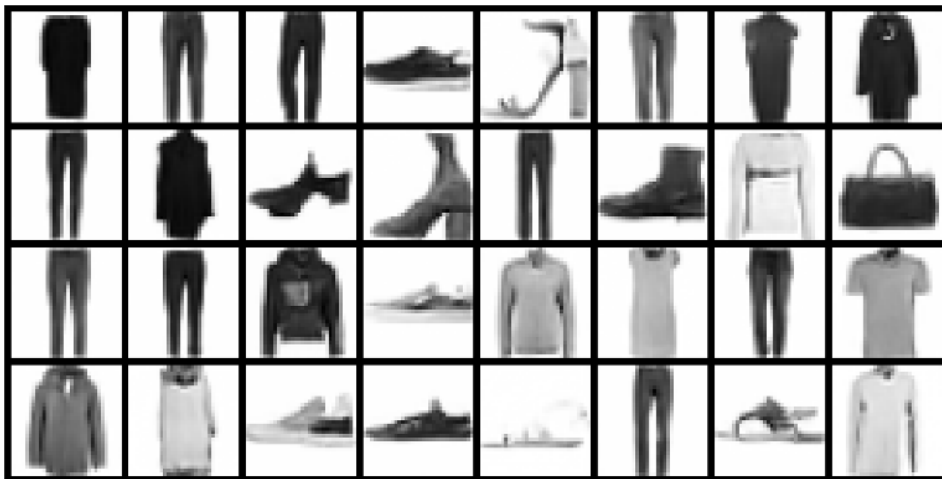
#### Listing 5.8 Visualizing generated images using the trained model

```
torch.manual_seed(42)
model.load_state_dict(torch.load(
    "files/diffusion_fashion.pth",
    map_location=device,
    weights_only=True))
generated_images = sample(32, model, noise_scheduler)
grid = make_grid(0.5-generated_images/2, nrow=8)
save_image(grid, f"files/diffusion_fashion.png")
plt.figure(dpi=100)
plt.imshow(grid.cpu().permute(1,2,0))
plt.axis('off')
plt.tight_layout()
plt.show()
```

← Loads the trained model weights

← Generates 32 images using the sample() function defined before

← Plots the images in a 4 × 8 grid



**Figure 5.11** Generated clothing-item images with a trained denoising U-Net model. We train the model for 10 epochs. We feed random noise to the trained model. The model iteratively removes noise from the noisy images for 1,000 time steps, resulting in what is shown here.

There are 10 different types of clothing items, and the model randomly generates images from one of these 10 types. As you can see, the trained denoising U-Net model generates clothing item images resembling those in the training dataset.

**Exercise 5.3**


Set the PyTorch random state to 0. Use the trained model to generate 16 images, and plot them in a  $2 \times 8$  grid.

If you're wondering whether you can guide the model to generate specific items, such as a sandal, a t-shirt, or a coat, the answer is yes! In the next chapter, you'll learn how to design and train a diffusion model to use conditioning information (e.g., the label of a clothing item) to generate the type of item you're looking for.

**Summary**

- In the forward diffusion process, we start with a sample of clean images. We gradually introduce small amounts of random noise into these clean images. After many time steps of adding noise, the clean images become pure random noise.
- Two commonly used diffusion schedules in the forward process are the linear and cosine diffusion schedules:
  - A linear schedule defines the variance of noise added in each time step as an increasing linear function of time. The linear noise schedule is simple and commonly used due to ease of implementation.
  - The cosine schedule adjusts the noise variance in each time step using a cosine function. This leads to a nonlinear progression where noise increases more gradually at first and then accelerates. The cosine schedule helps maintain finer details in initial diffusion steps, which can enhance model performance and output quality.
- In the reverse diffusion process (also called the denoising process), we train a model to reverse the forward diffusion process. We feed the model with random noise and ask it to produce a noisy image. We iteratively repeat the process for many time steps, and the end result is a clean image.
- We usually use a U-Net as the denoising model. The U-Net has a symmetric shape with a contracting encoder path and an expansive decoder path, connected by a bottleneck layer. In denoising, U-Nets are adapted to remove noise while preserving details. U-Nets outperform simple convolutional networks in denoising tasks due to their efficient capturing of local and global features in images.

# *Control what images to generate in diffusion models*



## ***This chapter covers***

- Creating and training a conditional diffusion model to generate images
- Building a denoising U-Net from scratch
- Coding diffusion processes in a Denoising Diffusion Probabilistic Model
- Injecting labeling information into the U-Net model for controlled generation
- Implementing classifier-free guidance

In the previous chapter, you learned how diffusion models generate images by gradually transforming random noise into clothing-item images, such as coats, bags, and sandals, using a denoising U-Net. However, the model could only generate images randomly from among 10 classes. A natural next question arises: Can we direct the model to create a specific image—a sandal, a t-shirt, or a coat—on demand? This chapter shows you how.

Here, you'll learn conditional diffusion models that let you specify what you want to generate by conditioning on label information. The conditioning isn't just limited

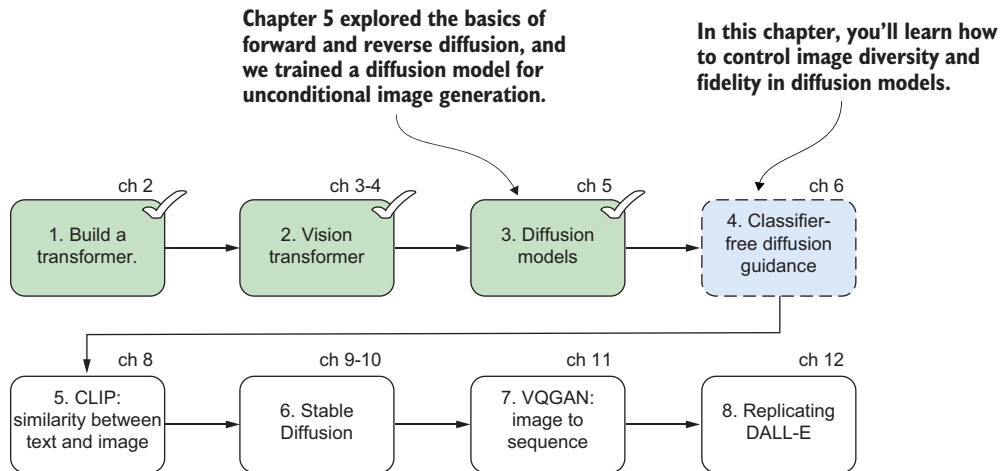


to image classes; later in the book, we'll extend the idea to text-to-image generation, where the conditioning is an open-ended text prompt. Mastering conditioning is therefore essential groundwork for the more advanced generative models we'll tackle in later chapters.

You'll also move from using the U-Net as a “black box” to building your own U-Net architecture from scratch. This hands-on approach provides a deeper understanding of how class labels and time steps are embedded in the network, as well as how these embeddings control the generative process.

A major breakthrough in conditional diffusion is classifier-free guidance (CFG). Earlier methods relied on an external classifier to steer image generation, which added complexity and limited flexibility. CFG removes the need for that extra model by training the diffusion network to handle both conditional (prompted) and unconditional (free) generation. This simplification not only makes the pipeline more efficient but also leads to higher-quality outputs: images can better match prompts while remaining visually coherent. At the same time, CFG offers greater flexibility by letting you control the tradeoff between *fidelity* (accurately following the prompt) and *diversity* (producing varied, creative results), all within a single model.

Figure 6.1 illustrates how this chapter fits into your journey of building a text-to-image generator from scratch. We'll zero in on step 4, using CFG to control image diversity and fidelity in diffusion models.



**Figure 6.1** The eight-step road map for building a text-to-image generator from scratch. This chapter zeroes in on step 4, empowering your diffusion model to generate specific types of images on command using CFG, a breakthrough technique used in state-of-the-art text-to-image generation models.

By the end of this chapter, you'll be able to build and train a diffusion model that can generate specific classes of images upon request. Specifically, you'll learn to inject

conditioning information directly into your U-Net architecture using label embeddings. You'll implement CFG to control the tradeoff between image diversity and adherence to your prompt. This lays the technical foundation for more advanced text-to-image diffusion models (e.g., latent diffusion and Stable Diffusion) discussed in subsequent chapters.

Later in the book, you'll see that CFG isn't just an advanced trick: it's the method powering much of the controllability in state-of-the-art generative models. In fact, the CFG scale you'll experiment with here is the same parameter that allows models such as Stable Diffusion to let users adjust how closely the generated image should follow the input prompt.

## 6.1 *Classifier-free guidance in diffusion models*

In their 2022 paper titled “Classifier-Free Diffusion Guidance,” Jonathan Ho and Tim Salimans introduced a groundbreaking technique that reshapes how diffusion models can generate high-quality samples [1]. They demonstrated that effective guidance during generation can be achieved using a purely generative model without relying on an external classifier. Their approach involves jointly training a diffusion model under both conditional and unconditional settings, striking a balance between the quality and diversity of the generated samples.

In this section, I'll first provide an overview of CFG and its implications for advanced text-to-image models such as latent diffusion models (LDMs) and Stable Diffusion. After that, I'll explain how to implement CFG when generating grayscale clothing-item images.

### 6.1.1 *An overview of classifier-free guidance*

To understand CFG, it's important to compare it with traditional diffusion models. Traditional diffusion models typically rely on external classifiers (e.g., class labels) to guide the image-generation process using a technique known as *classifier guidance*. This process enhances the quality of generated samples by using the classifier to steer the diffusion model's sampling toward desired outcomes, such as specific class labels or prompts.

From the previous chapter, we know that diffusion models generate images by gradually reversing a noising process. Starting from pure noise, the model iteratively refines the sample at each step using a learned denoising process. Without guidance, the generated samples are only influenced by the prior distribution learned during training, which may result in generic outputs. This is why in the previous chapter, the generated image can be any one of the 10 classes of items: a t-shirt, a coat, a sandal, and so on.

If we want to control what item to generate, the traditional way is to use an external classifier, trained separately from the diffusion model. The classifier is used to evaluate intermediate outputs (partially denoised samples) at each step. The classifier predicts class probabilities for the intermediate outputs, providing feedback on how well the sample aligns with the desired class or condition.

In contrast, CFG doesn't rely on an external classifier. The diffusion model itself is trained to conditionally generate images based on class labels or prompts. This approach eliminates the need for a separate classifier and simplifies the pipeline while maintaining control and improving sample quality. Specifically, CFG simplifies this process by embedding the guidance mechanism directly within the generative model itself. This is achieved by training the diffusion model to operate both with and without conditioning information. For example, if the conditioning information is a class label, the model alternates between being trained with and without the label during training. The same principle applies when the conditioning information is a text prompt, which we'll explore in later chapters. This dual training strategy allows the model to learn how to generate samples in both scenarios, either guided by specific conditioning or completely unconstrained.

During inference, CFG is applied by combining the model's conditional and unconditional predictions. This allows fine-grained control over the generated samples, balancing adherence to the conditioning information against the diversity of outputs. The extent of this balance is controlled by a parameter known as the CFG scale. In later chapters, you'll see how this parameter is used in LDMs and Stable Diffusion to optimize results.

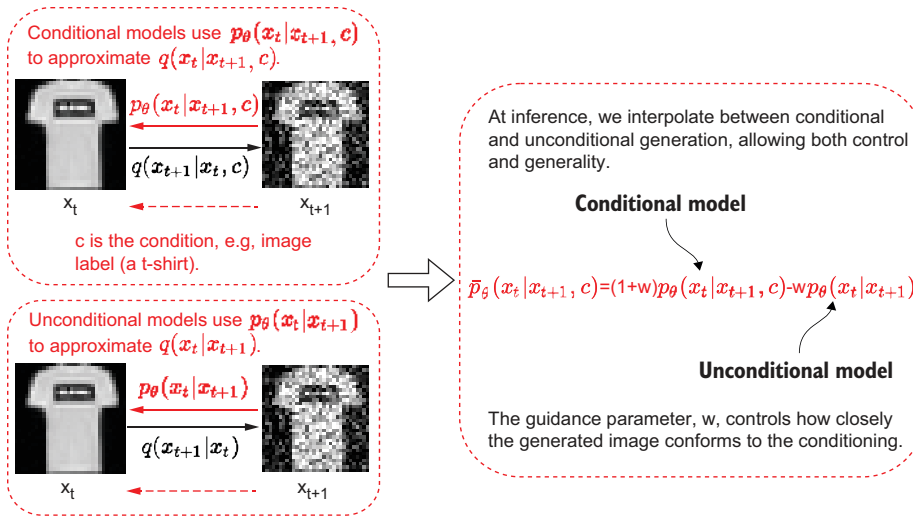
The CFG scale parameter is pivotal in determining how much influence the conditioning information has on the generated output. A higher CFG scale amplifies the impact of the conditioning, be it a class label or a text prompt, resulting in images that closely align with the specified condition. However, this often comes at the expense of diversity, as the model prioritizes strict adherence to the condition. Conversely, a lower CFG scale reduces the conditioning's influence, leading to greater creative diversity but potentially at the cost of misalignment with the specified condition.

### 6.1.2 A blueprint to implement CFG

To make the concept of CFG more relatable, you'll learn to use it to train a diffusion model to generate grayscale clothing-item images. Specifically, you'll create a diffusion model that can act both as a conditional model and an unconditional model. Figure 6.2 is a diagram of how this works.

In chapter 5, we discussed the idea behind the reverse diffusion process in unconditional models. In the forward diffusion process, we add noise to the image in time step  $t$ ,  $x_t$ , to obtain the image in time step  $t + 1$ ,  $x_{t+1}$ , based on a function  $q(x_{t+1}|x_t)$ . However, upon observing  $x_{t+1}$ , we can't obtain  $x_t$  because the function  $q(x_t|x_{t+1})$  is unknown. Therefore, we use the training data to learn a model to approximate the conditional probability  $p_\theta(x_t|x_{t+1})$ , where  $\theta$  represents model parameters.

The logic is similar in conditional diffusion models. We embed the conditional information,  $c$ , in the diffusion model. Our goal is to infer  $x_t$  upon observing  $x_{t+1}$  conditional on the labeling information  $c$ . Because  $q(x_t|x_{t+1}, c)$  is unknown, we use the training data to learn a model to approximate  $p_\theta(x_t|x_{t+1}, c)$ .



**Figure 6.2** A diagram of CFG. In CFG, we jointly train a conditional (top left) and unconditional (bottom left) diffusion model to attain a tradeoff between sample quality and diversity. By embedding the conditioning information directly in the diffusion model, we can use the model as either a conditional model or an unconditional model. At inference, CFG is applied by interpolating between the model's conditional and unconditional predictions. The guidance parameter,  $w$ , controls how closely the generated image adheres to the conditioning information.

We'll build a diffusion model that can function both as a conditional and unconditional model. The diffusion model itself is trained to conditionally generate images based on class labels or prompts. Specifically, CFG simplifies this process by embedding the guidance mechanism directly within the generative model itself. This is achieved by training the diffusion model to operate both with and without conditioning information. For example, if the conditioning information is a class label, the model alternates between being trained with and without the label during training. This dual training strategy allows the model to learn how to generate samples in both scenarios, either guided by specific conditioning or completely unconstrained.

During inference, CFG is applied by interpolating between the model's conditional and unconditional predictions. Specifically, to generate an image with label  $c$ , we use the conditional model to generate the conditional prediction  $p_\theta(x_t|x_{t+1}, c)$ . At the same time, we also use the unconditional model to generate the unconditional prediction  $p_\theta(x_t|x_{t+1})$ . The interpolated output is a weighted sum of the preceding two predictions, like this:

$$\bar{p}_\theta(x_t|x_{t+1}, c) = (1+w)p_\theta(x_t|x_{t+1}, c) - wp_\theta(x_t|x_{t+1}) \quad (6.1)$$

The guidance parameter,  $w$ , controls how closely the final output adheres to the conditioning information. In state-of-the-art text-to-image generation models, the guidance

parameter is the CFG scale. In later chapters, you'll see how this parameter is used in LDMs and Stable Diffusion to optimize results.

Later, you'll experiment with the guidance parameter,  $w$ , to control the characteristics of the generated images. Setting this parameter to a high value biases the model toward generating outputs that closely match specific labeling classes or conditions. While this ensures high fidelity to the target condition, it may lead to less diverse outputs. On the other hand, using a lower guidance parameter encourages the generation of more diverse and creative samples, albeit with a potential decrease in fidelity to the specified condition. Note that in the special case when the value of  $w$  is 0, the output is based on conditional prediction only.

The ability to dynamically adjust the balance between fidelity and diversity is one of the core strengths of CFG. It provides users with flexible control over the image-generation process, empowering them to tailor outputs to specific goals. This approach eliminates the dependency on external classifiers, reduces computational overhead, and integrates seamlessly into modern diffusion-based frameworks such as LDMs and Stable Diffusion.

As we dive deeper into the mechanics of this technique, we'll explore its implementation and applications in greater detail, including how the CFG scale can be fine-tuned to achieve desired results. By mastering CFG, you'll gain valuable insights into one of the most impactful advancements in generative modeling, enabling you to create high-quality images that are also highly customizable.

## 6.2 Different components of a denoising U-Net model

In the next section, you'll learn to create a denoising U-Net model from scratch. However, before we implement the U-Net, I'll provide you with a high-level overview of the structure of the model and its components. You'll learn how convolutional and max pooling layers can extract feature maps from images and downsample these images. You'll also learn how transposed convolutional layers upsample and reconstruct feature maps in images.

Furthermore, you'll learn how to incorporate labeling and time step information and then embed them in the U-Net model so that the U-Net model can denoise conditional on both the time step and the label of the clothing item.

**NOTE** The Python programs for this chapter can be accessed on the book's GitHub repository (<https://github.com/markhliu/txt2img>) and are also available as a Google Colab notebook (<https://mng.bz/dWzw>) for interactive exploration.

The Python code in this chapter is adapted from the excellent GitHub project by Tim Pearce (<https://mng.bz/rZoy>). For brevity, we use a local module to store various functions and classes we use. Download the file `cfr_util.py` from the book's GitHub repository (<https://github.com/markhliu/txt2img>), and place it in the `/utils/` folder on your computer (or simply clone the repository to your computer). If you're working in

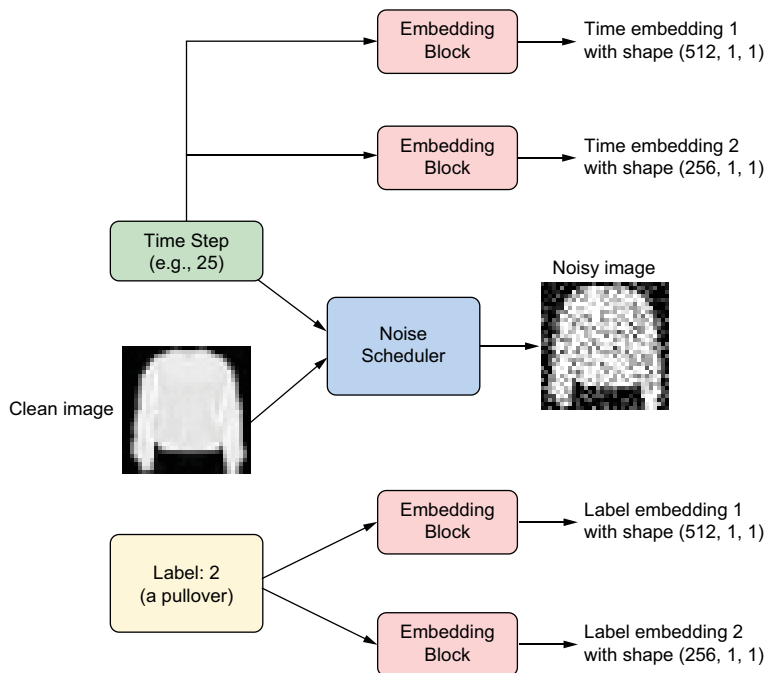
Google Colab, you can quickly set up the repository and add it to your working directory by running

```
!git clone https://github.com/markhliu/txt2img
import sys
sys.path.append("/content/txt2img")
```

### 6.2.1 Time step embedding and label embedding

When we train the denoising U-Net, we add noise to clean images in the training set to create noisy images. We then present the noisy images, along with the time steps the noisy images are in and the class labels of the images, to the U-Net. We train the U-Net to predict the noise in the noisy images, conditional on the class labels.

We'll embed the time steps and the labels when creating the U-Net so that it can effectively understand these pieces of information when predicting the noise in the noisy images. Figure 6.3 illustrates how to embed time steps and the class labels of the clothing-item images in several embedding blocks of the U-Net model. The parameters in these embedding blocks are first randomly initialized and then adjusted during training. The figure shows that we use a noise schedule to generate a noisy image of a clean pullover image from the training set. In this case, the time step is 25, and the class



**Figure 6.3** How to embed the time steps and the class labels in the denoising U-Net model

label is 2 (any other time step or class label will work; here we use 25 and 2 as concrete examples). We generate two time embeddings for later use, with dimensions (512, 1, 1) and (256, 1, 1), respectively. Similarly, we use embedding blocks in the U-Net model to create two label embeddings with dimensions (512, 1, 1) and (256, 1, 1), respectively. The time embeddings and label embeddings will be used in the expansive path (i.e., decoder) of the U-Net.

In the local module file `cfr_util.py` you just downloaded, we define the following embedding block to embed both time steps and the labeling information.

#### Listing 6.1 Defining an embedding block

```
from torch import nn
class EmbedLayer(nn.Module):
    def __init__(self, input_dim, emb_dim):
        super().__init__()
        self.input_dim = input_dim
        layers = [nn.Linear(input_dim, emb_dim),
                  nn.GELU(),
                  nn.Linear(emb_dim, emb_dim),]
        self.model = nn.Sequential(*layers)
    def forward(self, x):
        x = x.view(-1, self.input_dim)
        return self.model(x)
```

← The input and embedding dimensions are `input_dim` and `emb_dim`, respectively.

← The input goes through the two linear layers, with a GELU activation in between.

The `EmbedLayer()` class contains two linear layers, with a Gaussian error linear unit (GELU) activation in between. To illustrate exactly how the embedding block works, let's assume that the time step is 25, as shown earlier in figure 6.3. We'll instantiate two time embedding blocks first, with an input dimension of 1 (because the time step is a one-dimensional number) and embedding dimensions of 512 and 256, respectively. We'll then use the time step, 25, as the input in the two embedding blocks. Finally, we print out the dimensions of the two outputs.

#### Listing 6.2 Creating two time embeddings

```
import torch
from utils.cfr_util import EmbedLayer

timeembed1=EmbedLayer(1, 512)
timeembed2=EmbedLayer(1, 256)

timesteps=torch.tensor([25]).long()
t=timesteps/1000 #C
temb1 = timeembed1(t).view(-1, 512, 1, 1)
temb2 = timeembed2(t).view(-1, 256, 1, 1)
print("the shape of the first time embedding is",
      temb1.shape)
print("the shape of the second time embedding is",
      temb2.shape)
```

← Imports the `EmbedLayer()` class from the local module

← Instantiates two time embedding blocks, with embedding dimensions 512 and 256

← Uses the relative time step as the input

← Prints out the shape of the two time embeddings

The output from listing 6.2 code block is

```
the shape of the first time embedding is torch.Size([1, 512, 1, 1])
the shape of the second time embedding is torch.Size([1, 256, 1, 1])
```

The shape of the first time embedding is (1, 512, 1, 1). The first value in (1, 512, 1, 1) means there's one sample in the batch. The last three values in (1, 512, 1, 1) indicate that the time step is converted into an embedding tensor with a shape of (512, 1, 1). The shape of the second time embedding is (1, 256, 1, 1), meaning the time step is converted into an embedding tensor with a shape of (256, 1, 1).

### Exercise 6.1

Suppose the time step value is 42. Modify listing 6.2 to create two time embeddings with embedding dimensions 512 and 256, respectively. Then, print out the shape of the two time embeddings.

To embed the labeling information, we first convert the label, which can take a value between 0 and 9, into a 10-value one-hot tensor. We mask the labeling information with a 10% probability so that we train an unconditional model with a 10% probability and a conditional model with a 90% probability. We'll instantiate two label embedding blocks and pass the one-hot label tensor through the embedding blocks to obtain the label embeddings, as shown in the following code listing.

### Listing 6.3 Generating label embeddings

```
n_classes=10
contextembed1=EmbedLayer(n_classes, 512)
contextembed2=EmbedLayer(n_classes, 256)
label=torch.tensor([2, 9, 0]).long()
context_mask = torch.bernoulli(
    torch.zeros_like(label)+0.1)
onehot = torch.nn.functional.one_hot(label,
    num_classes=n_classes).type(torch.float)
context_mask = context_mask[:, None]
context_mask = context_mask.repeat(1,n_classes)
context_mask = (-1*(1-context_mask))
c = onehot * context_mask
cemb1 = contextembed1(c).view(-1, 512, 1, 1)
cemb2 = contextembed2(c).view(-1, 256, 1, 1)
print("the shape of the first label embedding is", cemb1.shape)
print("the shape of the second label embedding is", cemb2.shape)
```

Instantiates two label embedding blocks

Creates a batch of three labels

Generates a mask to hide the label with a 10% probability

Converts each label to a one-hot tensor with ten values

The output is shown here:

```
the shape of the first label embedding is torch.Size([3, 512, 1, 1])
the shape of the second label embedding is torch.Size([3, 256, 1, 1])
```



Instead of using just one noisy image in the batch, we assume there are three images, so we put three values—2, 9, and 0—in the label tensor in the preceding code listing. Further, we convert the labels into 10-value one-hot tensors because there are 10 different types of clothing items in the training dataset. We instantiate two label embedding blocks, with an input dimension of 10 (because labels are represented by 10-value one-hot tensors now) and embedding dimensions of 512 and 256, respectively. Masking the labeling information is our way of implementing CFG discussed in the first section. This way, we train an unconditional model and a conditional model simultaneously.

The shapes of the two label embeddings are (3, 512, 1, 1) and (3, 256, 1, 1), respectively. The first number in the shape, 3, means that there are three labels in the batch. The last three values in the shape indicate that the labels are converted into embedding tensors with a shape of (512, 1, 1) by the first embedding block, and they are converted into embedding tensors with a shape of (256, 1, 1) by the second embedding block.

### Exercise 6.2

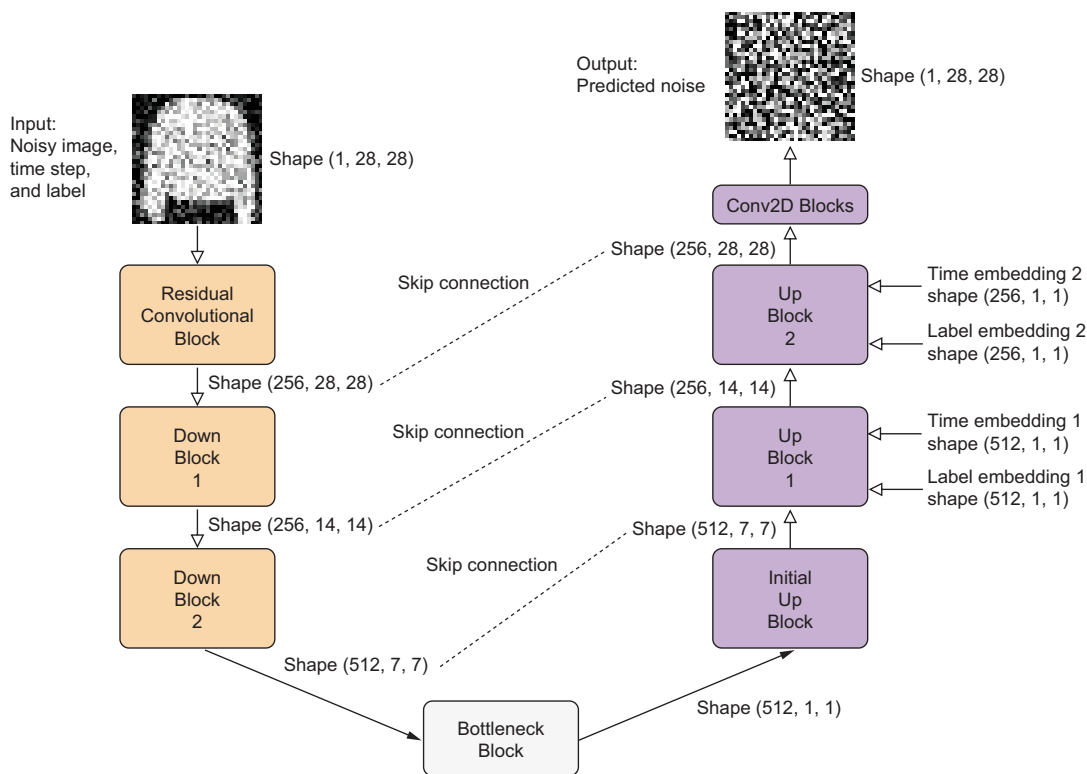
Suppose there are four labels in the batch with values 1, 3, 5, and 7, respectively. Modify listing 6.3 to create two label embeddings. Then, print out the shape of the two label embeddings.

## 6.2.2 The U-Net denoising model architecture

A U-Net model consists of three main components: a contracting path (encoder), an expansive path (decoder), and a bottleneck path connecting the two. U-Nets are specifically designed for tasks such as image denoising, where preserving both fine-grained details and global context is critical. Compared to simple convolutional neural networks (CNNs), U-Nets excel in these tasks due to their ability to effectively capture and combine local and global features.

The U-Net is trained to predict the noise present in a noisy image. Because the noisy image is a weighted combination of the original clean image and noise, accurately predicting the noise allows us to reconstruct the clean image. This makes U-Nets highly effective in denoising applications.

Figure 6.4 provides a detailed illustration of the U-Net architecture. The U-Net processes noisy images alongside their corresponding time steps and labels as inputs, predicting the noise injected into the images. The U-Net takes noisy images, time steps, and labels as inputs and predicts the noise in the noisy images. The architecture consists of three main parts: a contracting path (encoder), an expansive path (decoder), and a bottleneck connecting the two. The contracting path extracts features and downsamples the images through convolutional and pooling layers, progressively capturing higher levels of abstraction. The expansive path reconstructs the feature maps by upsampling the images and uses skip connections to merge corresponding features from the contracting path. Finally, the bottleneck serves as a bridge between the encoder and decoder, processing the most abstracted features before reconstruction.



**Figure 6.4** The components of the denoising U-Net model

The left side of figure 6.4 represents the contracting path, which is designed to learn feature representations and progressively downsample the input image. This path consists of several blocks of convolutional layers followed by pooling layers. Convolutional layers extract feature maps by detecting patterns such as edges, textures, or shapes at various levels of abstraction. Pooling layers reduce the spatial dimensions of the feature maps, making the model computationally efficient while capturing more abstract representations. In the next subsection, we'll explain in detail how max pooling specifically achieves this downsampling.

The bottleneck, shown at the bottom of figure 6.4, serves as a bridge between the contracting and expansive paths. It processes the most abstracted features extracted by the contracting path and prepares them for reconstruction in the expansive path. The bottleneck contains layers with the smallest spatial dimensions and the most condensed information about the image.

The right side of figure 6.4 shows the expansive path, which upscales the downsampled feature maps and reconstructs the image. This path consists of transposed convolutional layers and additional convolutional layers. Transposed convolutional layers

upsample the feature maps, effectively filling in gaps and increasing spatial resolution. These layers reconstruct the image details step-by-step, and I'll explain in detail how this is achieved in the following subsection.

To enhance the reconstruction process, skip connections are used to combine feature maps from the contracting path with those in the expansive path. This ensures that fine-grained details lost during downsampling are retained and incorporated back into the upsampled image. Skip connections are represented by dashed lines in figure 6.4. They play a vital role in preserving spatial information by concatenating feature maps from corresponding levels of the contracting and expansive paths. These feature maps have the same spatial dimensions but are processed differently based on their respective paths.

For example, the dashed line at the bottom of figure 6.4 illustrates that the output from the second down block in the contracting path (with a shape of  $512 \times 7 \times 7$ ) is concatenated with the input to the first up block in the expansive path (also with a shape of  $512 \times 7 \times 7$ ). This concatenation results in an input to the first up block with a shape of  $1024 \times 7 \times 7$ . By combining low-level details from the contracting path with high-level abstract features in the expansive path, the U-Net effectively balances precision and abstraction, leading to superior denoising performance.

In summary, the U-Net model uses three components—contracting path, bottleneck, and expansive path—along with skip connections, to predict noise and reconstruct clean images. This architecture, illustrated in figure 6.4, highlights the power of U-Nets in handling complex denoising tasks while preserving crucial image details.

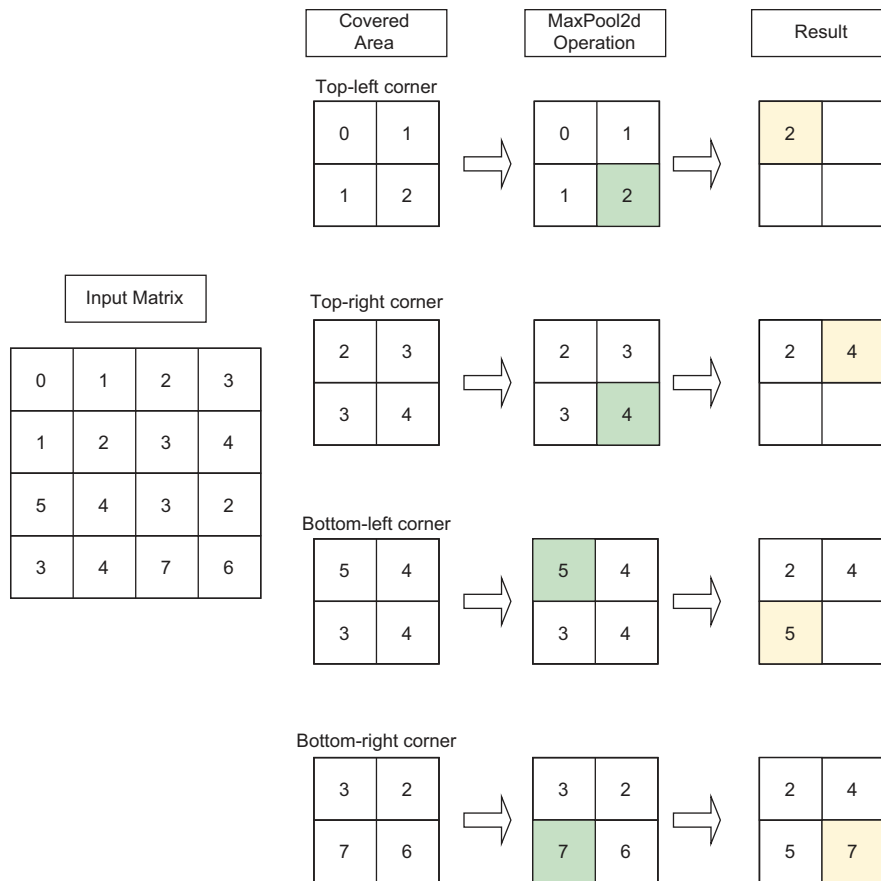
### 6.2.3 Down blocks and up blocks in the U-Net

The contracting path in the U-Net consists of multiple down blocks. Each down block uses multiple convolutional layers and pooling layers to downsample the image, extracting and encoding features at different levels of abstraction. The down blocks are designed to recognize patterns and features that are relevant for denoising.

Convolutional layers use filters to extract spatial patterns on the input data. A convolutional layer is capable of automatically detecting a large number of patterns. Convolutional operations involve applying a filter to an input image to produce a feature map (see chapter 4 of my book *Learn Generative AI with PyTorch* [Manning, 2024]) on how convolutional layers work.

Max pooling is a technique commonly used in CNNs to downsample images or feature maps. This helps reduce the spatial dimensions (height and width) while retaining the most important features in an efficient way. The input image is divided into smaller subregions called windows or patches. The size of the window is specified by a kernel size (e.g.,  $2 \times 2$  or  $3 \times 3$ ). The *stride* determines how much the window moves between operations. For example, a stride of 2 means the filter moves on the image 2 pixels at a time horizontally or vertically. Within each patch, the maximum value of the pixels in that window is selected. This value represented the patch in the downsampled output.

Figure 6.5 shows a numerical example of how max pooling works. The input image is shown on the left. We use a kernel size of  $2 \times 2$ , with a stride of 2. The second column



**Figure 6.5** A numerical example of how max pooling works, with a kernel size of  $2 \times 2$  and a stride equal to 2

shows the four covered areas ( $2 \times 2$  matrices). Max pooling (the third column) involves selecting the largest value in the covered area and outputting it as the result (the last column).

To gain a deep understanding of exactly how max pooling works, let's implement the max pooling operation in PyTorch in parallel so that you can verify the numbers shown in figure 6.5. First, let's create a PyTorch tensor to represent the input image in the figure:

```
import torch

img = torch.Tensor([[0,1,2,3],
                    [1,2,3,4],
                    [5,4,3,2],
                    [3,4,7,6]]).reshape(1,1,4,4)
```

The four values in the shape of the image, (1, 1, 4, 4), are the number of images in the batch, number of color channels, image height, and image width, respectively.

The image is reshaped so that it has a dimension of (1, 1, 4, 4), indicating that there is just one observation in the batch, and the image has one color channel. The height and the width of the image are both 4 pixels.

Let's create a max pooling layer with a kernel size of  $2 \times 2$  and a stride of 2 in PyTorch. We then apply the max pooling layer on the image and print out the outcome:

```
maxpool = torch.nn.MaxPool2d(2)
out=maxpool(img)
print(out)
```

Creates a 2D max pooling layer, with a kernel size of 2 (and the default stride value is equal to the kernel size)

Applies the max pooling layer on the image

Prints out the output

We first create a 2D max pooling layer in PyTorch by using the command `torch.nn.MaxPool2d(2)`. The argument 2 indicates that the kernel size is  $2 \times 2$ . The default stride value is equal to the kernel size, so we have a stride value of 2 here. (See <https://mng.bz/V9nG> for the official documentation on the 2D max pooling layer.) The output from the preceding code block is

```
tensor([[[[2., 4.],
          [5., 7.]]]])
```

The output has a shape of (1, 1, 2, 2) with four values in it: 2, 4, 5, and 7. These numbers match those in figure 6.5.

The input image is a  $4 \times 4$  matrix, and the filter is a  $2 \times 2$  matrix with a stride of 2. When the filter scans over the image, it first covers the 4 pixels in the top-left corner of the image, which have values `[[0, 1], [1, 2]]`, as shown in the first row, second column, in figure 6.5. The 2D max pooling operation selects the maximum value in the window, which is 2. This explains why the top-left corner of the output has a value of 2. Similarly, when the filter is applied to the top-right corner of the image, the covered area is `[[2, 3], [3, 4]]`. The output is therefore 4.

### Exercise 6.3

Suppose you create an image using the following lines of code:

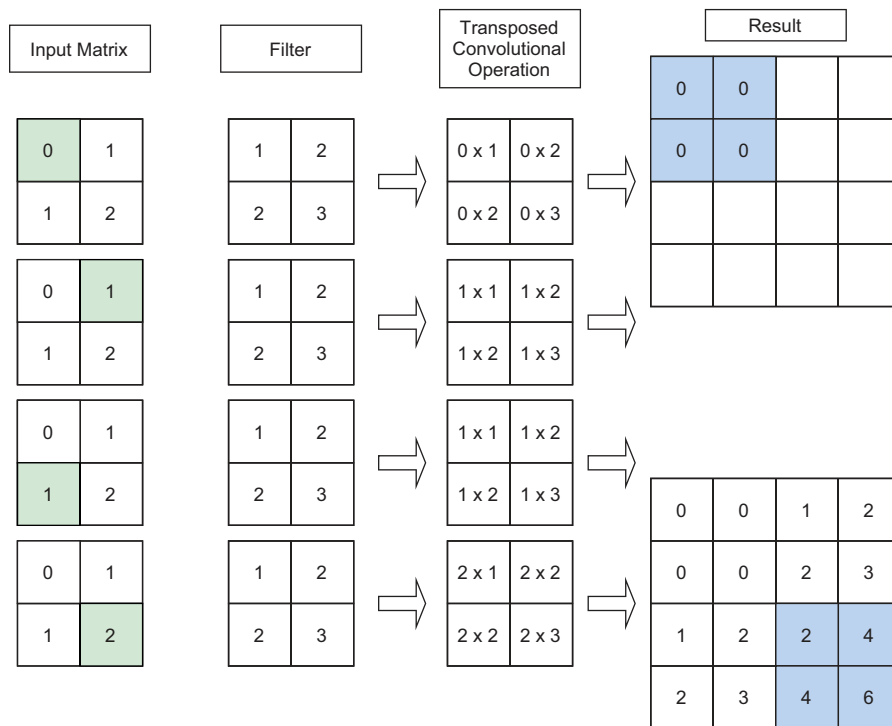
```
img2 = torch.Tensor([[8,1,9,3],
                    [1,2,3,4],
                    [5,6,3,2],
                    [3,4,7,8]]).reshape(1,1,4,4)
```

You apply the same 2D max pooling layer on the image (assume the kernel size is  $2 \times 2$  and the stride is 2). What is the output?

The up block uses the 2D transposed convolutional layer to upsample features in the input. Contrary to convolutional layers, transposed convolutional layers upsample and

fill in gaps in an image to generate features and increase resolution using kernels (i.e., filters). The output is usually larger than the input in a transposed convolutional layer. Therefore, transposed convolutional layers are essential tools when it comes to generating high-resolution images.

Strides can be used in transposed convolution layers to control the amount of upsampling. The greater the value of the stride, the more upsampling the transposed convolution layer has on the input data. To see exactly how 2D transposed convolutional operations work, consider the following simple example and figure. Suppose you have a very small  $2 \times 2$  input image, as shown in the left column in figure 6.6.



**Figure 6.6** A numerical example of how transposed convolutional operations work

The input image has the following values in it:

```
image = torch.Tensor([[0,1],
                      [1,2]]).reshape(1,1,2,2)
```

You want to upsample the image so that it has higher resolutions. You can create a 2D transposed convolutional layer in PyTorch:

```

transconv=torch.nn.ConvTranspose2d(in_channels=1,
                                   out_channels=1,
                                   kernel_size=2,
                                   stride=2)
sd=transconv.state_dict()
weights={'weight':torch.tensor([[[[1,2],
                                   [2,3]]]]), 'bias':torch.tensor([0])}
for k in sd:
    with torch.no_grad():
        sd[k].copy_(weights[k])

```

← A transposed convolutional layer with one input channel, one output channel, a kernel size of 2, and a stride of 2

← Replaces the weights and bias in the transposed convolutional layer with handpicked values

This 2D transposed convolutional layer has one input channel, one output channel, a kernel size of  $2 \times 2$ , and a stride of 2. The  $2 \times 2$  filter is shown in the second column in figure 6.6. We replaced the randomly initialized weights and the bias in the layer with our handpicked whole numbers so that it's easy to follow the calculations. The `state_dict()` method in the preceding code listing returns the parameters in a deep neural network.

When the transposed convolutional layer is applied to the  $2 \times 2$  image we mentioned earlier, what is the output? Let's find out:

```

transoutput = transconv(image)
print(transoutput)

```

The output is

```

tensor([[[[0., 0., 1., 2.],
          [0., 0., 2., 3.],
          [1., 2., 2., 4.],
          [2., 3., 4., 6.]]]], grad_fn=<ConvolutionBackward0>)

```

The output has a shape of (1, 1, 4, 4), meaning we've upsampled a  $2 \times 2$  image to a  $4 \times 4$  image. How does the transposed convolutional layer generate the preceding output through the filter? We'll explain in detail next.

The image is a  $2 \times 2$  matrix, and the filter is also a  $2 \times 2$  matrix. When the filter is applied to the image, each element in the image multiplies with the filter and goes to the output. The top-left value in the image is 0, and we multiply it with the values in the filter,  $\begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}$ , which leads to the four values in the top-left block of the output matrix `transoutput`, with values  $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$ , as shown at the top-right corner in figure 6.6. Similarly, the bottom-left value in the image is 1, which we multiply by the values in the filter,  $\begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}$ , leading to the four values in the bottom-left block of the output matrix `transoutput`,  $\begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}$ .

### Exercise 6.4

If an image has values  $\begin{bmatrix} 3 & 2 \\ 5 & 4 \end{bmatrix}$  in it, what is the output after you apply the 2D transposed convolutional layer `transconv` to the image? Assume `transconv` has values  $\begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}$  in it. Assume a kernel size of 2 and a stride size of 2.

The 2D transposed convolutional layer in the preceding code block has a stride of 2 and a kernel size of 2. When the transposed convolutional layer is applied to input, each element in the input tensor multiplies with the filter and goes to the output. This effectively doubles the height and width of the input image.

### 6.3 Building and training the denoising U-Net model

You'll learn to create a denoising U-Net model from scratch in this section. Along the way, we'll implement label embedding and time embedding so that the U-Net model can denoise conditional on both time steps and the label of the clothing item. We'll also discuss how to code the CFG and how to choose the strength of the generative guidance.

We'll build a Denoising Diffusion Probabilistic Model (DDPM) from scratch. We'll use the Fashion MNIST dataset to train the model. Specifically, we'll use the noise scheduler in DDPM to add noise to clean images in the training dataset to create noisy images. The U-Net model is used to predict the noise in the noisy images. We use the DDPM to calculate the mean squared error between the actual noise and the predicted noise. We adjust model parameters to minimize this error.

#### 6.3.1 Building the denoising U-Net

The contracting path of the U-Net consists of multiple down blocks. In the local module file `cfr_util.py` you downloaded in section 6.2, we define down blocks in the `UnetDown()` class:

```
class UnetDown(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        layers = [ResidualConvBlock(in_channels,
                                    out_channels), nn.MaxPool2d(2)]
        self.model = nn.Sequential(*layers)
    def forward(self, x):
        return self.model(x)
```

← The down block consists of a residual convolutional block and a 2D max pooling layer.

The down block uses the 2D max pooling layer to reduce the size of the input by half in both image width and image height. Specifically, the `nn.MaxPool2d(2)` class in PyTorch selects the maximum pixel value in a  $2 \times 2$  window, as we explained in the previous section. Each down block also uses residual convolutional blocks, which are defined in the `ResidualConvBlock()` class in the local module `cfr_util.py`.

The expansive path of the U-Net consists of multiple up blocks. In the local module, we define the `UnetUp()` class to construct an up block:

```
class UnetUp(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        layers = [nn.ConvTranspose2d(in_channels,
                                    out_channels, 2, 2),
                  ResidualConvBlock(out_channels, out_channels),
```



```

        ResidualConvBlock(out_channels,
                           out_channels),]
    self.model = nn.Sequential(*layers)
def forward(self, x, skip):
    x = torch.cat((x, skip), 1)
    x = self.model(x)
    return x

```

← The up block consists of two residual convolutional blocks and a 2D transposed convolutional layer.

The up block uses the 2D transposed convolutional layer to upsample features in the input. As explained in the previous section, transposed convolutional layers fill in gaps in an image to generate features and increase resolution. The output is usually larger than the input in a transposed convolutional layer.

In the local module, we define a UNet() class that can be either a conditional denoising model or an unconditional denoising model. A *conditional denoising model* generates a clothing-item image of a particular class, say, class 4 (a coat) or class 2 (a pullover). In contrast, an *unconditional denoising model* randomly generates a clothing-item image that belongs to any of the 10 classes. Part of the UNet() class is shown in the following code listing.

#### Listing 6.4 Defining the denoising U-Net model

```

class Unet(nn.Module):
    ""
    def forward(self, x, c, t, context_mask):
        x = self.init_conv(x)
        down1 = self.down1(x)
        down2 = self.down2(down1)
        hiddenvec = self.to_vec(down2)
        c=nn.functional.one_hot(c,
                                num_classes=self.n_classes).type(torch.float)
        context_mask = context_mask[:, None]
        context_mask = context_mask.repeat(1,self.n_classes)
        context_mask = (-1*(1-context_mask))
        c = c * context_mask
        cemb1 = self.contextembed1(c).view(-1, self.n_feat * 2, 1, 1)
        temb1 = self.timeembed1(t).view(-1, self.n_feat * 2, 1, 1)
        cemb2 = self.contextembed2(c).view(-1, self.n_feat, 1, 1)
        temb2 = self.timeembed2(t).view(-1,
                                         self.n_feat, 1, 1)
        up1 = self.up0(hiddenvec)
        up2 = self.up1(cemb1*up1+ temb1, down2)
        up3 = self.up2(cemb2*up2+ temb2, down1)
        out = self.out(torch.cat((up3, x), 1))
        return out

```

← The input images first go through multiple down blocks and then the bottleneck.

← The labels are masked with a 10% probability, so the model acts both as an unconditional model and a conditional model.

← Label and time embeddings are used in up blocks.

← Skip connections are used to merge features from contracting and expansive paths.

The UNet() class takes noisy images, x, labels, c, and time steps, t, as inputs to predict the noise in the images. It also takes context\_mask as an argument. If we set context\_mask to 1, the labeling information is masked, and the model is therefore an unconditional model. On the other hand, if context\_mask=0, the labeling information is used,

and the generation is conditional on the label. Later, when training the model, we'll set `context_mask` to 1 with a 10% probability to train an unconditional model. In the remaining 90% of the time, `context_mask=0`, and we train a conditional model.

The input image first goes through multiple down blocks and then the bottleneck block. The output from the bottleneck goes through multiple up blocks. We create two label embeddings and two time embeddings to be used in the two up blocks. In addition, skip connections are used to merge corresponding features from the contracting and expansive path.

### 6.3.2 The Denoising Diffusion Probabilistic Model

Now that we have a denoising U-Net model, we'll add noise to clean images to generate noisy images. These noisy images are then used to train the denoising U-Net model. In the local module, we define the following `DDPM()` class to create the DDPM.

**Listing 6.5** Creating the DDPM

```
device="cuda" if torch.cuda.is_available() else "cpu"
class DDPM(nn.Module):
    def __init__(self, model, n_T, device=device, drop_prob=0.1):
        super().__init__()
        self.model = model.to(device)
        for k, v in noise_scheduler(n_T).items():
            self.register_buffer(k, v)
        self.n_T = n_T
        self.device = device
        self.drop_prob = drop_prob
        self.loss_mse = nn.MSELoss()
    def forward(self, x, c):
        _ts=torch.randint(1,self.n_T+1,
            (x.shape[0],)).to(self.device)
        noise = torch.randn_like(x)
        x_t = (self.sqrtab[_ts, None, None, None] * x
            + self.sqrtmab[_ts,
                None, None, None] * noise)
        context_mask = torch.bernoulli(torch.zeros_like(c)+
            self.drop_prob).to(self.device)
        return self.loss_mse(noise,
            self.model(x_t, c, _ts / self.n_T,
                context_mask))
```

Randomly generates time steps between 1 and 1,000

Generates noisy images as the weighted sum of clean images and noise

The output is the mean squared error between the actual noise and the predicted noise.

The `DDPM()` class takes in two arguments, `x` and `c`, where `x` is a batch of clean images, and `c` is a tensor with the corresponding class labels. The `DDPM()` class randomly generates a time step for each image and injects noise into these images. We then train the U-Net denoising model we created earlier to predict the noise in the noisy images. The `DDPM()` class returns the mean squared error between the predicted noise and the ground truth (the actual noise injected into the images). We also define a `sample()` function to generate images using the trained model.

**Listing 6.6** Defining a `sample()` function to generate images

```

import numpy as np

@torch.no_grad()
def sample(ddpm, model, n_sample, size, device,
          guide_w = 0.0, step_size=1):
    x_i = torch.randn(n_sample, *size).to(device)
    c_i = torch.arange(0,10).to(device)
    c_i = c_i.repeat(int(n_sample/c_i.shape[0]))
    context_mask = torch.zeros_like(c_i).to(device)
    c_i = c_i.repeat(2)
    context_mask = context_mask.repeat(2)
    context_mask[n_sample:] = 1.
    x_i_store = []
    for i in range(ddpm.n_T, 0, -step_size):
        t_is = torch.tensor([i / ddpm.n_T]).to(device)
        t_is = t_is.repeat(n_sample,1,1,1)
        x_i = x_i.repeat(2,1,1,1)
        t_is = t_is.repeat(2,1,1,1)
        z = torch.randn(n_sample,*size).to(device) if i>1 else 0
        eps = model(x_i, c_i, t_is, context_mask)
        eps1 = eps[:n_sample]
        eps2 = eps[n_sample:]
        eps = (1+guide_w)*eps1 - guide_w*eps2
        x_i = x_i[:n_sample]
        x_i = (ddpm.oneover_sqrt_alpha_bar[i] * (x_i -
            eps * ddpm.mab_over_sqrt_alpha_bar[i])
            + ddpm.sqrt_beta_bar_t[i] * z)
        if i%20==0 or i==ddpm.n_T or i<8:
            x_i_store.append(x_i.detach().cpu().numpy())
    x_i_store = np.array(x_i_store)
    return x_i, x_i_store

```

The class labels have values 0 to 9, so we generate 10 different types of images.

The first half of the mask is turned off as 0s, and the second half is turned on as 1s.

Generates a set of images using the conditional model

Generates a set of images using the unconditional model

The final output is a weighted sum of conditional and unconditional generation.

The `sample()` function follows the guidance sampling scheme described in the 2022 paper by Ho and Salimans [1]. We create class labels with values 0 to 9 so we can generate all 10 types of clothing-item images. Values in the first half of the tensor `context_mask` are turned off as 0s, and we use them for conditional generation. That is, clothing-item images are generated based on specific labels, ranging from 0 to 9. The values in the second half of the tensor `context_mask` are turned on as 1s. Therefore, clothing labels are masked. Clothing-item images are generated unconditionally and can be any of the 10 types. The output is a mix of conditional and unconditional generation. The parameter `guide_w` controls the strength of the label guidance in image generation.

### 6.3.3 Training the diffusion model

Next, we'll train the CFG diffusion model. We'll first preprocess the training data and then discuss the training loop. As in chapter 5, we use grayscale clothing-item images as our training data:

```

from torchvision import transforms, datasets

torch.manual_seed(42)
tf = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Lambda(lambda x: 2*(x-0.5)),])
dataset = datasets.FashionMNIST(
    r".",
    train=True,
    download=True,
    transform=tf,)

```

Converts raw image data into PyTorch tensors with values between -1 and 1

Downloads the training dataset using the TorchVision library

The transforms package in TorchVision creates a series of transformations to preprocess images. The images have a size of (1, 28, 28). The `ToTensor()` class converts image data into PyTorch tensors, with values in the range of 0.0 and 1.0. We normalize the input data to the range [-1.0, 1.0] for faster convergence during training. You can then use the datasets package in TorchVision to download the dataset to a folder on your computer and perform the transformation.

There are 10 different types of clothing items. The labels in the dataset are numbered from 0 to 9. The list `text_labels` contains the 10 text labels corresponding to the numerical labels 0 to 9:

```

text_labels=['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']

```

We put the training data in batches, as follows, and our training data is ready:

```

from torch.utils.data import DataLoader

dataloader = DataLoader(dataset, batch_size=256, shuffle=True)

```

We create a denoising U-Net model by instantiating the `UNet()` class we defined earlier. We also create a DDPM by instantiating the `DDPM()` class.

#### Listing 6.7 Creating the U-Net and DDPM

```

from utils.cfr_util import Unet, DDPM

n_epoch = 20
n_T = 1000
device="cuda" if torch.cuda.is_available() else "cpu"
n_classes = 10
n_feat = 256
lr=0.00001
model=Unet(in_channels=1,
           n_feat=n_feat,
           n_classes=n_classes)
ddpm=DDPM(model, n_T=n_T).to(device)
optim = torch.optim.Adam(ddpm.parameters(), lr=lr)

```

Creates a U-Net model by instantiating the `Unet()` class

Creates a DDPM to help train the U-Net model

The objective of the U-Net model is to predict the noise in noisy images. It plays a crucial role in the DDPM we just created. Specifically, the DDPM takes clean images from the training set and adds noise to them to create noisy images. It then presents the noisy images to the U-Net and asks it to predict the noise in the images. The DDPM then calculates the mean squared error between the actual noise and the predicted noise. We'll train the model for 20 epochs.

#### Listing 6.8 The training loop

```
from tqdm import tqdm
scaler = torch.amp.GradScaler("cuda")
for ep in range(n_epoch):
    print(f'epoch {ep}')
    ddpm.train()
    optim.param_groups[0]["lr"] = lr * (1 - ep / n_epoch)
    pbar = tqdm(data_loader)
    loss_ema = None
    for x, c in pbar:
        x = x.to(device)
        c = c.to(device)
        with torch.amp.autocast("cuda"):
            loss = ddpm(x, c)
        optim.zero_grad()
        scaler.scale(loss).backward()
        scaler.step(optim)
        scaler.update()
        if loss_ema is None:
            loss_ema = loss.item()
        else:
            loss_ema = 0.95 * loss_ema + 0.05 * loss.item()
        pbar.set_description(f"loss: {loss_ema:.4f}")
        optim.step()
    torch.save(model.state_dict(), "files/conditional.pth")
```

Iterates through all batches of training data

Uses the DDPM to calculate the error between the predicted noise and the actual noise

Backpropagates to adjust model parameters

Saves the trained model parameters

In each epoch, we iterate through all batches of training data ( $x$ ,  $c$ ), where  $x$  is the clean clothing-item images and  $c$  is the labels. The DDPM calculates the mean square error between the actual noise and the noise predicted by the denoising U-Net model. We adjust model parameters to minimize this error.

After training, the model weights are saved on your computer. Alternatively, you can download the trained model from my Google Drive (<https://mng.bz/xZyX>). Unzip the file after downloading, and place it in the `/files/` folder on your computer.

## 6.4 Generating images with the trained diffusion model

In this section, we'll generate images conditional on the labeling information using the model we just trained. First, you'll see how the trained model generates images from random noise. We use 1,000 reference steps, and you'll see the generated image after every 200 time steps.

Second, we'll experiment with the generation guidance parameter `guide_w`. When the value of `guide_w` is large, the generated images match the labels well but lack diversity. When `guide_w` is small, you see more diversity, but the generated images may not match the labels closely.

### 6.4.1 Visualizing generated images

We'll first load the trained model weights to the U-Net denoising model. We then generate 10 images, as shown in the following listing.

**Listing 6.9** Generating images using the trained model

```
from utils.cfr_util import sample
model.load_state_dict(torch.load("files/conditional.pth",
                                weights_only=True,
                                map_location=device))
x,c=next(iter(dataloader))
x,c=x.to(device),c.to(device)
ddpm.eval()
n_sample = n_classes
x_gen,x_gen_store=sample(ddpm,model,n_sample,(1,28,28),
                        device,guide_w=2)
x_real = torch.Tensor(x_gen.shape).to(device)
for k in range(n_classes):
    try:
        idx = torch.squeeze((c == k).nonzero())[0]
    except:
        idx = 0
    x_real[k] = x[idx]
x_all = torch.cat([x_gen, x_real])
```

← Loads trained model parameters

← Generates 10 images using the trained model

← Selects 10 real images for comparison

The 10 generated images are in the tensor `x_gen`, with labels 0 to 9. We then select 10 clean images from the training set, with labels 0 to 9 as well. The real images are saved in `x_real`. We concatenate the 20 images in a tensor `x_all`. Next, we plot the generated images along with the real images so that we can compare them, as in the following listing.

**Listing 6.10** Comparing generated images with real images

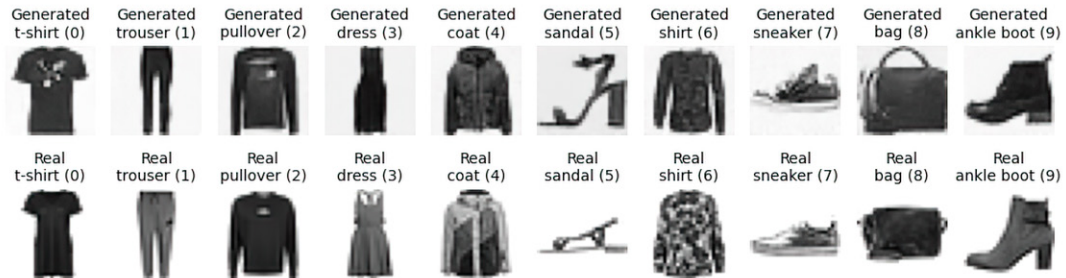
```
captions=[]
for i in range(20):
    gen="Generated" if i<10 else "Real"
    num=i%10
    label_and_num=f"{gen}\n{text_labels[num]} ({num})"
    captions.append(label_and_num)
from matplotlib import pyplot as plt
plt.figure(figsize=(10,3),dpi=100)
for i in range(20):
    plt.subplot(2,10,i+1)
    plt.imshow(x_all[i].cpu().permute(1,2,0)/2+0.5,
               cmap="binary")
```

← Creates a description for each image

← Plots the generated images on top and the real images at the bottom

```
plt.axis('off')
plt.title(captions[i], fontsize=10)
plt.tight_layout()
plt.show()
```

After running the code block from listing 6.10, you'll see the image shown in figure 6.7.



**Figure 6.7** Comparing the generated images with real images. The top row shows images generated by the trained denoising U-Net model, with labels 0 to 9, respectively. The bottom row displays real images from the training set, also with labels 0 to 9.

We plot the generated images along with the real images. We create a description for each image, displaying whether it's generated or real, along with the text label and the numerical label. We plot the images in a  $2 \times 10$  grid, with generated images in the top row and real images at the bottom row. As you can see, the generated images match the label, and they resemble the real images from the training set.

#### 6.4.2 How the guidance parameter affects generated images

So far, we've set the guidance parameter `guide_w` to 2. What happens if we set the value to a different number, say, 5? Let's find out.

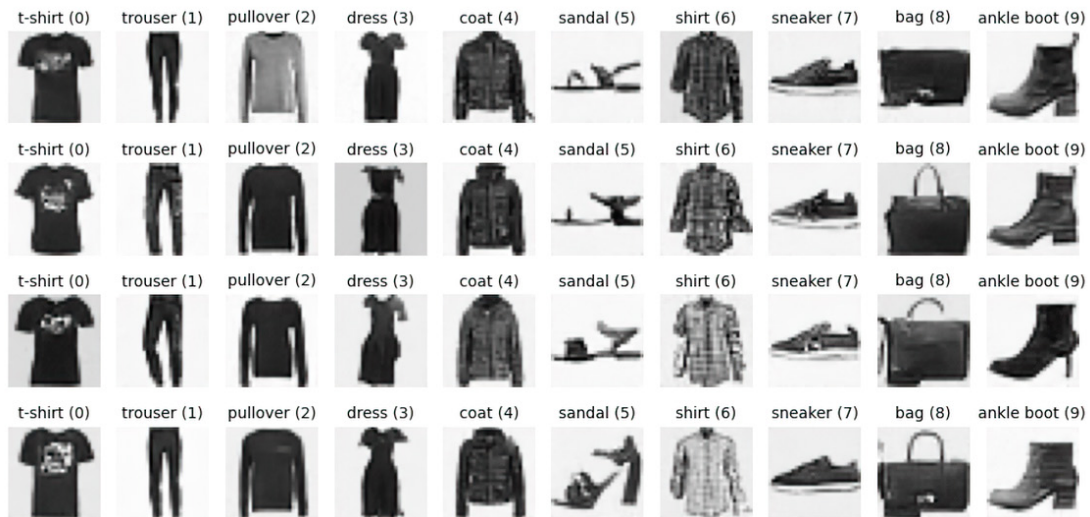
##### Listing 6.11 Generating images with more guidance

```
n_sample = 4*n_classes
x_gen, x_gen_store = sample(ddpm, model, n_sample, (1, 28, 28),
                             device, guide_w=5)
plt.figure(figsize=(10, 5), dpi=100)
for i in range(40):
    plt.subplot(4, 10, i+1)
    plt.imshow(x_gen[i].cpu().permute(1, 2, 0)/2+0.5,
               cmap="binary")
    plt.axis('off')
    c=i%10
    plt.title(f"{text_labels[c]} ({c})", fontsize=10)
plt.tight_layout()
plt.show()
```

Generates 40 images by setting the guidance parameter to 5

Plots the generated images in a  $4 \times 10$  grid

After running the preceding code block, you'll see an image similar to figure 6.8.



**Figure 6.8** Generated images by the trained denoising U-Net model when the guidance parameter is high (with a value of 5)

We generate 40 images and plot them in a  $4 \times 10$  grid. The generated images match the label closely. The first column displays t-shirts with labels 0, the second column shows trousers with labels 1, and so on. The last column shows four ankle boots with label 9. The items match the labels fairly closely because the guidance parameter value is relatively high (value of 5).

### Exercise 6.5

Generate 40 images by setting the guidance parameter to 1. Then, plot the generated images in a  $4 \times 10$  grid.

So far, the images generated by our diffusion models have been grayscale. This simplification is intentional, allowing us to focus on the core principles of how diffusion models work. In the next chapter, we'll build a more advanced diffusion model capable of generating high-resolution color images. Additionally, we'll explore how the initial noise tensor at time step 1,000 influences the final output. By interpolating between different noise tensors, we can create composite images that blend multiple color images. This also allows us to generate images that smoothly transition from one to another by adjusting the weights on the noise tensors.



## Summary

- Traditional diffusion models typically rely on external classifiers to guide the image-generation process using a technique known as classifier guidance. In 2022, Jonathan Ho and Tim Salimans introduced classifier-free guidance (CFG) and showed that effective guidance during generation can be achieved using a purely generative model without relying on an external classifier. Their approach involves jointly training a diffusion model under both conditional and unconditional settings.
- The contracting path of a U-Net is designed to learn feature representations and progressively downsample the input image. This path usually consists of several blocks of convolutional layers followed by pooling layers. Convolutional layers extract feature maps by detecting patterns such as edges, textures, or shapes at various levels of abstraction. Pooling layers reduce the spatial dimensions of the feature maps, making the model computationally efficient while capturing more abstract representations.
- The bottleneck path in U-Net serves as a bridge between the contracting and expansive paths. It processes the most abstracted features extracted by the contracting path and prepares them for reconstruction in the expansive path. The bottleneck contains layers with the smallest spatial dimensions and the most condensed information about the image.
- The expansive path of a U-Net upscales the downsampled feature maps and reconstructs the image. This path usually consists of transposed convolutional layers and additional convolutional layers. Transposed convolutional layers upsample the feature maps, effectively filling in gaps and increasing spatial resolution.
- To enhance the reconstruction process in a U-Net, skip connections are used to combine feature maps from the contracting path with those in the expansive path. This ensures that fine-grained details lost during downsampling are retained and incorporated back into the upsampled image.

# 7

## *Generate high-resolution images with diffusion models*

---

### ***This chapter covers***

- The Denoising Diffusion Implicit Model noise scheduler
- Adding the attention mechanism in denoising U-Net models
- Generating high-resolution images with advanced diffusion models
- Interpolating initial noise tensors to generate a series of images

In the previous two chapters, you've gained a foundational understanding of diffusion models, learning how they add noise to clean images and then reverse this process to generate new images from pure noise. By using the powerful denoising U-Net architecture, you saw how a model can be trained to transform pure noise into grayscale clothing-item images, step-by-step.

But what does it take to move from simple grayscale images to richly detailed, high-resolution color images? And how can we make these models not only more accurate but also faster and more efficient at generating such images? This chapter

tackles these questions by introducing advanced tools and techniques that are now the backbone of state-of-the-art text-to-image generators.

You'll learn how to generate high-resolution color images using advanced diffusion models. You'll use a more sophisticated noise scheduler, the Denoising Diffusion Implicit Model (DDIM), and understand its mechanics and advantages. We'll emphasize the crucial role of attention mechanisms in scaling up U-Nets for complex image generation. To illustrate the role of the initial noise tensor in determining the final outcome, we'll interpolate between different initial noise tensors to create smooth transitions from one image to another, illustrating the power of controlled generative modeling.

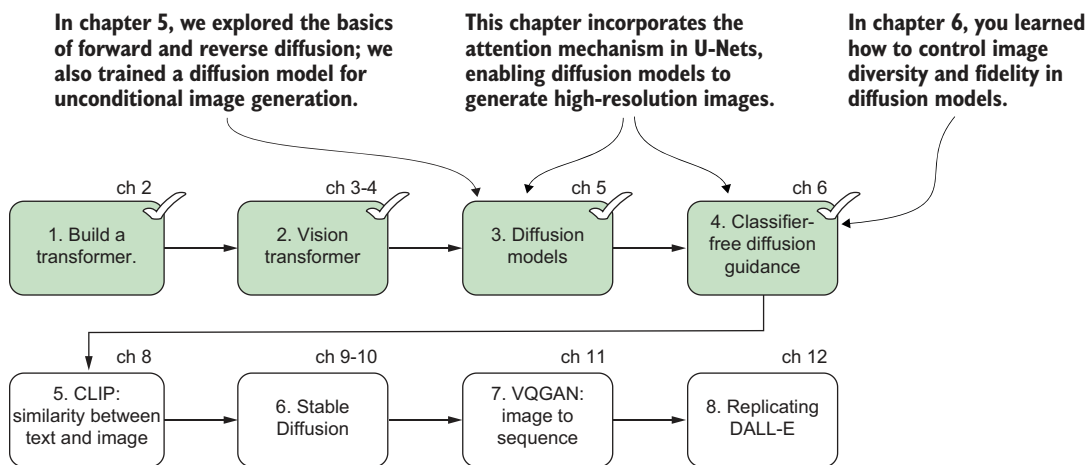
Higher dimensionality and increased data complexity associated with high-resolution image generation demand deeper, more sophisticated neural networks. As a result, you'll build a significantly enhanced U-Net, with more downsampling and upsampling layers, to capture the features of high-resolution images. However, simply adding more layers isn't enough. As the network grows, it faces a new problem: identifying the most relevant features in a flood of information. This is where the attention mechanism comes in, allowing the model to dynamically focus on important image regions, just as you saw in transformer-based models for language translation in earlier chapters.

You've already experienced Denoising Diffusion Probabilistic Models (DDPMs) in earlier chapters, which achieve impressive image quality but require hundreds or thousands of iterative steps. This chapter introduces DDIM, which dramatically accelerates image generation without sacrificing output quality. By rethinking the noise scheduling process, DDIM makes diffusion models vastly more practical for generating high-resolution images.

A defining property of diffusion models is that each output image is determined by the initial noise tensor fed into the model. Change the noise, and you change the result: one noise tensor may lead to a white lily while a different noise tensor may result in a yellow aster. To illustrate the role played by the initial noise tensor, you'll learn how to interpolate between two noise tensors, generating a seamless transition of images that blend characteristics from both starting points. This not only illustrates the flexibility of diffusion models but also opens the door to creative applications in generative art and animation.

Figure 7.1 lays out the eight steps of building your own text-to-image models. This chapter reinforces and goes deeper into steps 3 and 4. By incorporating the attention mechanism in the denoising U-Net model, we allow a U-Net with more layers to generate high-resolution images.

The integration of attention mechanisms and efficient schedulers such as DDIM leads to a large improvement in the capabilities of diffusion models, paving the way for groundbreaking text-to-image generators such as Stable Diffusion, which can produce vivid, high-resolution images matching any text description. By mastering these techniques now, you're building a solid foundation for the next chapters, where you'll see how models such as latent diffusion and Stable Diffusion achieve their remarkable results.



**Figure 7.1** Eight steps for building a text-to-image generator from scratch. In this chapter, you'll focus on mastering steps 3 and 4: enhancing the U-Net architecture with attention and using advanced noise scheduling with DDIM. By incorporating attention into a deeper U-Net, you empower your diffusion models to generate highly detailed images that can accurately match text prompts in future chapters.

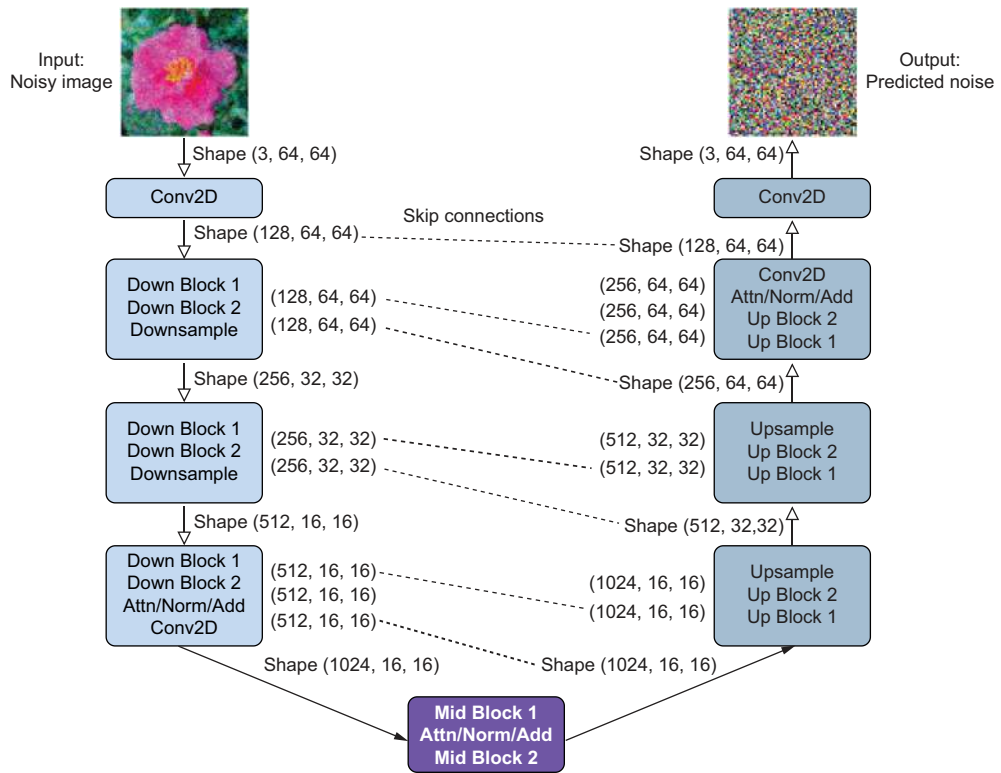
## 7.1 *Attention in U-Net, DDIM, and image interpolation*

In this section, we'll discuss the new features in the denoising U-Net model compared to the one we used in the previous chapter. We'll add more downsampling and upsampling layers and include the attention mechanism in the U-Net. This attention mechanism helps the network focus on the most relevant parts of the image, such as shapes, textures, or regions tied to the input prompt, so it can learn more effectively and generate higher-quality results.

DDIM and DDPM are two types of noise schedulers used in diffusion models for generative tasks. DDPM generates images through a randomized (stochastic) process and follows a Markovian process, meaning each step depends only on the previous step. While effective, this method requires many steps to produce high-quality images. In contrast, DDIM uses a deterministic, non-Markovian process, allowing it to generate images in fewer steps without sacrificing quality. This makes DDIM significantly more efficient than DDPM, an important advancement for generating high-resolution images quickly. While you've coded DDPM from scratch in the previous chapter, you'll use DDIM in this chapter and learn why it's more efficient than DDPM. Finally, we'll explain the role of the initial noise tensor in determining the final output in diffusion models and how interpolating two initial noise tensors can lead to interpolated images.

### 7.1.1 *Incorporating the attention mechanism in the U-Net model*

Compared to the U-Net model we created in chapter 6, the U-Net model in this chapter has more layers and also incorporates the attention mechanism. Figure 7.2 is a diagram of the components of the U-Net.



**Figure 7.2** The architecture of the denoising U-Net model to generate high-resolution flower images. The U-Net is characterized by its symmetric U-shape, with the contracting path on the left and the expansive path on the right, connected by the bottleneck block at the bottom. The model takes noisy images as inputs and predicts the noise in the images.

The top-left corner of figure 7.2 is a noisy image, which is the input to the denoising U-Net. The U-Net predicts the noise in the image, based on which time step the noisy image is in. Once we know the noise in the image, we can reconstruct the clean image because the noisy image is a weighted sum of the original clean image and noise.

The noisy image goes through multiple down blocks in the contracting path (left side of figure 7.2), which consists of various convolutional layers and pooling layers. The model extracts and encodes features at different levels of abstraction. The bottleneck block (bottom of figure 7.2) consists of convolutional layers, and it captures the most abstract representations of the image. The expansive path (right side of the figure) consists of multiple up blocks, which upsample the feature maps, reconstructing the image. It does this by incorporating features from the contracting path through skip connections (denoted by dashed lines in the figure).

Skip connections combine high-level, abstract features from the contracting path with low-level, detailed features from the expansive path. This allows the U-Net model

to better reconstruct fine details in the denoising process. However, skip connections lead to redundant feature extractions in our denoising U-Net, which complicates the identification of relevant features. The attention mechanism is needed so that the model can learn to emphasize important features while disregarding irrelevant ones.

The attention mechanism is used in both the contracting and expansive paths (as shown in figure 7.2 with the label Attn/Norm/Add). The attention mechanism is the same as the one we used in chapter 2. However, in this context, the attention mechanism is applied to image pixels instead of text tokens. Including the scaled dot-product attention (SDPA) mechanism allows the model to weigh the importance of different spatial regions dynamically. This is crucial for denoising tasks, where understanding the broader context of the noisy data helps in predicting the clean data more accurately.

### 7.1.2 *Denoising Diffusion Implicit Models*

DDIM was first proposed by Jiaming Song, Chenlin Meng, and Stefano Ermon in 2020 [1]. The authors showed that while DDPMs have achieved high-quality image generation without adversarial training, they require simulating a Markov chain for many steps to produce a sample (in a Markov process, the future state depends only on the present state and not on past states). For color images with high resolutions, this can be extremely computationally intensive and time-consuming. DDIMs were shown as a more efficient class of iterative implicit probabilistic models.

Song, Meng, and Ermon [1] constructed a class of non-Markovian diffusion processes that lead to the same training objective, but whose reverse process can be much faster to sample from compared to DDPMs. They demonstrated that DDIMs can produce high-quality samples 10 to 50 times faster compared to DDPMs. Further, DDPMs use a stochastic sampling process, while DDIMs can use a deterministic approach. DDIMs generally require fewer steps for sampling, making them more efficient.

DDIM's non-Markovian process allows for more flexibility in the denoising step. In particular, Equation (12) in Song, Meng, and Ermon [1] shows the reverse diffusion process as

$$x_{t-1} = \sqrt{\tilde{\alpha}_{t-1}} * (\text{predicted } x_0) + (\text{direction pointing to } x_t) + \sigma_t \epsilon_t \quad (7.1)$$

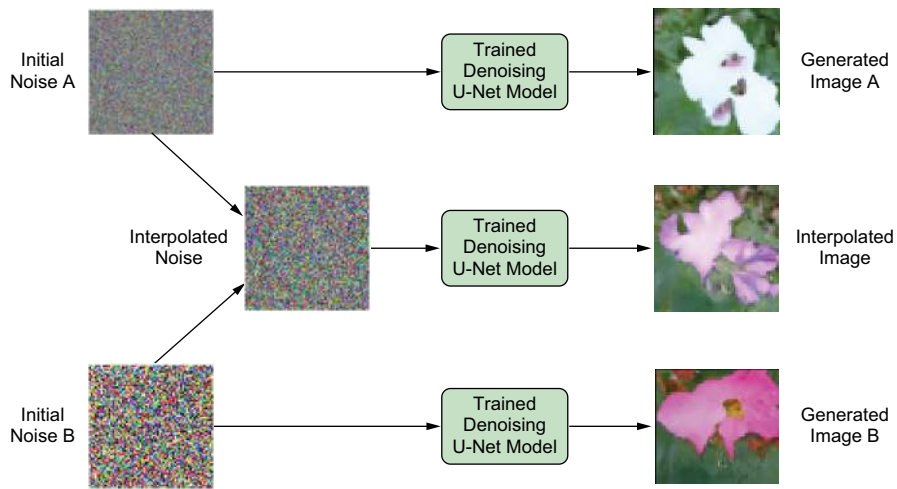
In this equation,  $\epsilon_t$  follows a standard normal distribution, and  $\sigma_t$  governs how much random noise is in the reverse diffusion process. Recall that  $x_0$  is the clean image at time step 0. A special case is when  $\sigma_t = 0$ , and the reverse diffusion process becomes deterministic: this is the setting used in deterministic DDIM sampling, which enables faster image generation by skipping stochastic noise and following a fixed trajectory through the latent space.

### 7.1.3 *Image interpolation in diffusion models*

Once the model is trained, we feed an initial noise tensor to the U-Net, which refines the noise tensor into a structured and detailed image. Because the output is influenced

by the specific noise tensor provided at the start, different noise inputs lead to different final images. This property offers flexibility and variety in image generation, allowing us to explore the relationship between the initial noise tensor and the final output.

In particular, this chapter explores this relationship through interpolation between different noise tensors. By blending two different noise tensors in varying proportions, we can create a sequence of intermediate noise states, which in turn generate a series of final images that gradually transition from one image to another. This technique provides valuable insights into how the initial noise tensor affects generative outcomes, demonstrating the nuanced control possible in diffusion-based image synthesis. Figure 7.3 illustrates how image interpolation works in diffusion models.



**Figure 7.3** How image interpolation works in diffusion models. Initial noise tensor A (top left) leads to final output A (top right), while initial noise tensor B (bottom left) leads to final output B (bottom right). We create an interpolated noise tensor (middle left) by averaging initial noise tensors A and B. We then feed this interpolated noise tensor into the trained denoising U-Net model to obtain an interpolated output image (middle right), which shows the characteristics of both final images A and B.

The top-left corner of figure 7.3 shows the initial noise tensor A. If we feed it to the trained diffusion model, the final output is a white flower (image A, as shown at the top-right corner of the figure). The bottom-left corner of figure 7.3 shows another Initial Noise tensor, B. If we feed the noise tensor B to the trained diffusion model, the final output is a red flower (image B in the bottom-right corner, shown in a darker shade in this book).

We can interpolate between two noise tensors. Interpolation involves blending two different noise tensors in varying proportions to create a continuum of intermediate noise states. Mathematically, if we have two noise tensors,  $z_1$  and  $z_2$ , we can define an interpolated noise tensor as

$$z_{\alpha} = \alpha z_1 + (1 - \alpha) z_2 \quad (7.2)$$

where  $\alpha$  is a blending coefficient that varies between 0 and 1. When  $\alpha = 1$ , the interpolated tensor is identical to  $z_1$ , while  $\alpha = 0$  results in  $z_2$ . For values of  $\alpha$  between 0 and 1, the resulting noise tensor represents a mixture of both original noise tensors, allowing for a gradual transition from one image to another.

In the middle of figure 7.3, we interpolate the initial noise tensors A and B by placing equal weights on them (i.e., setting  $\alpha = 0.5$  in equation 7.2). We then feed the interpolated noise tensor to the trained denoising U-Net model, and the output is an image that has characteristics of both image A and image B (see middle-right in the figure).

Later, we'll apply interpolation to image generation to create a sequence of images that smoothly transition from one image to another. Suppose we start with two noise tensors: one that produces an image of a white lily and another that produces an image of a yellow aster. By generating images at different interpolation levels (e.g.,  $\alpha = 0.05, 0.15, \dots, 0.95$ ), we obtain a set of composite images that change gradually from an image similar to a lily to one similar to an aster.

Noise interpolation has several practical applications in generative AI and artistic image synthesis. One benefit is controlled image generation: by adjusting the interpolation parameter, you can exert fine-grained control over the generated output, enabling the creation of hybrid images that incorporate elements from multiple sources. It can also lead to smooth transitions in animation: interpolated noise sequences can be used to generate animations that smoothly transition between different visual concepts, providing a novel approach to AI-generated motion graphics. Artists and designers can use noise interpolation to explore new creative possibilities, generating variations of images that blend characteristics in unique and unexpected ways.

## 7.2 *High-resolution flower images as training data*

To train the denoising U-Net model, we'll use the Oxford 102 Flower dataset, which contains roughly 8,000 flower images. You'll use the `datasets` library to download the data from Hugging Face (a community that hosts and collaborates on AI models, datasets, and applications) to your computer. You'll then visualize the images in the training dataset and place the data in batches so that we can use them to train the denoising U-Net model later.

**NOTE** The Python programs for this chapter can be accessed on the book's GitHub repository (<https://github.com/markhliu/txt2img>) and are also available as a Google Colab notebook (<https://mng.bz/AGgx>) for interactive exploration.

The Python programs in this chapter are adapted from Hugging Face's GitHub repository (<https://github.com/huggingface/diffusers>) and Filip Basara's GitHub



repository (<https://github.com/filipbasara0/simple-diffusion>). Before we start, run the following line of code to install several libraries that we'll use:

```
!pip install datasets einops diffusers
```

### 7.2.1 Visualizing images in the training dataset

The `load_dataset()` method from the `datasets` library allows you to download the Oxford 102 Flower dataset from Hugging Face directly, as shown in the following listing.

#### Listing 7.1 Downloading the training dataset

```
from datasets import load_dataset
from torchvision.transforms import (CenterCrop,
    Compose, InterpolationMode, RandomHorizontalFlip,
    Resize, ToTensor)

resolution=64

augmentations = Compose([
    Resize(resolution,
        interpolation=InterpolationMode.BILINEAR),
    CenterCrop(resolution),
    RandomHorizontalFlip(),
    ToTensor(),])

def transforms(examples):
    images = [augmentations(image.convert("RGB"))
        for image in examples["image"]]
    return {"input": images}

dataset = load_dataset("huggan/flowers-102-categories",
    split="train",)
dataset.set_transform(transforms)
```

Defines several image augmentation methods

Downloads the data from Hugging Face directly

Performs image transformations

After downloading the data from Hugging Face, we perform several image transformations such as center crop (the central region of the image is extracted while the rest is discarded) and random horizontal flip (creates a mirror image of the original image). These image transformations and augmentations help our model learn more robust and generalized representations from the training data, thus improving the quality of the generated images.

To prepare for training, we place the dataset in batches of four. Adjust the batch size to two or even one (or switch to CPU training) if your GPU memory is small:

```
import torch

torch.manual_seed(42)
batch_size=4
```

```
train_dataloader=torch.utils.data.DataLoader(
    dataset, batch_size=batch_size, shuffle=True)
```

Next, we use the matplotlib library to visualize some examples of the flower images in the dataset:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(5.9,batch_size),dpi=150)
for col in range(6):
    imgs=next(iter(train_dataloader))["input"]
    for row in range(batch_size):
        plt.subplot(batch_size,6,col+1+row*6)
        img=imgs[row].permute(1,2,0)
        plt.imshow(torch.clip(img,0,1))
        plt.axis('off')
plt.tight_layout()
plt.show()
```

Plots sample images in four rows and six columns

Changes from channel first to channel last

Clips image values so they fall in the range [0, 1]

We plot 24 sample images in a  $4 \times 6$  grid. PyTorch uses channel-first tensors to represent images, so we use the `permute()` method to change the tensors to channel-last format before displaying them.

### Channel-first and channel-last image formats

In image processing, channel-first and channel-last formats refer to how the color channels (RGB, grayscale, etc.) are arranged in an image tensor or array. PyTorch uses the channel-first format, meaning the shape is (C, H, W), where C is the number of channels (e.g., 3 for RGB, 1 for grayscale), and H and W are height and width, respectively.

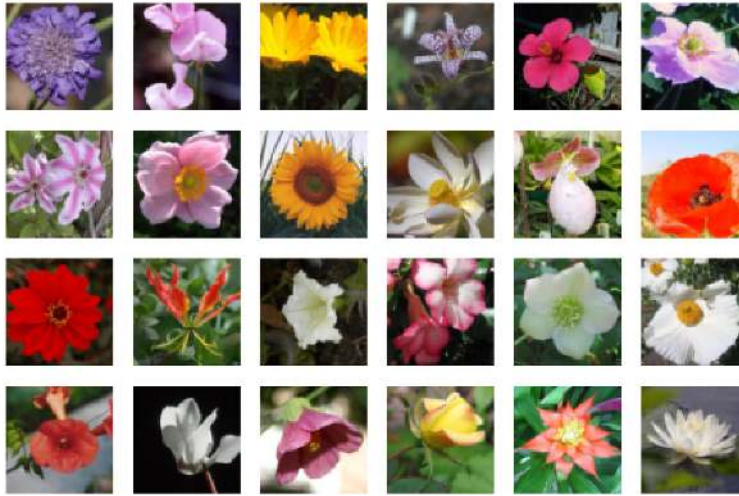
For example, in this chapter, images have a format of (3, 64, 64), meaning they are three-channel color images with a height and width of 64 pixels. Most other Python libraries use the channel-last format (Matplotlib, NumPy, TensorFlow, and Python Imaging Library [PIL]). The image tensor has a shape of (H, W, C) in channel-last format. The color flower images are converted to a shape of (64, 64, 3) before we display them with the Matplotlib library.

After running the preceding code block, you'll see an image similar to figure 7.4.

These are color images of various types of flowers (shown in grayscale in this book). We've standardized the size of each image to (3, 64, 64) so that we can place them in batches to feed to the model.

### Exercise 7.1

Plot images from the training set in a  $2 \times 5$  grid.



**Figure 7.4** Sample images from the Oxford 102 Flower dataset, which contains roughly 8,000 flower images. We use this dataset to train the denoising U-Net model.

### 7.2.2 Applying forward diffusion on flower images

Next, we apply the DDIM noise scheduler on the clean flower images to add noise to them so that we can use the noisy images to train our model. To save space, we'll place most helper functions and classes in a local module `ddim_util.py`. Download the file from the book's GitHub repository (<https://github.com/markhliu/txt2img>), and place it in the `/utils/` folder on your computer (or simply clone the repository to your computer). If you're working in Google Colab, you can quickly set up the repository and add it to your working directory by running

```
!git clone https://github.com/markhliu/txt2img
import sys
sys.path.append("/content/txt2img")
```

In the file `ddim_util.py` that you just downloaded, we define a class `DDIMScheduler()` to model the DDIM, as shown in listing 7.2. Take a look at the definition of the class to become familiar with its various attributes and methods. In particular, we'll add noise to several images to visualize how the forward diffusion process works in DDIM; the output is shown in figure 7.5.

#### Listing 7.2 Applying forward diffusion to flower images

```
from utils.ddim_util import DDIMScheduler
noise_scheduler=DDIMScheduler(num_train_timesteps=1000)

imgs=next(iter(train_data_loader))["input"]
imgs=imgs.permute(0,2,3,1).reshape(-1,64,3)

all_imgs=[imgs]
```

← Obtains a batch of images and changes them to channel-last format

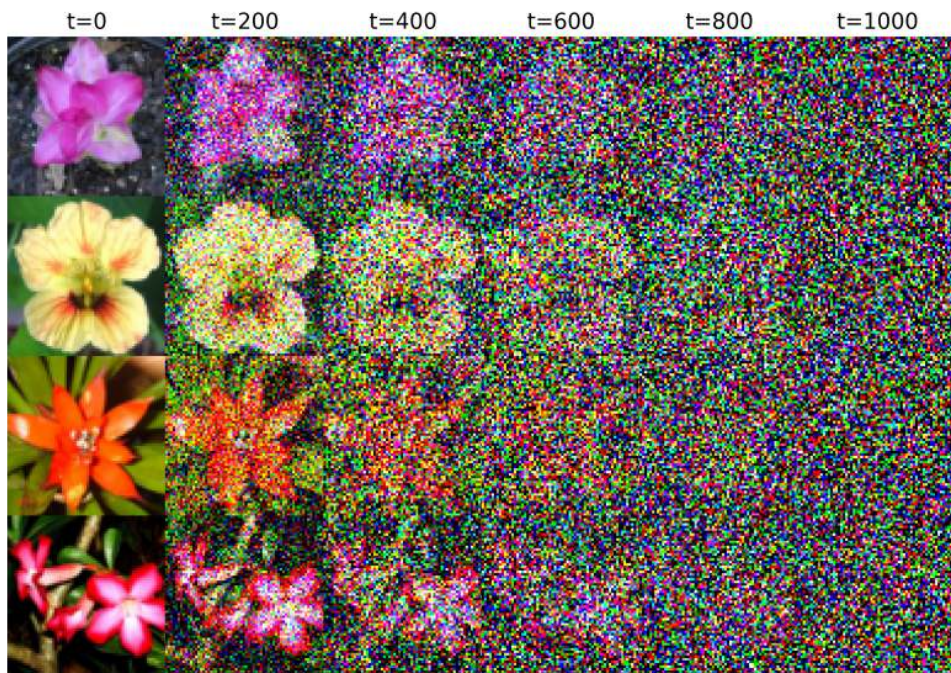
```

for step in [200,400,600,800,1000]:
    timesteps=torch.tensor([step-1]).long()
    noisy_image=noise_scheduler.add_noise(imgs,
        torch.randn(imgs.shape), timesteps)
    allimgs.append(noisy_image)
plt.figure(figsize=(10,8),dpi=80)
for i in range(6):
    plt.subplot(1,6,i+1)
    plt.imshow(torch.clip(allimgs[i],0,1))
    plt.axis('off')
    plt.title(f"t={200*i}",fontsize=16)
plt.tight_layout(w_pad=-0.1)
plt.show()

```

← Creates noisy images at time steps 200, 400, ..., 1,000

← Plots the images in a 4 × 6 grid



**Figure 7.5** The forward diffusion process using the DDIM noise scheduler. The four images in the leftmost column are clean images from the training dataset to which we gradually add noise.

We use  $T = 1,000$  time steps in the forward diffusion process, in which noise is gradually added to the images. The left column is time step 0, when the four images are the original clean images, without any noise added to them. The second column shows images in time step 200, when some noise is added to the images. As we move to the right, more and more noise is added to the images, until at time step  $T = 1,000$ , the four images become pure random noise.

**Exercise 7.2**

Modify code listing 7.2. Use four images from the training set to generate images in time steps 0, 250, 500, 750, and 1,000. Plot them in a  $4 \times 5$  grid.

Now that we have the training data ready, we'll build and train a denoising model to generate high-resolution flower images that resemble those in the training set.

**7.3 Building and training a U-Net for high-resolution images**

In this section, we build a denoising U-Net model that integrates the attention mechanism discussed earlier in the chapter. Our U-Net model is large, comprising more than 133 million parameters. It features multiple convolutional layers connected through skip connections, which merge features from different network levels. This design preserves spatial information, enhancing the model's learning capability.

Due to the model's large size and redundant feature extraction, we incorporate the SDPA attention mechanism to focus on the most relevant aspects of the input. To compute SDPA attention, we flatten the image, treating its pixels as a sequence. This allows SDPA to capture dependencies among pixels, similar to how we modeled token relationships in text in chapter 2. Finally, we train the model using the flower images prepared earlier.

**7.3.1 Building the denoising U-Net model**

The diffusion model we construct is an unconditional model, meaning it generates flower images without any labeling information. As a result, the model can produce an image of a flower from any category in the training set, such as a lily, aster, peony, or other types. Unlike the conditional model we discussed in the previous chapter, the U-Net in this chapter only embeds time step information and doesn't incorporate class labels.

We define the U-Net model in the local module file `ddim_util.py`. In particular, the file contains a `sinusoidal_embedding()` function, as follows, to embed the time step information:

```
def sinusoidal_embedding(timesteps, dim):
    half_dim = dim // 2
    exponent = -math.log(10000) * torch.arange(
        start=0, end=half_dim, dtype=torch.float32)
    exponent = exponent / (half_dim - 1.0)
    emb = torch.exp(exponent).to(device=timesteps.device)
    emb = timesteps[:, None].float() * emb[None, :]
    return torch.cat([emb.sin(), emb.cos()], dim=-1)
```

In fixed positional encoding, the sine function applies to even positions and the cosine function to odd positions.

We use fixed positional encoding here, similar to what we did in chapter 2, instead of the learned positional encoding we used in chapter 3. Because we have a large model,



fixed positional encoding reduces the number of learnable parameters, which can speed up the training process. To implement the attention mechanism, we defined an `Attention()` class in the local module, as shown in the following listing.

**Listing 7.3** The attention mechanism in the U-Net

```
class Attention(nn.Module):
    def __init__(self, dim, heads=4, dim_head=32):
        super().__init__()
        self.scale = dim_head**-0.5
        self.heads = heads
        hidden_dim = dim_head * heads
        self.to_qkv = nn.Conv2d(dim, hidden_dim * 3, 1, bias=False)
        self.to_out = nn.Conv2d(hidden_dim, dim, 1)

    def forward(self, x):
        b, c, h, w = x.shape
        qkv = self.to_qkv(x).chunk(3, m=1)  # ← Calculates query, key,
        q, k, v = map(                        # and value tensors
            lambda t: rearrange(t, 'b (h c) x y -> b h c (x y)', h=self.heads),
            qkv)
        q = q * self.scale
        sim = einsum('b h d i, b h d j -> b h i j', q, k)  # ← Splits query, key, and
        attn = sim.softmax(dim=-1)                          # value into multiple
        out = einsum('b h i j, b h d j -> b h i',           # heads to apply multi-
            attn, v)                                          # head attention
        out = rearrange(out, 'b h (x y) d -> b (h d) x y', x=h, y=w)  # ←
        return self.to_out(out)                             # Calculates the attention
                                                            # tensor in each head

        # Concatenates attention tensors
        # from different heads into one
        # attention tensor as the output
```

This is the standard SDPA attention mechanism that we often use in transformers and natural language processing (NLP). Specifically, we first pass the input through three linear layers to obtain query, key, and value. We then split the query, key, and value tensors into four heads and calculate the attention in each head. Finally, we concatenate the four attention tensors from the four heads into one single attention tensor as the final output. We create a denoising U-Net model as follows:

```
from utils.ddim_util import UNet  # ← Imports the UNet() class
                                   # from the local module

device="cuda" if torch.cuda.is_available() else "cpu"
resolution=64
model=UNet(3,hidden_dims=[128,256,512,1024],
            image_size=resolution).to(device)  # ← Instantiates the UNet() class to
num=sum(p.numel() for p in model.parameters()) # create a denoising U-Net model
print("number of parameters: %.2fM" % (num/1e6,))  # ← Counts the number of
                                                    # model parameters
```

The output is

```
number of parameters: 133.42M
```

We first import the `UNet()` class from the local module. We then instantiate the class to create a denoising U-Net model. We count the total number of model parameters. The preceding output shows that there are more than 133 million parameters.

### 7.3.2 Training the denoising U-Net model

During each training epoch, we iterate through all batches in the training dataset. For each image, we randomly select a time step and introduce noise to the clean image based on this value, generating a noisy version. These noisy images, along with their corresponding time step values, are then passed into the denoising U-Net model, which predicts the added noise. The predicted noise is compared to the ground truth (the actual noise introduced), and the model parameters are updated to minimize the mean absolute error between them.

To optimize the model, we use the AdamW optimizer, a variant of the Adam optimizer employed throughout this book. Additionally, we implement a learning rate scheduler from the `diffusers` library to dynamically adjust the learning rate during training. Using a higher initial learning rate helps the model escape local minima, while gradually reducing it in later stages promotes stable convergence toward a global minimum. The learning rate scheduler is defined as shown here:

```
from diffusers.optimization import get_scheduler

optimizer=torch.optim.AdamW(model.parameters(),lr=0.0001,
    betas=(0.95,0.999),weight_decay=0.00001,eps=1e-8)
lr_scheduler=get_scheduler(
    "cosine",
    optimizer=optimizer,
    num_warmup_steps=300,
    num_training_steps=(len(train_data_loader) * 100))
```

Uses the AdamW optimizer

Uses the learning rate scheduler from the `diffusers` library

For simplicity, we train the model for 100 epochs, as shown in listing 7.4. If you like, you can use early stopping to determine how many epochs to train, as we did in chapter 3. We print out the average loss during the training process.

#### Listing 7.4 Training the denoising U-Net model

```
for epoch in range(100):
    model.train()
    tloss = 0
    print(f"start epoch {epoch}")
    for step, batch in enumerate(train_data_loader):
        clean_images = batch["input"].to(device)*2-1
        nums = clean_images.shape[0]
        noise = torch.randn(clean_images.shape).to(device)
        timesteps = torch.randint(0,
            noise_scheduler.num_train_timesteps,
            (nums, ),
            device=device).long()
        noisy_images = noise_scheduler.add_noise(clean_images,
```

Iterates through all batches in the training dataset

```

        noise, timesteps)
noise_pred = model(noisy_images,
                   timesteps)["sample"]
loss = torch.nn.functional.l1_loss(noise_pred,
                                   noise)
loss.backward()
optimizer.step()
lr_scheduler.step()
optimizer.zero_grad()
tloss += loss.detach().item()
if step%100==0:
    print(f"step {step}, average loss {tloss/(step+1)}")
torch.save(model.state_dict(), 'files/diffusion.pth')

```

← Adds noise to clean images to create noisy images

← Predicts the noise using the denoising U-Net model

← Calculates the mean absolute error between the predicted noise and the actual noise

After training, the trained model weights are saved on your computer. Alternatively, you can download the trained weights from my Google Drive at <https://mng.bz/Z9XA>. Unzip the file after downloading.

Now that the denoising U-Net model is trained, we'll use it to generate flower images. You'll also learn how to interpolate initial noise tensors to generate composite images.

## 7.4 *Image generation and interpolation*

With the denoising U-Net model now trained, we can use it to generate flower images. The generation process unfolds over 50 inference steps, where we set the time step values to 980, 960, . . . , 20, and 0. The image starts as pure random noise, and we feed it into the trained model to produce a slightly denoised image. This output is then iteratively refined by feeding it to the model again, gradually reducing noise over 50 reference steps until we obtain a final image that closely resembles the flowers in the training set.

Beyond generating individual images, we'll also explore how to interpolate between two initial noise tensors. This technique enables you to create a smooth transition between two images, producing a series of intermediate images that visually morph from one flower to another.

### 7.4.1 *Using the trained denoising U-Net to generate images*

Now let's generate the flower images. To do so, we define the `generate()` method in the `DDIMScheduler()` class in the local module `ddim_util.py`.

#### Listing 7.5 Defining a function to generate flower images

```

@torch.no_grad()
def generate(self, model, device, batch_size=1, generator=None,
            eta=1.0, use_clipped_model_output=True, num_inference_steps=50):
    imgs=[]
    image=torch.randn((batch_size,model.in_channels,model.sample_size,
                      model.sample_size),
                      generator=generator).to(device)

```

← Creates a list to store intermediate images

← Generates the initial noise tensor



```

self.set_timesteps(num_inference_steps)
for t in tqdm(self.timesteps):
    model_output = model(image, t)["sample"]
    image = self.step(model_output, t, image, eta,
                      use_clipped_model_output=use_clipped_model_output)
    img = unnormalize_to_zero_to_one(image)
    img = img.cpu().permute(0, 2, 3, 1).numpy()
    imgs.append(img)
image = unnormalize_to_zero_to_one(image)
image = image.cpu().permute(0, 2, 3, 1).numpy()
return image, imgs

```

← Iteratively denoises the image using the trained U-Net model

← Outputs the final clean image and the intermediate noisy images

The `generate()` method feeds a random noise tensor to the trained model to denoise it iteratively. The function returns the final clean image as well as the intermediate noisy images. In the following listing, we use the trained denoising U-Net and the `generate()` method defined previously to create 10 flower images. After running this code, you'll see an output similar to figure 7.6.

#### Listing 7.6 Image generation with the trained denoising U-Net

```

sd=torch.load('files/diffusion.pth',weights_only=True,map_location=device)
model.load_state_dict(sd)
with torch.no_grad():
    generator = torch.manual_seed(100)
    generated_images,imgs = noise_scheduler.generate(
        model,device=device,
        num_inference_steps=50,
        generator=generator,
        eta=0,
        use_clipped_model_output=True,
        batch_size=10)
imgnp=generated_images

import matplotlib.pyplot as plt
plt.figure(figsize=(10,4),dpi=100)
for i in range(10):
    ax = plt.subplot(2, 5, i + 1)
    plt.imshow(imgnp[i])
    plt.xticks([])
    plt.yticks([])
    plt.tight_layout()
plt.show()

```

← Loads the trained model parameters

← Generates 10 flower images using the trained model

← Plots the generated images in a 2 × 5 grid

Because we use an unconditional diffusion model, the images are generated without labeling information. The 10 images in figure 7.6 are of different types, but they do look similar to those in the training set.

#### Exercise 7.3

Modify code listing 7.6, and change the random seed to 42. Keep the rest of the code the same. Rerun the code listing to see what the generated images look like.



**Figure 7.6** Ten flower images generated by the trained denoising U-Net model. We feed 10 random noise tensors to the trained U-Net and use 50 inference steps to gradually remove noise from the noisy images. The 10 images in this figure all look different due to the different initial noise tensors used.

If you're wondering whether we can control which flower images the trained model generates, the answer is yes. Each initial noise tensor produces a unique flower image. In the next subsection, we'll explore how to choose different noise tensors to generate specific images. Even better, you'll learn how to interpolate between noise tensors to create composite images.

#### 7.4.2 *Transition from one image to another*

When we generate flower images with the trained U-Net, the denoising process is influenced by the initial noise tensor we provide. Different noise tensors lead to different final images. Next, we'll interpolate between two noise tensors by blending them in varying proportions to create new noise tensors. We then feed the interpolated noise tensors to the U-Net to generate a series of images that transition from one image to another. To that end, we've defined the following `interpolate()` method in the `DDIM-Scheduler()` class in the local module `ddim_util.py`.

##### Listing 7.7 Defining the `interpolate()` method

```
@torch.no_grad()
def interpolate(self, model, a_idx, b_idx, batch_size=1, generator=None,
               eta=1.0, use_clipped_model_output=True, num_inference_steps=50,
               device=None):
    if device is None:
        device = "cuda" if torch.cuda.is_available() else "cpu"
    image0 = torch.randn((batch_size, model.in_channels,
                          model.sample_size, model.sample_size), generator=generator,
                          ).to(device)
    image = torch.zeros((batch_size, model.in_channels,
                        model.sample_size, model.sample_size)).to(device)
    a, b = image0[a_idx], image0[b_idx]
```

← Picks two initial noise tensors

```

for i in range(10):
    ab=torch.sin(torch.tensor(0.5*math.pi*(0.05+i/10)))*a+\
        torch.cos(torch.tensor(0.5*math.pi*(0.05+i/10)))*b
    image[i]=ab
self.set_timesteps(num_inference_steps)
for t in tqdm(self.timesteps):
    model_output = model(image, t)["sample"]
    image = self.step(model_output,t,image,eta,
        use_clipped_model_output=use_clipped_model_output,
        generator=generator)
image = unnormalize_to_zero_to_one(image)
image = image.cpu().permute(0, 2, 3, 1).numpy()
return image

```

Creates 10 interpolated noise tensors by placing different weights on the 2 noise tensors

Generates images by iteratively denoising the interpolated noise tensors

The `a_idx` and `b_idx` arguments in the `interpolate()` method determine which two initial noise tensors to use. If we fix the random state, the two arguments lead to two specific flower images. By applying different weightings to these initial tensors, we generate 10 interpolated noise tensors. These interpolated tensors are then fed into the trained model to produce a sequence of interpolated images. Now let's create a series of interpolated images using the `interpolate()` method we defined previously.

#### Listing 7.8 Interpolating initial noise tensors

```

with torch.no_grad():
    generator = torch.manual_seed(100)
    generated_images = noise_scheduler.interpolate(
        model,1,7,
        num_inference_steps=50,
        generator=generator,
        eta=0,
        use_clipped_model_output=True,
        batch_size=10)

imgnp=generated_images

import matplotlib.pyplot as plt
plt.figure(figsize=(10,4),dpi=300)
for i in range(10):
    ax = plt.subplot(1,10, i + 1)
    plt.imshow(imgnp[i])
    plt.xticks([])
    plt.yticks([])
    plt.tight_layout()
plt.show()

```

Fixes the random state to 100

Chooses the second and the eighth initial noise tensors

Plots the 10 interpolated images based on the two initial noise tensors

Note that we've fixed the random state to 100, the same random state we used in listing 7.6. Therefore, by setting `a_idx=1` and `b_idx=7` in the `interpolate()` method, we're using the same two initial noise tensors that produce images 2 and 8 in figure 7.6.

(Python uses zero-based indexing, so indexes 1 and 7 correspond to the second and eighth images.) We then create 10 interpolated images based on these two initial noise tensors. The output is shown in figure 7.7.



**Figure 7.7** Interpolated flower images. With the random state set to 100, we select the second and eighth initial noise tensors to generate 10 interpolated noise tensors by applying varying weightings to the originals. These interpolated tensors are then passed through the trained denoising U-Net, producing a series of 10 flower images that smoothly transition from one to the other.

Because the 10 images in figure 7.6 are also generated by fixing the random state to 100, the right image in figure 7.7 looks similar to the second image in figure 7.6, while the left image looks similar to the eighth image in figure 7.6.

### Exercise 7.4

Modify code listing 7.8 to use the third and fourth initial noise tensors to generate 10 interpolated images. Keep the rest of the code the same (in particular, keep the random state to 100). Rerun the code listing to see what the generated images look like.

Finally, we generate three more sets of interpolated images. The following listing shows how.

### Listing 7.9 Generating three sets of composite images

```
with torch.no_grad():
    generator = torch.manual_seed(100)
    generated_images = noise_scheduler.interpolate(
        model, 2, 5,
        num_inference_steps=50,
        generator=generator,
        eta=0,
        use_clipped_model_output=True,
        batch_size=10)
    imgnp1=generated_images
with torch.no_grad():
    generator = torch.manual_seed(100)
    generated_images = noise_scheduler.interpolate(
        model, 7, 2,
        num_inference_steps=50,
        generator=generator,
        eta=0,
        use_clipped_model_output=True,
```

Interpolates using the third and sixth initial noise tensors

Interpolates using the eighth and third initial noise tensors

```

        batch_size=10)
imgnp2=generated_images
with torch.no_grad():
    generator = torch.manual_seed(100)
    generated_images = noise_scheduler.interpolate(
        model,3,1,
        num_inference_steps=50,
        generator=generator,
        eta=0,
        use_clipped_model_output=True,
        batch_size=10)
imgnp3=generated_images

```

Interpolates using the fourth and second initial noise tensors

We first generate a set of 10 interpolated images using the third and sixth initial noise tensors. We then create another set using the eighth and third initial noise tensors. Finally, a third set of interpolated images are created using the fourth and second initial noise tensors. These three sets of images are saved as NumPy arrays, `imgnp1`, `imgnp2`, and `imgnp3`, respectively. Next, we plot them in three rows, like this:

```

import numpy as np
img_array=np.concatenate([imgnp1,imgnp2,imgnp3])
fig, axs = plt.subplots(nrows=3,
                        ncols=10,sharex=True,
                        sharey=True,figsize=(10,3),dpi=100)
for row in range(3):
    for col in range(10):
        axs[row, col].clear()
        axs[row, col].set_xticks([])
        axs[row, col].set_yticks([])
        axs[row, col].imshow(img_array[col+row*10])
plt.subplots_adjust(bottom=0.001,right=0.999,top=0.999,
left=0.001, hspace=-0.1,wspace=-0.1)

```

Concatenates the three sets of images

Plots the images in a  $3 \times 10$  grid

After running the preceding code block, you'll see sets of images as shown in figure 7.8. With the random state set to 100, we select three pairs of initial noise tensors to generate interpolated noise tensors by applying varying weightings to the originals.



**Figure 7.8** Three sets of interpolated flower images generated from using a fixed random state of 100

These initial tensors are passed through the trained U-Net to generate interpolated images, as shown in the figure. In each row, as we go from left to right, the image smoothly transitions from the leftmost image to the rightmost image.

Now that you've grasped how diffusion models function, you'll soon put them to work in building advanced text-to-image models, such as latent diffusion models and Stable Diffusion, later in this book. These models generate images based on text prompts, but how can you tell if the resulting images truly match the given text? The next chapter tackles this very question. In it, you'll be introduced to the contrastive language-image pretraining (CLIP) model, which lets you measure the cosine similarity between any image and its corresponding text prompt.

## Summary

- The Denoising Diffusion Probabilistic Models (DDPMs) we studied in chapters 5 and 6 use a randomized process and follow a Markovian process, meaning each step depends only on the immediate previous step. In contrast, a Denoising Diffusion Implicit Model (DDIM) takes a more direct and predictable approach. It uses a deterministic, non-Markovian process, allowing it to generate images in fewer steps without sacrificing quality.
- While a simple U-Net may be adequate for generating grayscale images due to the reduced complexity, generating high-definition color images requires a more sophisticated architecture to handle the increased dimensionality and capture complex spatial and color relationships.
- The advanced U-Net architecture in this chapter improves the model's ability to reconstruct fine details during denoising. However, adding more contracting and expansive blocks can lead to redundant feature extractions, making it challenging to identify the most relevant features. To mitigate this issue, the scaled dot-product attention (SDPA) mechanism is incorporated, allowing the model to focus on significant features while filtering out the less important ones.
- Once the denoising model is trained, we can generate high-resolution images by inputting an initial noise tensor into the U-Net. This noise tensor serves as a starting point that the model gradually refines into a clear and detailed image. Because the final image is influenced by the specific noise tensor used, different initial noise tensors will yield different results.
- If we start with two different initial noise tensors, we can generate an interpolated tensor by taking a weighted average of them. Feeding this interpolated tensor into the trained denoising U-Net produces an image that blends the characteristics of the two images generated from the original noise tensors.
- By varying the weights assigned to the two original noise tensors, we can create a series of interpolated initial noise tensors. Feeding these tensors into the trained denoising U-Net model generates a sequence of interpolated final images, displaying a smooth transition between the two original images.

## Part 3

# *Text-to-image generation with diffusion models*

**N**ow that you've mastered the basics of transformers and diffusion models, we'll show you how they come together for text-to-image generation in this part. In chapter 8, you'll learn how to build and train a contrastive language-image pretraining (CLIP) model from scratch. A trained CLIP model enables you to measure the similarity between a text prompt and an image. As a result, you can perform an image selection task: inputting a text description and using the model to identify the image from a large pool that best matches the text. These examples highlight just a few of the many real-world applications of text-to-image generation models and the valuable skill sets they offer.

In chapter 9, we move into latent diffusion, a more efficient version of diffusion models that generates images in compressed latent space rather than pixel space. This sets the stage for a deep dive into Stable Diffusion (in chapter 10), one of the most influential open source models in generative AI. By the end of this part, you'll have implemented the building blocks of modern diffusion pipelines and gained insight into how text prompts are translated into compelling, high-quality images.





# 8

## *CLIP: A model to measure the similarity between image and text*

---

### ***This chapter covers***

- Compressing a text description and an image into the same latent space
- Building and training a CLIP model to match text–image pairs
- Measuring text–image similarity
- Using the trained CLIP model to select an image based on a text prompt

State-of-the-art text-to-image models such as DALL-E 2, Google’s Imagen, and Stable Diffusion are built on three foundational components: (1) a text encoder to convert language into a latent representation, (2) a mechanism for injecting text information into the image-generation process, and (3) a diffusion model to generate realistic images from noise.

In previous chapters, we explored how diffusion models generate images and how to encode text information for machine learning. Now we turn to the key bridge that connects text and vision: understanding how a model can “see” an image through

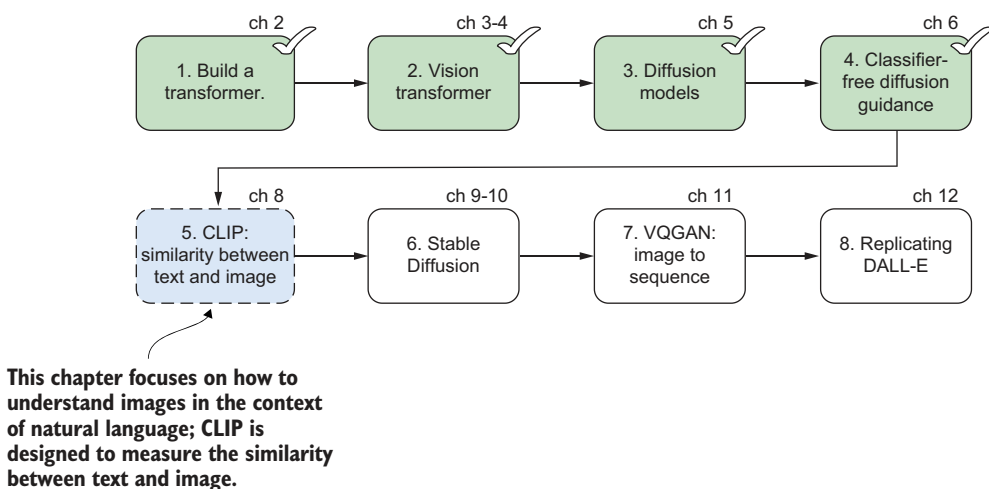
the lens of natural language. This is where contrastive language-image pretraining (CLIP) comes in.

Released by OpenAI in 2021, CLIP is a multimodal transformer that learns to align images and text in a shared latent space [1]. Unlike traditional models that rely on explicit image labels, CLIP uses enormous datasets of real-world image-caption pairs, making it incredibly effective for associating images and their textual descriptions.

This chapter guides you through the process of building a CLIP model from scratch and training it using the Flickr 8k dataset of image-caption pairs. You'll learn how contrastive learning allows CLIP to match images and captions by projecting both into a shared embedding space. You'll use your trained model to select the image that best matches a given text prompt, which can be viewed as "text-to-image selection." Finally, we'll explore OpenAI's pretrained CLIP model, which offers even broader generalization thanks to training on massive data.

Although CLIP doesn't generate new images from scratch, it plays an essential role in evaluating and steering text-to-image generators. By measuring how well an image matches a given caption, CLIP acts as a judge, and later, as a guide, for generative models.

Figure 8.1 lays out the eight steps for creating a text-to-image generator from scratch. This chapter focuses on step 5, understanding images in the context of natural language.



**Figure 8.1** Eight steps for building a text-to-image generator from scratch. In this chapter, we'll focus on step 5: enabling the model to understand images in the context of natural language. By mastering this step, you'll equip your models with the ability to align and compare images and texts, a capability that is foundational for all subsequent text-to-image generation methods.

CLIP's core innovation is its ability to compress both text and images into vectors in the same latent space. The closer these vectors are, the more closely the image matches

the text. During training, CLIP learns to pull together the vectors of matching image–caption pairs and push apart those of mismatched pairs. This powerful alignment lets us directly measure, compare, and retrieve the best-matching images for any description, without manual labeling.

In this chapter, you’ll see how a simple text prompt such as “a dog walks on the beach” can be used to select the most relevant photo from thousands in the Flickr 8k dataset. Later, this same similarity measure will become crucial for guiding image generation in advanced text-to-image models. In future chapters, you’ll see how CLIP’s ability to measure image–text similarity is woven into the very core of generative models such as Stable Diffusion, ensuring that generated images truly reflect the intent of your prompt.

## 8.1 The CLIP model

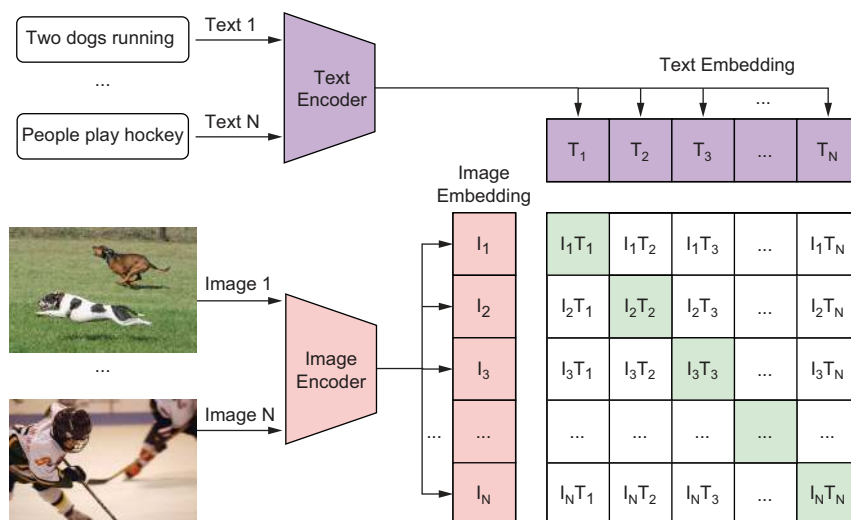
The CLIP model consists of two key components: a text encoder and an image encoder. The text encoder compresses the text description into a text embedding. The image encoder converts the corresponding image into an image embedding of the same dimension (e.g., 256 values). To train the model, we collect a large-scale training dataset of text–image pairs. During training, a batch of  $N$  text–image pairs is converted to  $N$  text embeddings and  $N$  image embeddings. CLIP uses a contrastive learning approach to maximize the similarity between paired embeddings while minimizing the similarity between embeddings from nonmatching text–image pairs. Once the model is trained, we’ll use it to select an image from the Flickr 8k dataset that best matches a given text prompt.

### 8.1.1 How the CLIP model works

In recent years, merging computer vision with natural language processing (NLP) has led to remarkable breakthroughs, one of the most notable being OpenAI’s CLIP model. The model is designed to interpret images within the context of natural language, opening up exciting applications such as image generation and classification.

CLIP uses a massive dataset of image–text pairs to develop a more general and versatile understanding of visual concepts. As mentioned earlier, at its core, the CLIP model features two main components: a text encoder and an image encoder. The text encoder transforms captions into a latent space, while the image encoder converts images into the same latent space. The training objective is to ensure that the embeddings for matching image–caption pairs are similar.

Figure 8.2 illustrates how the CLIP model is trained. First, we collect a large-scale dataset of text–image pairs. We’ll use the Flickr 8k dataset that you downloaded in chapter 4 as our training dataset. The text encoder in the CLIP model compresses each text description into a 256-dimensional embedding, and the image encoder does the same for the corresponding image. In each training batch of  $N$  pairs, CLIP employs a contrastive learning approach that maximizes similarity between matching pairs (the diagonal values in the figure) while reducing similarity between nonmatching pairs (the off-diagonal values).



**Figure 8.2** How the CLIP model is trained

Specifically, we'll use the trained DistilBERT model as our text encoder. DistilBERT is a smaller, faster, and lighter version of the bidirectional encoder representations from transformers (BERT) model, created through a process known as knowledge distillation. In this process, a "student" model (DistilBERT) learns to mimic the behavior of a larger "teacher" model (BERT), capturing most of its language understanding capabilities while significantly reducing the number of parameters.

We'll use ResNet50 as our image encoder in the CLIP model. ResNet50 is a deep convolutional neural network (CNN) that has 50 layers and was introduced in a paper by Kaiming He et al. in 2016 [2]. It's widely used for image classification, object detection, and many other computer vision tasks.

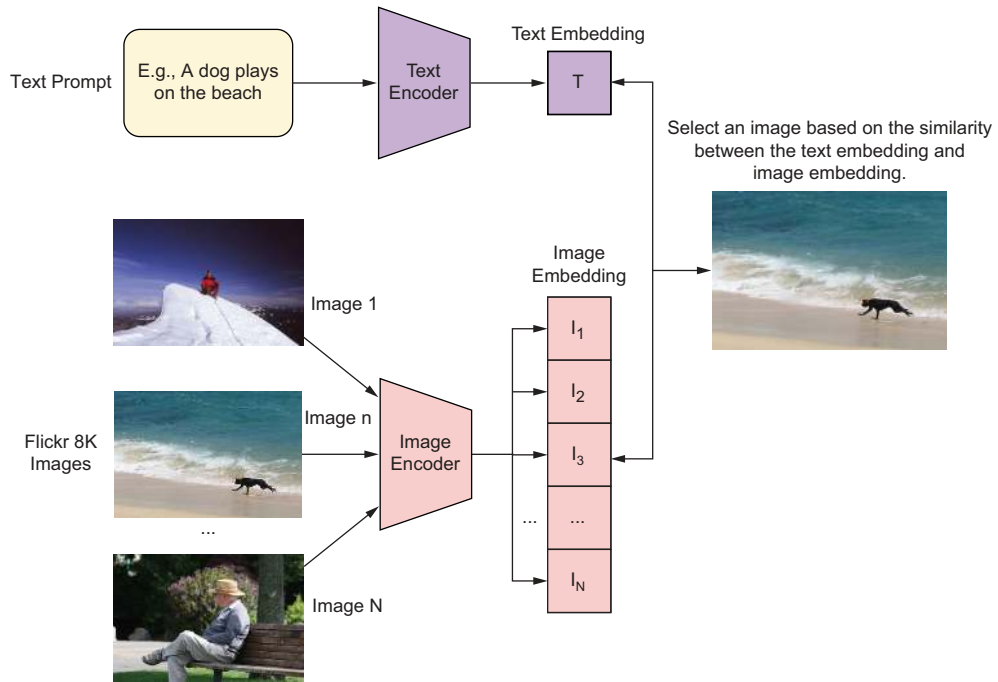
By combining DistilBERT and ResNet50 within the CLIP framework, we ensure that both text and image data are processed into a shared semantic space. The contrastive learning method then aligns these embeddings such that semantically related text and images are pulled together, while unrelated pairs are pushed apart. This approach improves the model's ability to perform cross-modal retrieval (selecting an image based on a text description or selecting a caption based on an image).

### 8.1.2 Selecting an image from Flickr 8k based on a text description

Later in this book, we'll use the pretrained CLIP model extensively in various text-to-image models. To demonstrate the usefulness of CLIP, you'll learn how to select an image from the Flickr 8k dataset based on a text description in this chapter.

Figure 8.3 is a diagram of the steps used to select an image from a dataset based on a text prompt. First, we use the text encoder in the CLIP model to convert the text prompt

into a text embedding (top left). For example, if the text prompt is “A dog plays on the beach,” the text encoder converts the text prompt into a 256-value text embedding. At the same time, we use the image encoder in the CLIP model to convert all images in the Flickr 8k dataset into  $N$  image embeddings (bottom left). The Flickr 8k dataset contains 8,091 images, and we convert each image into a 256-value image embedding.



**Figure 8.3** How to select an image from the Flickr 8k dataset using the trained CLIP model based on a text prompt. The text encoder in the CLIP model converts the prompt (e.g., “A dog plays on the beach,” shown at the top left) into a text embedding. Then, the image encoder processes every image in the dataset to generate  $N$  image embeddings. The similarity between the text embedding and each image embedding is then computed. Finally, the images are sorted by their similarity scores, and the one with the highest score is chosen as the match.

Next, we compute the similarity scores between the text embedding and all the image embeddings. Specifically, we’ll calculate the cosine similarity between the text embedding and each of the 8,091 image embeddings. Cosine similarity is a measure that calculates the similarity between two nonzero vectors by measuring the cosine of the angle between them. The value of cosine similarity ranges from  $-1$  to  $1$ , with  $1$  indicating that the vectors are identical in orientation and  $-1$  indicating opposite orientation. We choose cosine similarity because it measures the orientation (rather than magnitude) of vectors. In text analysis or image embeddings, the content similarity is important, not the absolute values.

Similarity scores help quantify how closely related each image is to the text description. A higher cosine similarity indicates that the image's content is more likely to match the text prompt. We sort the images based on the similarity scores and identify the image with the highest similarity score as the match. Alternatively, you can select more than one match. For example, you can select the five images with the top five similarity scores as the output.

## 8.2 Preparing the training dataset

You'll use the Flickr 8k dataset to train the CLIP model in this chapter. As you've learned in chapter 4, the Flickr 8k dataset is a collection of 8,091 photographs sourced from the Flickr website, each annotated with five different descriptive captions provided by human annotators. In this chapter, our purpose is to select images from the dataset based on a text prompt.

Let's walk through the steps in downloading the dataset, splitting them into a train set and test set and placing them in batches for training later. The Python program in this chapter is adapted from a great GitHub repository by Moein Shariatnia (<https://github.com/moein-shariatnia/OpenAI-CLIP>). Before we start, run the following line of code to install a few libraries that we'll use:

```
!pip install timm albumentations ftfy
```

**NOTE** The Python programs for all chapters in this book can be accessed from the GitHub repository at <https://github.com/markhliu/txt2img>. Ideally, you'll want a CUDA-enabled GPU to run the programs in this chapter. If you don't have one, you can follow along using the Google Colab notebook at <https://mng.bz/RwBv>.

### 8.2.1 Image-caption pairs in Flickr 8k

As we discussed in chapter 4, you can download the Flickr 8k images and the associated captions from the Kaggle website ([www.kaggle.com/datasets/adityajn105/flickr8k](http://www.kaggle.com/datasets/adityajn105/flickr8k)). Log in to your Kaggle account, download the zip file (if you haven't already done so), and extract the data from the zip file. Place the text file captions.txt and the folder /Images/ in the folder /files/ on your computer. We'll first use the pandas library to load the captions for these images as follows:

```
import pandas as pd

df=pd.read_csv(r'files/captions.txt',
               delimiter=",")
```

Make sure you use the `delimiter=","` argument in the `read_csv()` method in the pandas library because each row in the file captions.txt is separated into two columns by a comma (","). You can print out the first 12 observations as follows:

```
print(df.head(n=12))
```

The output is

```

                                image \
0  1000268201_693b08cb0e.jpg
1  1000268201_693b08cb0e.jpg
2  1000268201_693b08cb0e.jpg
3  1000268201_693b08cb0e.jpg
4  1000268201_693b08cb0e.jpg
5  1001773457_577c3a7d70.jpg
6  1001773457_577c3a7d70.jpg
7  1001773457_577c3a7d70.jpg
8  1001773457_577c3a7d70.jpg
9  1001773457_577c3a7d70.jpg
10 1002674143_1b742ab4b8.jpg
11 1002674143_1b742ab4b8.jpg

                                caption
0  A child in a pink dress is climbing up a set o...
1      A girl going into a wooden building .
2  A little girl climbing into a wooden playhouse .
3  A little girl climbing the stairs to her playh...
4  A little girl in a pink dress going into a woo...
5      A black dog and a spotted dog are fighting
6  A black dog and a tri-colored dog playing with...
7  A black dog and a white dog with brown spots a...
8  Two dogs of different breeds looking at each o...
9      Two dogs on pavement moving toward each other .
10 A little girl covered in paint sits in front o...
11 A little girl is sitting in front of a large p...

```

As you can see, each image has five text descriptions. Next, we can visualize 10 images with their shortest captions.

### Listing 8.1 Visualizing image–caption pairs in Flickr 8k

```

imgfolder=r"files/Images"
import os
with os.scandir(imgfolder) as fb:
    files=[f.name for f in fb]
start=100
imgs=files[start:start+10]
dfi=df[df["image"].isin(imgs)].copy()
dfi["length"]=dfi["caption"].str.len()
dfi=dfi.sort_values(['image','length'])
dfi=dfi.groupby("image").first()

import PIL
from matplotlib import pyplot as plt
plt.figure(dpi=200,figsize=(15,10))
for i in range(10):
    plt.subplot(5,2, i+1)
    img=f"{imgfolder}/{dfi.index[i]}"
    nparray=PIL.Image.open(img)

```

← Selects 10 images

← Selects the shortest caption for each image

← Plots the 10 images in five rows and two columns

```
plt.imshow(nparray)
plt.title(f"{dfi.iloc[i]['caption']}")
plt.axis("off")
plt.tight_layout()
plt.show()
```

← Places the caption on top of the image

We select 10 images from the Flickr 8k dataset. Because each image has five different captions, we select the shortest one for each image so that the caption can fit on top of each image. We plot the 10 images in a  $5 \times 2$  grid and set the caption as the title of each image. After running the code block, you'll see a plot of 10 images, as shown in figure 8.4. Next, we'll process the captions and images and get them ready for training.

A dog jumps to catch a Frisbee , while many people watch.



A small boy is playing with rocks.



A young man climbs a rocky hill.



A man is rock climbing.



A man climbs a rocky wall.



A child plays with a fountain.



A dog is running on the beach.



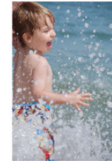
A black and white bird standing on a hand.



A dog is standing in the sand.



A little boy stands in the surf.



**Figure 8.4** Ten image-caption pairs from the Flickr 8k dataset. We select 10 images and place the shortest caption on top of each image.



### 8.2.2 The DistilBERT tokenizer

Because deep neural networks can't directly process raw text or image files, we'll first preprocess captions and images into representations that the CLIP model can accept as inputs. In particular, we'll use the DistilBERT tokenizer to convert the raw text in captions into tokens (parts of words) first. We'll then insert two special tokens, CLS and SEP, at the beginning and the end of each sentence. This is similar to what we've done in chapter 4 when training a model to add captions to images. Further, we'll pad shorter sentences in a batch with 0s so that all captions in a batch have the same length, similar to what we've done in chapter 2. To see how the DistilBERT tokenizer works, let's examine the following code listing.

**Listing 8.2 Tokenizing captions with the DistilBERT tokenizer**

```
from transformers import DistilBertTokenizer
tokenizer = DistilBertTokenizer.from_pretrained(
    "distilbert-base-uncased")
encoded=tokenizer(
    ["two dogs run",
     "an eagle flies in the sky"],
    padding=True, truncation=True,
    max_length=200)
print(encoded)
for indexes in encoded['input_ids']:
    tokens = tokenizer.convert_ids_to_tokens(indexes)
    print(tokens)
```

Imports the DistilBERT tokenizer from the transformers library

Generates input indexes and attention masks for a batch of two captions

Prints out the individual tokens in the two captions

The output is

```
{'input_ids': [[101, 2048, 6077, 2448, 102, 0, 0, 0],
               [101, 2019, 6755, 10029, 1999, 1996, 3712, 102]],
 'attention_mask': [[1, 1, 1, 1, 1, 0, 0, 0],
                    [1, 1, 1, 1, 1, 1, 1, 1]]}
['[CLS]', 'two', 'dogs', 'run', '[SEP]', '[PAD]', '[PAD]', '[PAD]']
['[CLS]', 'an', 'eagle', 'flies', 'in', 'the', 'sky', '[SEP]']
```

We use two hypothetical captions as our example: “two dogs run” and “an eagle flies in the sky.” We put the two captions as a batch and feed them to the DistilBERT tokenizer. The preceding output shows that the tokens for the first caption are ['[CLS]', 'two', 'dogs', 'run', '[SEP]', '[PAD]', '[PAD]', '[PAD]']. The CLS token signals the start of the caption, while the SEP token signals the end of the caption. Furthermore, because the first caption is shorter than the second one, three padding tokens, PAD, are added to the end of the sequence to make two sequences the same length. The tokens for the second caption are ['[CLS]', 'an', 'eagle', 'flies', 'in', 'the', 'sky', '[SEP]']. The special tokens CLS and SEP are inserted into the beginning and the end of the sequence. Because the second caption is the longest one in the batch, no padding token is added to the sequence.

The attention mask for the first caption is [1, 1, 1, 1, 1, 0, 0, 0], which tells the model to pay attention to the first five tokens while ignoring the remaining three padding tokens. The attention mask for the second caption is [1, 1, 1, 1, 1, 1, 1, 1], which tells the model to pay attention to all eight tokens because there's no padding token in the second sequence.

### 8.2.3 Preprocess captions and images for training

Now that you understand how the DistilBERT tokenizer works, let's preprocess the raw text and image data into PyTorch tensors so that we can use them to train the CLIP model later. We first split the dataset into train and validation subsets.

#### Listing 8.3 Creating train and validation subsets

```
import numpy as np
image_ids = np.arange(0, len(df))
np.random.seed(42)
valid_ids = np.random.choice(
    image_ids, size=int(0.2 * len(df)), replace=False)
train_ids = [id_ for id_ in image_ids
              if id_ not in valid_ids]
train = df[df.index.isin(train_ids)].reset_index(drop=True)
valid = df[df.index.isin(valid_ids)].reset_index(drop=True)
```

← Fixes the random state so results are reproducible

← Uses 20% of the sample as the validation set

← The remaining 80% of the sample is the train subset.

We use 20% of the sample as the validation subset and the remaining 80% of the sample as the train subset. To save space, we'll define various functions and classes for this chapter in a local module `CLIPUtil`. Download the file `CLIPUtil.py` from the book's GitHub repository (<https://github.com/markhliu/txt2img>), and place it in the folder `/utils/` on your computer (or simply clone the repository to your computer). If you're working in Google Colab, you can quickly set up the repository and add it to your working directory by running

```
!git clone https://github.com/markhliu/txt2img
import sys
sys.path.append("/content/txt2img")
```

In the local module `CLIPUtil.py`, we've defined a `CFG()` class to store all hyperparameters used in this chapter. The following listing shows this class.

#### Listing 8.4 Defining a `CFG()` class to store hyperparameters

```
class CFG:
    image_path = r"files/Images"
    captions_path = r"files"
    batch_size = 32
    head_lr = 1e-3
    weight_decay = 1e-3
    patience = 1
    factor = 0.8
```

```

device = "cuda" if torch.cuda.is_available() else "cpu"
model_name = 'resnet50'
image_embedding = 2048
text_encoder_model = "distilbert-base-uncased"
text_embedding = 768
text_tokenizer = "distilbert-base-uncased"
max_length = 200
pretrained = True
trainable = False
temperature = 1.0
size = 224
num_projection_layers = 1
projection_dim = 256
dropout = 0.1

```

Uses the pretrained ResNet50 model to process images

Uses the pretrained DistilBERT model to encode captions

Converts all images to a size of  $3 \times 224 \times 224$

Both text embeddings and image embeddings are 256-value vectors.

We'll use the pretrained ResNet50 and DistilBERT models to encode images and captions, respectively. All images will be converted to a standard size of  $3 \times 224 \times 224$ . Both text embeddings and image embeddings are 256-value vectors. The `CFG()` class has many other parameters that we'll use later, and we'll explain them as we go along.

In the local module, we've also defined a `CLIPDataset()` class to convert raw text and image data in Flickr 8k into tensors. The following listing shows the necessary code.

#### Listing 8.5 Defining the `CLIPDataset()` class

```

class CLIPDataset(torch.utils.data.Dataset):
    def __init__(self, image_filenames, captions, tokenizer,
                 transforms):
        self.image_filenames = image_filenames
        self.captions = list(captions)
        self.encoded_captions = tokenizer(
            list(captions), padding=True, truncation=True,
            max_length=CFG.max_length)
        self.transforms = transforms
    def __getitem__(self, idx):
        item = {key: torch.tensor(values[idx])
                for key, values in self.encoded_captions.items()}
        image = cv2.imread(
            f"{CFG.image_path}/{self.image_filenames[idx]}")
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        image = self.transforms(image=image)['image']
        item['image'] = torch.tensor(image).permute(
            2, 0, 1).float()
        item['caption'] = self.captions[idx]
        return item
    def __len__(self):
        return len(self.captions)

```

Uses the DistilBERT tokenizer to convert captions to tensors

Converts raw image data to tensors

Each item consists of an image-caption pair.

The `CLIPDataset()` class uses the DistilBERT tokenizer to convert captions into tokens and then sequences of indexes, with each index representing a token. The class also

converts the corresponding image from the raw image data into a standard shape of (3, 224, 224)—three color channels, with a height and width of 224 pixels.

In the following listing, we apply the `CLIPDataset()` class on both the train and validation subsets. We also place them in batches, with a batch size of 32.

#### Listing 8.6 Creating data loaders

```
from transformers import (DistilBertModel, DistilBertConfig,
                          DistilBertTokenizer)
from utils.CLIPUtil import get_transforms, CLIPDataset, CFG
import torch

tokenizer = DistilBertTokenizer.from_pretrained(
    CFG.text_tokenizer)
transforms = get_transforms()
trainset = CLIPDataset(train["image"].values,
                       train["caption"].values, tokenizer=tokenizer,
                       transforms=transforms)
trainloader = torch.utils.data.DataLoader(trainset,
                                           batch_size=CFG.batch_size, shuffle=True)
valset = CLIPDataset(valid["image"].values,
                     valid["caption"].values, tokenizer=tokenizer,
                     transforms=transforms)
valloader = torch.utils.data.DataLoader(valset,
                                         batch_size=CFG.batch_size, shuffle=False)
```

Instantiates the DistilBERT tokenizer

Applies the CLIPDataset() class on the train subset and creates a data loader

Applies the CLIPDataset() class on the validation subset and creates a data loader

We apply the `CLIPDataset()` class on both the train and validation subsets and create two data loaders. We'll use the data loaders to train and test the CLIP model later in the chapter. To get a sense of what the training data looks like, we can retrieve the first batch in the train loader and print out the components in the data:

```
batch0=next(iter(trainloader))
print(batch0.keys())
```

The output is

```
dict_keys(['input_ids', 'attention_mask', 'image', 'caption'])
```

Each batch is a Python dictionary with four key–value pairs. The keys are `input_ids`, `attention_mask`, `image`, and `caption`. We can print out the value or the shape of the components in the first observation as follows:

```
print(batch0['input_ids'][0])
print(batch0['attention_mask'][0])
print(batch0['image'][0].shape)
print(batch0['caption'][0])
```

The output is

```

tensor([ 101, 1037, 2158, 2003, 5362, 12701, 2091, 1037,
4586, 3139, 2940, 4147, 1037, 13383, 1012, 102, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
torch.Size([3, 224, 224])
A man is carefully skiing down a snow covered hill wearing a backpack .

```

The `input_ids` for a caption is a sequence of integers. The sequence is padded with 0 at the end to ensure that all sequences in the batch have the same length. The `attention_mask` tensor masks out the paddings so the model pays attention only to the indexes corresponding to the captions, not to the padded 0s at the end of the sequence. Each image is now represented by a tensor with a size of (3, 224, 224). The original caption is also included in the dictionary, which says, “A man is carefully skiing down a snow-covered hill wearing a backpack.” Note that because data batches are randomly shuffled, your result is likely different. Next, we’ll create a CLIP model and use the data to train the model so that it can match captions to images.

### 8.3 Creating a CLIP model

Now that we have the training data ready, we’ll move on to creating a CLIP model. The model consists of a text encoder and an image encoder that work together to convert an image and its corresponding captions into two 256-dimensional vectors. This transformation lays the groundwork for training the model to align these two representations. In the following section, we’ll use the prepared data to train the model to associate captions with images. Once training is complete, we can input any text prompt, and the CLIP model will retrieve the image that best matches the description.

#### 8.3.1 Creating a text encoder

The text encoder in our CLIP model converts sequences of indexes, which represent tokens in the text descriptions of the images, into embeddings in the latent space. Later, we’ll create an image encoder that converts an image into an embedding of the same length in the same latent space. In the local module `CLIPutil.py`, we define the `TextEncoder()` class, as shown in the following listing.

##### Listing 8.7 Creating a text encoder

```

class TextEncoder(nn.Module):
    def __init__(self, model_name=CFG.text_encoder_model,
                  pretrained=CFG.pretrained,
                  trainable=CFG.trainable):
        super().__init__()
        if pretrained:
            self.model=DistilBertModel.from_pretrained(model_name)
        else:
            self.model=DistilBertModel(config=DistilBertConfig())
        for p in self.model.parameters():

```

```

        p.requires_grad = trainable
        self.target_token_idx = 0
    def forward(self, input_ids, attention_mask):
        output = self.model(input_ids=input_ids,
                             attention_mask=attention_mask)
        last_hidden_state = output.last_hidden_state
        return last_hidden_state[:,
                                   self.target_token_idx:]

```

← **Focuses only on the first token in the caption (the starting token CLS)**

← **Uses the last layer of the pretrained DistilBERT model**

← **The text embedding is the output associated with the CLS token in the last layer.**

In chapter 2, when constructing a vision transformer (ViT), we inserted a CLS token into the image token sequence. This token isn't related to any particular part of the image; instead, it aggregates information from the entire image for classification. We adopt a similar strategy when creating a text embedding for a caption in this chapter. Here, the CLS token isn't tied to an individual word in the caption but serves as a summary of the entire caption. Consequently, we focus only on the first token, the CLS token.

Of the many different layers in the DistilBERT model, we'll use the last layer. As input flows through different layers, the representations become more abstract and refined. The final layer has integrated information from all previous layers. Therefore, the text embedding for the caption is the output associated with the CLS token in the very last layer.

### The CLS token in the last layer: A learned summary of the entire caption

The text embedding of a caption is created by extracting the embedding from the CLS token at the very last layer of the DistilBERT model. The CLS token is trained to capture the overall meaning of the input. It acts as a learned summary, aggregating contextual information from all other tokens through self-attention. It provides a compact, fixed-size vector that is well-suited for aligning with image embeddings in CLIP.

As input flows through the multiple layers of DistilBERT, the representations become progressively more abstract and contextualized. The final layer contains the most refined features because it has integrated information from all previous layers. Using any earlier layer might mean missing out on some of the high-level contextualization that the full network has learned to provide.

Instead of training a text encoder from scratch, we'll use the pretrained DistilBERT model from Hugging Face to reduce the training time. We can count the number of trainable and untrainable parameters in the text encoder defined earlier:

```

from utils.CLIPutil import TextEncoder

textencoder=TextEncoder()
num_trainable = sum([p.numel() for p in
    textencoder.parameters() if p.requires_grad])
print(f"Number of trainable parameters: {num_trainable}")

```

```
non_trainable = sum([p.numel() for p in
    textencoder.parameters() if not p.requires_grad])
print(f"Number of untrainable parameters: {non_trainable}")
```

The output is as follows:

```
Number of trainable parameters: 0
Number of untrainable parameters: 66362880
```

The DistilBERT model has more than 66 million parameters. By turning off the trainable argument, we can speed up the training process.

An additional benefit of using the embedding of the CLS token in the very last layer as the text embedding for the whole caption is the fixed-length output. No matter what the length of the caption is, the size of the text embedding is the same.

You can apply the text encoder on the first batch of training data:

```
encoded_text=textencoder(batch0['input_ids'],
                        batch0['attention_mask'])
print(encoded_text.shape)
```

The output is

```
torch.Size([32, 768])
```

We feed the input indexes and attention masks in the first batch to the text encoder, and the output has a shape of (32, 768). There are 32 captions in the batch, and each caption is converted into a 768-value vector. Later, we'll pass this 768-value vector through a projection head to generate a 256-value vector, which will be our final text embedding.

### 8.3.2 Creating an image encoder

We'll use the ResNet50 model from the timm library as our image encoder. In the local module CLIPutil.py, we define the ImageEncoder() class:

```
class ImageEncoder(nn.Module):
    def __init__(self, model_name=CFG.model_name,
                 pretrained=CFG.pretrained,
                 trainable=CFG.trainable):
        super().__init__()
        self.model = timm.create_model(model_name, pretrained,
                                       num_classes=0, global_pool="avg")
        for p in self.model.parameters():
            p.requires_grad = trainable
    def forward(self, x):
        return self.model(x)
```

Uses ResNet50 as the image encoder

Uses the pretrained model instead of training from scratch

Sets the trainable argument to False

We'll use the pretrained weights to reduce the number of trainable parameters and eventually the training time. To count the number of trainable and untrainable parameters in the image encoder defined previously, run the following code cell:

```
from utils.CLIPUtil import ImageEncoder

imageencoder=ImageEncoder()
num_trainable = sum([p.numel() for p in imageencoder.parameters()
                     if p.requires_grad])
print(f"Number of trainable parameters: {num_trainable}")
non_trainable = sum([p.numel() for p in imageencoder.parameters()
                     if not p.requires_grad])
print(f"Number of untrainable parameters: {non_trainable}")
```

The output is

```
Number of trainable parameters: 0
Number of untrainable parameters: 23508032
```

The ResNet50 model has more than 23 million parameters. By turning off the trainable argument, we can speed up the training process.

You can apply the image encoder on the first batch of the training data and print out the shape of the output:

```
encoded_image=imageencoder(batch0['image'])
print(encoded_image.shape)
```

The output is

```
torch.Size([32, 2048])
```

We feed the images in the first batch to the image encoder, and the output has a shape of (32, 2048). There are 32 images in the batch, and each image is converted into a 2,048-value vector. Later, we'll pass each vector through a projection head to produce a 256-value image embedding.

### 8.3.3 Building a CLIP model

You may have noticed that each caption is converted to a 768-value vector, while each image is converted to a 2,048-value vector. To make the dimensions of the text embedding and image embedding the same, we define the following ProjectionHead() class in the local module.

**Listing 8.8** Defining a projection head

```
class ProjectionHead(nn.Module):
    def __init__(self, embedding_dim,
                 projection_dim=CFG.projection_dim,
                 dropout=CFG.dropout):
```

← The dimension of the input (768 for text and 2,048 for image)

← The dimension of the output (256 for both text and image)



```

super().__init__()
self.projection = nn.Linear(embedding_dim,
                             projection_dim)

self.gelu = nn.GELU()
self.fc = nn.Linear(projection_dim, projection_dim)
self.dropout = nn.Dropout(dropout)
self.layer_norm = nn.LayerNorm(projection_dim)
def forward(self, x):
    projected = self.projection(x)
    x = self.gelu(projected)
    x = self.fc(x)
    x = self.dropout(x)
    x = x + projected
    x = self.layer_norm(x)
    return x

```

← The input goes through the two linear layers, GELU activation, and a dropout layer, with residual connection and layer normalization.

The `embedding_dim` argument in the `ProjectionHead()` class specifies the dimension of the input, which is 768 for text and 2,048 for image. The `projection_dim` argument is the dimension of the output, which is 256 for both text and image. We'll instantiate a text projection head and an image projection head later. The projection heads convert both text features and image features into 256-value vectors so that we can train the CLIP model to match the two embeddings, that is, to make the values in the two embeddings as close to each other as possible.

The CLIP model aligns images and texts by mapping both modalities into a shared embedding space, enabling direct comparison between image and text representations. The CLIP model is defined in the local module as shown in the following listing.

#### Listing 8.9 Creating the CLIP model

```

class CLIPModel(nn.Module):
    def __init__(self, temperature=CFG.temperature,
                 image_embedding=CFG.image_embedding,
                 text_embedding=CFG.text_embedding):
        super().__init__()
        self.image_encoder = ImageEncoder()
        self.text_encoder = TextEncoder()
        self.image_projection = ProjectionHead(
            embedding_dim=image_embedding)
        self.text_projection = ProjectionHead(
            embedding_dim=text_embedding)
        self.temperature = temperature
    def forward(self, batch):
        image_features = self.image_encoder(batch["image"])
        text_features = self.text_encoder(
            input_ids=batch["input_ids"],
            attention_mask=batch["attention_mask"])
        image_embeddings = self.image_projection(
            image_features)
        text_embeddings = self.text_projection(
            text_features)
        logits = \

```

← Uses the image encoder and text encoder to extract features

← Uses projection heads to generate image and text embeddings

```

(text_embeddings @ image_embeddings.T) / self.temperature
images_similarity = image_embeddings @ image_embeddings.T
texts_similarity = text_embeddings @ text_embeddings.T
targets = F.softmax(
    (images_similarity +
     texts_similarity) / 2 * self.temperature,
    dim=-1)
texts_loss = cross_entropy(logits, targets,
                           reduction='none')
images_loss = cross_entropy(logits.T, targets.T,
                             reduction='none')
loss = (images_loss + texts_loss) / 2.0
return loss.mean()

```

**Calculates the loss to maximize the similarity between image and text embeddings**

The model encodes each image and each caption (text) into a fixed-length vector (size 256). For a batch of 32 image–text pairs, both `image_embeddings` and `text_embeddings` tensors have shape (32, 256).

The main similarity matrix is computed as `text_embeddings@image_embeddings.T`, resulting in a (32, 32) matrix. Each element (i,j) measures how similar the i-th text is to the j-th image. This is the primary comparison used for training.

The model also uses two other matrices, `image_embeddings@image_embeddings.T` and `text_embeddings@text_embeddings.T`, which are `images_similarity` and `texts_similarity`. They measure how similar each embedding is to every other embedding of the same type within the batch (image-to-image and text-to-text, respectively).

This step isn't used for the main matching objective, but rather for target smoothing: by averaging the image-to-image and text-to-text similarities, we produce softer, more informative targets for the cross-entropy loss instead of only relying on strict one-hot labels. This can help the model learn more robust and meaningful representations by recognizing that some images (or texts) in a batch may be semantically similar, even if they aren't the exact match.

The soft targets are created by taking the average of the image and text similarity matrices and applying a softmax (with temperature scaling). This approach encourages the model to distribute some probability mass to similar (but not identical) examples, making the loss less brittle.

Cross-entropy loss is computed for both text-to-image and image-to-text directions and then averaged. This symmetric approach ensures both encoders are trained equally to align the representations.

The model is trained so that the embeddings of matching image–text pairs are close together and nonmatching pairs are far apart. The image-to-image and text-to-text similarity computations are used to soften the target labels and improve training, not as the primary measure of performance.

## 8.4 Training and using the CLIP model

Now that we have both the training data and the CLIP model, we'll train the model to match images with captions. After that, we'll use the trained model to select an image from the Flickr 8k dataset based on a given text prompt.

### 8.4.1 Training the CLIP model

We first import the `CLIPModel()` class from the local module and instantiate the class to create our CLIP model:

```
from utils.CLIPUtil import CLIPModel, CFG
model = CLIPModel().to(CFG.device)
num_trainable = sum([p.numel() for p in model.parameters()
                    if p.requires_grad])
print(f"Number of trainable parameters: {num_trainable}")
non_trainable = sum([p.numel() for p in model.parameters()
                    if not p.requires_grad])
print(f"Number of untrainable parameters: {non_trainable}")
```

Imports the `CLIPModel()` class from the local module

Instantiates the `CLIPModel()` to create our CLIP model

Prints out the number of trainable and untrainable parameters

The output is

```
Number of trainable parameters: 854016
Number of untrainable parameters: 89870912
```

Because we use pretrained weights from the DistilBERT model for text encoding and the ResNet50 model for image encoding, our model has only 854,016 trainable parameters.

Here, we define the optimizer and the learning rate scheduler:

```
import itertools

params = [
    {"params": itertools.chain(
        model.image_projection.parameters(),
        model.text_projection.parameters()
    ), "lr": CFG.head_lr, "weight_decay": CFG.weight_decay}
]
optimizer = torch.optim.AdamW(params, weight_decay=0.)
lr_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode="min", patience=CFG.patience,
    factor=CFG.factor
)
```

We'll train the model for 10 epochs. We test the performance of the model after each epoch and save the best model based on model performance on the validation set. Therefore, we define a `validation()` function first, as shown in the following listing.

#### Listing 8.10 Defining an `evaluate()` function to assess performance

```
def evaluate():
    global best_loss
    model.eval()
```

```

losses = []
with torch.no_grad():
    tqdm_object = tqdm(valloader, total=len(valloader))
    for batch in tqdm_object:
        batch = {k: v.to(CFG.device) for k, v in
            batch.items() if k != "caption"}
        loss = model(batch)
        losses.append(loss.item())
    avgloss=sum(losses)/len(losses)
    tqdm_object.set_description(
        f"valid_loss={avgloss:.2f}")
if avgloss < best_loss:
    best_loss = avgloss
    torch.save(model.state_dict(), "files/best.pth")
    print("Saved Best Model!")

```

Iterates through all batches in the validation set

Calculates the average loss in the epoch

If the performance of the model is the best among all epochs, saves the model parameters

The `evaluate()` function measures the model's performance using the validation set. It processes each batch in the set, computing the loss for each. The epoch's average loss is then calculated by averaging the losses from all batches. If the model performance in this epoch is better than that in all previous epochs, the function saves the model parameters to the `/files/` folder on your computer. We can now go ahead and train the model.

#### Listing 8.11 Training the CLIP model

```

scaler = torch.amp.GradScaler("cuda")
best_loss = float('inf')
from tqdm import tqdm
for epoch in range(10):
    print(f"Epoch: {epoch + 1}")
    model.train()
    losses = []
    tqdm_object = tqdm(trainloader, total=len(trainloader))
    for batch in tqdm_object:
        batch = {k: v.to(CFG.device) for k, v in
            batch.items() if k != "caption"}
        with torch.amp.autocast("cuda"):
            loss = model(batch)
            optimizer.zero_grad()
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()
            losses.append(loss.item())
        avgloss=sum(losses)/len(losses)
        tqdm_object.set_description(f"loss is {avgloss:.5f}")
    evaluate()
    lr_scheduler.step(avgloss)

```

Iterates through all batches in the training set

Calculates the loss in the batch

Adjusts model parameters to minimize the loss

Evaluates the current model and saves model parameters if it achieves the best performance so far

The best model is achieved after eight epochs of training. The trained model is saved as `best.pth` in the `/files/` folder on your computer. Alternatively, you can download

the trained model parameters on my Google Drive at <https://mng.bz/26Va>. Unzip the file after downloading.

### 8.4.2 Using the trained CLIP model to select images

Next, we use the trained CLIP model to select an image from the validation set based on a text prompt. Here, we load the trained model parameters:

```
sd=torch.load(r"files/best.pth",weights_only=True,map_location=device)
model.load_state_dict(sd)
model.eval()
```

We first obtain the image embeddings of all images in the validation set and use them as potential candidates for a match for any text prompt:

```
image_embeds = []
with torch.no_grad():
    for batch in tqdm(valloader):
        image_features = model.image_encoder(
            batch["image"].to(CFG.device))
        image_embeds.append(
            model.image_projection(image_features))
image_embeddings = torch.cat(image_embeds)
```

We save the image embeddings on the computer so that we don't need to create image embeddings again later when we perform more image-selection tasks:

```
import pickle

with open("files/image_embeds.p","wb") as f:
    pickle.dump(image_embeddings, f)
```

We define a `match()` function to select an image based on a prompt in the following listing.

#### Listing 8.12 Selecting a Flickr 8k image based on a text prompt

```
def match(prompt):
    encoded = tokenizer([prompt])  # ← Tokenizes the prompt
    batch = {
        key: torch.tensor(values).to(CFG.device)
        for key, values in encoded.items()
    }
    with torch.no_grad():
        text_features = model.text_encoder(
            input_ids=batch["input_ids"],
            attention_mask=batch["attention_mask"]
        )
        text_embeddings = model.text_projection(  # ← Creates a text embedding
            text_features)
    dot_similarity = text_embeddings @ image_embeddings.T
```

```

values, idx = torch.topk(dot_similarity.squeeze(0),
                          1)
img=valid['image'].values[idx.item()]
caption=valid['caption'].values[idx.item()]
return img, caption

```

← Selects the image with the highest value of similarity with the text prompt

The `match()` function takes a prompt as the input. It first converts the text into tokens and then uses the trained CLIP model to convert the text prompt into a text embedding. Next, the function calculates the cosine similarities between the text embedding and all the image embeddings. The image with the highest similarity with the text prompt is selected as the match. We can try the `match()` function using the prompt “students having a class in the classroom”:

```

prompt="students having a class in the classroom"
file,cap=match(prompt)
plt.imshow(PIL.Image.open(
    rf"files/Images/{file}"))
plt.title(f"Prompt: {prompt}\nOriginal caption: {cap}")
plt.axis("off")
plt.show()

```

← Finds a match for the prompt

← Displays the selected image, the prompt, and the original caption

The `match()` function returns the name of the matched image file and the original caption associated with the selected image. We display the selected image and place both the text prompt and the original caption above the image. The output is shown in figure 8.5.

The selected image does show a teacher and several students in a classroom. The original caption of the image is “a woman helps boys on a computer.” Even though the prompt isn’t the same as the original caption, the selection matches the description of the text prompt. This indicates that we’ve successfully trained a CLIP model.

Prompt: Students having a class in the classroom  
Original caption: A woman helps boys on a computer.



**Figure 8.5** Selecting an image from the Flickr 8k dataset using the trained CLIP model. The prompt used to select the image is “students having a class in the classroom.” The original caption of the image is “A woman helps boys on a computer.”

### Exercise 8.1

Use the `match()` function to select an image from the Flickr 8k dataset based on the text prompt “dogs walk on the beach.” Plot the selected image, and place both the text prompt and the original caption above the image.

### 8.4.3 Using the OpenAI pretrained CLIP model to select images

OpenAI has made its pretrained CLIP model public. Later in this book, we'll rely mainly on that model because it's trained on more text–image pairs. Therefore, the model can provide a more generalizable representation of visual concepts.

In this section, we'll enter a text prompt and use the OpenAI pretrained CLIP model to search for an image from the Flickr 8k database to match the text description. We'll first use the OpenAI pretrained CLIP model to encode all images in Flickr 8k. To do that, we'll clone the OpenAI GitHub repository and place it in the local folder so that we can use the OpenAI CLIP model.

Make sure you have the Git app installed on your computer. If not, follow the instructions at <https://mng.bz/15Vq> to install Git based on your operating system. Once you have Git installed on your computer, run the following line of code in a cell in Jupyter Notebook:

```
!git clone https://github.com/openai/CLIP
```

Now all the files related to the OpenAI CLIP model are placed in the CLIP folder on your computer. We'll use the sys library to give Python access to the folder as follows:

```
import sys

sys.path.append("./CLIP")
```

Next, we encode all images in the Flickr 8k dataset using the image encoder in the OpenAI CLIP model.

#### Listing 8.13 Encoding all images in the Flickr 8k dataset

```
import os

folder=r"files/Images"
with os.scandir(folder) as files:
    names=[file.name for file in files]

import torch
import clip
from PIL import Image

device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocess = clip.load("ViT-B/32",
                              device=device)
images=[]
for i in names:
    images.append(preprocess(Image.open(f"{folder}/{i}")
                           ).unsqueeze(0).to(device))
image=torch.cat(images)
print(image.shape)
with torch.no_grad():
```

← Loads the pretrained OpenAI CLIP model

```

image_features = model.encode_image(image)
with open("files/imgfeas.p", "wb") as fb:
    pickle.dump(image_features, fb)

```

← Encodes all images in Flickr 8k

← Saves the image embeddings for later use

We first load the pretrained OpenAI CLIP model. We then use the image encoder in the model to convert all images in Flickr 8k to image embeddings, which are saved in the file `imgfeas.p` on your computer for later use. Next, we define a `find_match()` function to find and display five images that best match a text prompt.

#### Listing 8.14 Finding image matches for a text prompt

```

def find_match(prompt):
    print(f"prompt is {prompt}")
    text = clip.tokenize([prompt]).to(device)
    with torch.no_grad():
        text_features = model.encode_text(text)
    simu = text_features @ image_features.T
    values, indices = torch.topk(simu[0], 5)
    plots = []
    for i in indices:
        plots.append(names[i])
    plt.figure(dpi=200, figsize=(10, 2))
    for i in range(5):
        plt.subplot(1, 5, i+1)
        img = f"{folder}/{plots[i]}"
        nparray = PIL.Image.open(img)
        plt.imshow(nparray)
        plt.axis("off")
    plt.tight_layout()
    plt.show()

```

← Encodes the text prompt into a text embedding

← Calculates the similarities between the text and all images

← Selects the best-matched five images based on the similarities

← Plots the five images in a 1 × 5 grid

The function tokenizes the prompt and uses the pretrained OpenAI CLIP model to convert the text into a text embedding. We then calculate the similarities between the text embedding and all the image embeddings that we saved earlier. We select five matches based on the highest similarity scores. We display the five matches in a 1 × 5 grid, with the left image the most likely match. Now we can load the image embeddings and call the `find_match()` function to find matches:

```

with open("files/imgfeas.p", "rb") as fb:
    image_features = pickle.load(fb)
find_match("people eating at the restaurant")

```

We entered the prompt "people eating at the restaurant" as the argument for the function. The five matched images are displayed in figure 8.6.

The last image in figure 8.6 fits the text prompt well. Note that our dataset is small. If we apply the same method to a large dataset, say, with millions of images with diverse settings, the match will be much closer to the text description.





**Figure 8.6** Five matched images based on the prompt “people eating at the restaurant” using the pretrained OpenAI CLIP model

### Exercise 8.2

Use the `find_match()` function to select an image from the Flickr 8k dataset based on the text prompt “people shopping on the street.” Plot the selected images in a  $1 \times 5$  grid.


In the next chapter, you’ll combine the CLIP model with the diffusion model to generate images based on text prompts. The image-generation process starts with an image that is entirely noise. The pretrained diffusion model then iteratively denoises this image. As the image is denoised, we use the pretrained CLIP model to guide this process to ensure that the denoised image matches the given text prompt.

### Summary

- The contrastive language-image pretraining (CLIP) model is designed to understand and interpret images in the context of natural language. The model consists of a text encoder and an image encoder. The text encoder compresses the text description into a text embedding. The image encoder converts the corresponding image into an image embedding of the same dimension. During training, a batch of  $N$  text–image pairs are converted to  $N$  text embeddings and  $N$  image embeddings. CLIP uses a contrastive learning approach to maximize the similarity between paired embeddings while minimizing the similarity between embeddings from nonmatching text–image pairs.
- When creating a CLIP model from scratch, we can use the pretrained DistilBERT model to encode image captions. DistilBERT adds `CLS` and `SEP` tokens to the beginning and end of each caption. The text embedding of a caption is created by extracting the embedding from the `CLS` token at the very last layer of the DistilBERT model. The `CLS` token is trained to capture the overall meaning of the entire caption. It acts as a learned summary, aggregating contextual information from all other tokens through self-attention.
- When training the CLIP model, we compute two cross-entropy losses: one where the correct image should stand out among all images for a given text (text-to-image matching), and another where the correct text should be distinguished from all texts for a given image (image-to-text matching). Averaging these two

losses creates a symmetric objective that ensures both the text encoder and image encoder are equally optimized to produce aligned, comparable representations.

- A trained CLIP model can measure the cosine similarity between a text description and any given image. This has many practical applications in computer vision tasks such as image generation and image classification.
- One application of the CLIP model is image selection. We can use the trained CLIP model to select an image from a large dataset (e.g., Flickr 8k) that best matches a text description.



# *Text-to-image generation with latent diffusion*

---

## ***This chapter covers***

- Conducting forward and reverse diffusion processes in the lower-dimensional space
- Converting low-resolution images into high-resolution ones
- Generating high-resolution images based on text prompts using a latent diffusion model
- Modifying existing images based on text prompts

As we've explored so far, diffusion models can generate strikingly realistic images from random noise by gradually reversing a noising process. When conditioned on text prompts, these models become powerful text-to-image generators, capable of creating images that match detailed, open-ended descriptions. But there's a challenge: generating high-resolution images directly with diffusion models is computationally demanding, often requiring tens or hundreds of millions of pixel-level calculations over thousands of steps for every single image.

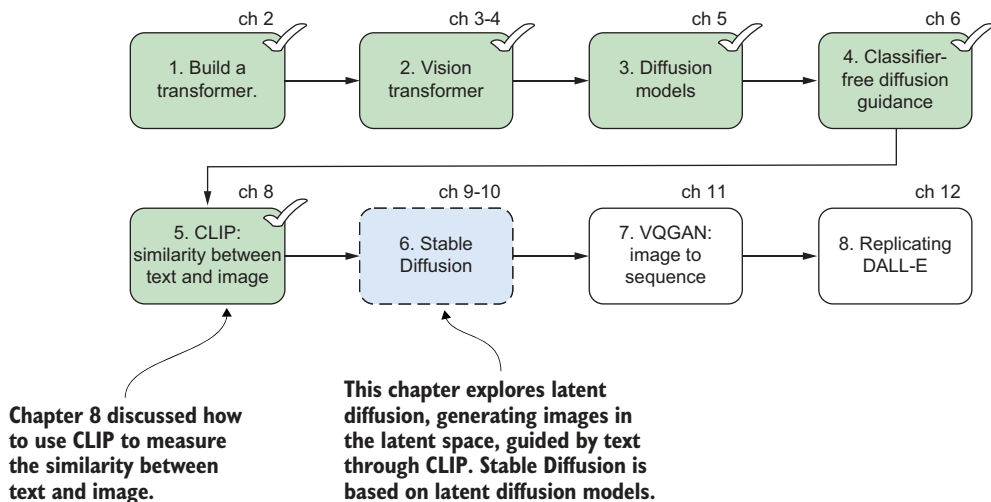
How do state-of-the-art text-to-image generators overcome this barrier to produce high-quality results efficiently? The answer lies in *latent diffusion models* (LDMs), introduced by Rombach et al. in 2022 [1]. LDMs move the heavy lifting away from the pixel space and into a compact, learned *latent space*, dramatically reducing both the memory footprint and computational cost of training and generation.

Instead of performing the forward and reverse diffusion processes on full-sized high-resolution images (e.g.,  $512 \times 512$  RGB images), LDMs conduct diffusion processes in a much smaller latent representation. For example, a  $3 \times 512 \times 512$  image might be mapped to a latent space of just  $4 \times 64 \times 64$ , a 48-fold reduction in data size, which enables far more efficient model training and inference.

Once an image is generated in the latent space, the model uses a variational auto-encoder (VAE) to decode the latent image into a detailed, high-resolution image in the pixel space. This architecture preserves the essential content and structure of the generated image, while offloading the fine-grained visual details to the powerful VAE decoder.

LDMs can edit existing images based on text descriptions as well. For instance, starting with an image of a white horse in the snow, you can use LDMs to create a version of the horse on a sunny beach, or even transform the horse into a zebra while preserving the background.

Figure 9.1 lays out where this chapter fits into your journey of building a text-to-image generator. It serves as a precursor to step 6 because one of the state-of-the-art text-to-image generators, Stable Diffusion, is based on LDMs.



**Figure 9.1** Eight steps for building a text-to-image generator from scratch. This chapter lays the groundwork for step 6, Latent Diffusion, showing you how to generate images in a lower-dimensional latent space, guided by text using the contrast language-image pretraining (CLIP) model from the previous chapter.

This chapter shows you how to compress high-resolution images into a latent space using a VAE and how to perform diffusion there for speed and efficiency. You'll gain a deep understanding of how a VAE enables high-quality reconstruction of images from compact latent representations. We'll dive into the mechanics of generating and editing images with an LDM, conditioned on text prompts. The contrast language-image pretraining (CLIP) model we discussed in chapter 8 guides the image generation in the latent space, ensuring that generated images match text descriptions.

## 9.1 What is a latent diffusion model?

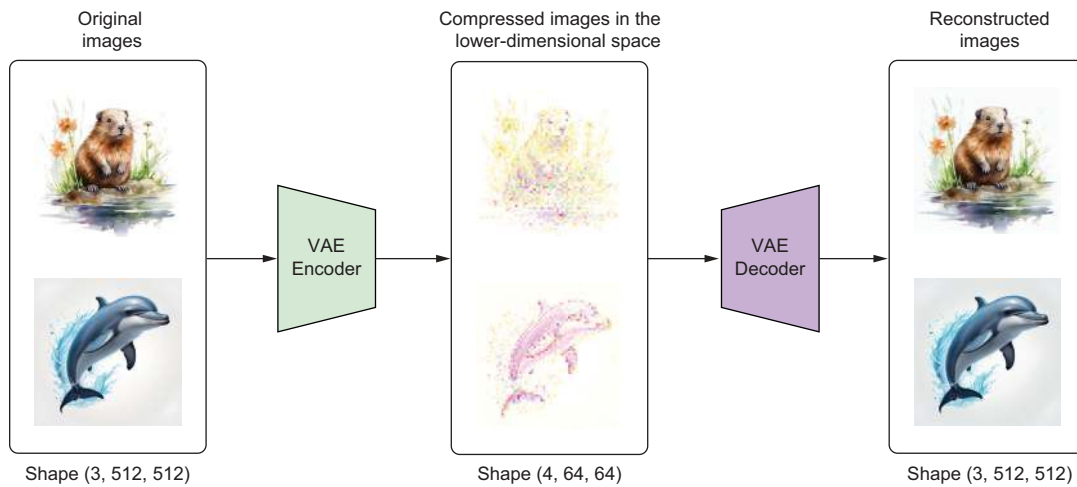
The core idea of an LDM is to conduct diffusion processes in lower-dimensional pixel spaces, which reduces the computational costs for both training and inference. Naturally, this approach results in lower image quality. To overcome this, we need a method to upscale these low-resolution images to high-resolution outputs, and that's where VAEs come in. In this section, we'll first explain how VAEs work and how they can enhance low-resolution images. Then, we'll discuss how combining an LDM with a VAE allows for the generation of coherent images based on text prompts.

### 9.1.1 How variational autoencoders work

VAEs are a class of generative models that blend deep learning with Bayesian inference to generate new data samples resembling those in the training set. A VAE is composed of two core components: an encoder and a decoder, both implemented as deep neural networks. The encoder compresses the input data into a lower-dimensional latent space by producing the parameters of a probability distribution (typically a normal distribution). Instead of mapping data to a single point, the encoder outputs a distribution over the latent space, encouraging smooth transitions and interpolation between data points. The decoder then samples from this distribution and reconstructs the original data from the latent variables. We'll provide a more formal introduction to VAEs in chapter 11 when we discuss vector quantized variational autoencoders (VQ-VAEs).

By working in the latent space (a compressed, abstract representation of images where similar visual patterns are mapped close together), diffusion models can operate on lower-dimensional data that is both computationally efficient and easier to handle than raw pixel values. The VAE constructs this latent space by learning to encode images into compact vectors that retain the most essential visual and semantic features (e.g., shapes, textures, and object relationships) while discarding less critical pixel-level details (e.g., minor noise or fine-grained textures). This tradeoff is guided by the VAE's loss function, which balances accurate reconstruction with smoothness and continuity in the latent space. When combined with LDMs, the VAE's ability to distill key image features allows the diffusion model to efficiently generate high-resolution, realistic images that preserve meaningful structure and content. Figure 9.2 illustrates how VAEs work.

The encoder of a VAE takes an input image, often high resolution, and maps it into an abstract representation in the lower-dimensional (latent) space. This process involves learning parameters that define the mean and variance of a normal distribution for



**Figure 9.2** How a VAE compresses images to representations in the lower-dimensional (latent) space and then reconstructs the images. A VAE consists of an encoder and a decoder. The encoder compresses images (e.g., with a size of  $3 \times 512 \times 512$ ) into a lower-dimensional space (i.e., the latent space, where images have a size of, e.g.,  $4 \times 64 \times 64$ ). The decoder takes the encoded images in the latent space and reconstructs the images to the original high-resolution space (with a size of  $3 \times 512 \times 512$  in this example).

each input image. By doing so, the encoder transforms the image into a condensed, informative representation while introducing controlled randomness that helps the model generalize.

During training, VAEs adjust model parameters to balance two objectives. The first is to minimize the reconstruction loss, which ensures that the decoder can accurately reproduce the input image from its latent representation. The second objective is to push the latent space distribution to align with the predefined one (usually a standard normal distribution). This dual objective helps the model learn a smooth, continuous latent space that is essential for generating coherent outputs. Given the large computational resources needed to train a VAE to generate high-resolution images, we'll use a pretrained VAE from Hugging Face in this chapter.

**NOTE** You can learn how to build and train a VAE from scratch in my book *Learn Generative AI with PyTorch* (Manning, 2025).

The latent space learned by the VAE captures the core attributes and features of the original images, even when starting with low-resolution inputs. This distilled representation captures the fundamental structure and visual details necessary for accurate reconstruction. As a result, even images with lower initial quality contain the building blocks for high-resolution reconstructions once processed through the latent space.

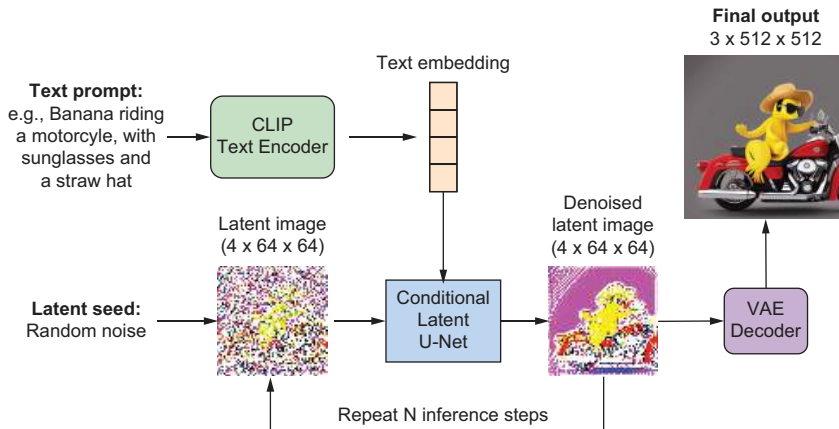
The decoder is responsible for converting these latent representations back into high-quality images. By sampling from the learned latent space, the decoder reconstructs images with added details and clarity. Essentially, the decoder learns how to

upscale and refine the information from the compact latent representations, effectively transforming low-resolution inputs into high-resolution outputs.

### 9.1.2 Combining a latent diffusion model with a variational autoencoder

LDMs have three main components: a VAE, a CLIP model, and a denoising U-Net model. The training data consists of image–caption pairs. LDMs are typically trained in two stages: (1) latent space encoding and (2) diffusion processes in the latent space. In the first stage, a VAE is trained on images to learn a compact, lower-dimensional latent representation. The goal here is to accurately reconstruct the original image from its latent representation, which reduces computational complexity for the second stage.

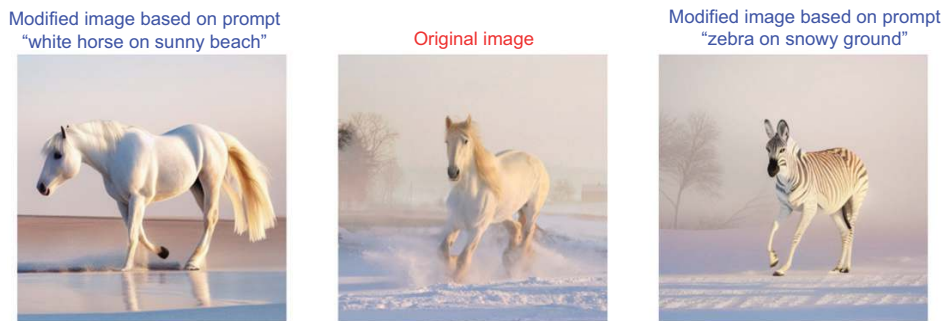
Once the latent space is established, a diffusion model with a U-Net architecture is trained to model the process of gradually adding and then removing noise from these latent representations. The text encoder and image encoder in the CLIP model convert captions and low-resolution images (i.e., latent images) into text embeddings and image embeddings, respectively. During training, a forward process progressively adds noise to latent images, and the model learns a reverse denoising process in the latent space. The training objective is to predict the noise added at each time step conditional on the text prompt. The CLIP model is used in the process to ensure that the generated latent images match the text descriptions by maximizing the similarities between text and image embeddings. Due to the excessive computational costs required to train LDMs, we'll use a pretrained model from Hugging Face. This way, we can focus on generating high-resolution images based on text prompts after training, and figure 9.3 shows how this process works.



**Figure 9.3** How a trained LDM generates an image based on a text prompt. The text prompt shown at the top-left corner is first encoded into a text embedding. To generate an image in the lower-dimensional space (latent space), we start with an image of pure noise (bottom left). We use the trained U-Net to iteratively denoise the image, conditional on the text embedding so the generated image matches the text prompt, with the guidance of a trained CLIP model. The generated latent image (bottom right) is presented to a trained VAE to convert it to a high-resolution image, which is the final output (top right).

Let's say we want to generate an image with the pretrained LDM that matches the following text prompt: "a banana riding a motorcycle, wearing sunglasses and a straw hat, highly detailed, ultra sharp, cinematic, 100 mm lens, 8k resolution" (a simplified version is shown in the top-left corner of figure 9.3). We first use the trained text encoder in the CLIP model to convert the prompt into a text embedding (top middle of the figure). We then start the image generation process in the latent space, as illustrated in the bottom half of the figure. We start with an image with a size of  $(4, 64, 64)$ , which is composed of pure random noise. We then iteratively remove noise from the image. This step is guided by the CLIP model by maximizing the similarity between the text embedding and the embedding of the generated image. Once a fixed number of reverse diffusion steps is finished in the latent space, the latent image is presented to the trained VAE to convert it into a high-resolution one, which is the final output of the model (top right of the figure). As you can see from the figure, the generated image matches the text prompt well: we see a banana-shaped rider with sunglasses and a straw hat.

We can also use the pretrained LDM to modify an image based on a text prompt. The steps are similar to those outlined in figure 9.3, with the exception that the denoising process starts with an encoded version of the existing image, not pure random noise. Imagine you have an image of a white horse walking on white, snow-covered ground, as shown in the middle of figure 9.4. We can modify the image to show a white horse walking on a sunny, sandy beach.



**Figure 9.4** Modifying an existing image with a text prompt using the pretrained LDM. The original image is shown in the middle of the figure. Using "a white horse walks on the sunny sandy beach" as the prompt, the resulting image is shown on the left. When we use "a zebra walks on the white, snow-covered ground" as the text prompt, the resulting image is shown on the right.

We'll use the encoder in the pretrained VAE to convert the original horse image into the lower-dimensional latent space, with a shape of  $(4, 64, 64)$ . We'll use this (instead of pure random noise) as the starting point of the reverse diffusion process in the latent space using the text prompt "a white horse walks on the sunny sandy beach" as guidance. The output is shown in the left image in figure 9.4.



If we use the prompt “a zebra walks on the white, snow-covered ground” to modify the image instead, the output is shown on the right of figure 9.4. It looks similar to the original image, with a similar snowy background, but the horse now becomes a zebra.

## 9.2 Compressing and reconstructing images with VAEs

To understand how VAEs compress and then reconstruct images, you’ll first download a pretrained VAE that we’ll use in our LDM. You’ll then use the encoder in the VAE to compress two example high-resolution images into the latent space. After that, you’ll use the decoder in the VAE to reconstruct the images. You’ll compare the reconstructed images with the original ones and see if any information is lost in the process.

The Python programs in this chapter are based on two excellent GitHub repositories: one by the Computer Vision and Learning research group at Ludwig Maximilian University of Munich (<https://github.com/CompVis/latent-diffusion>) and one by Umar Jamil (<https://mng.bz/PwDR>).

**NOTE** The Python programs for all chapters in this book can be accessed from the GitHub repository (<https://github.com/markhliu/txt2img>). Ideally, you’ll use a CUDA-enabled GPU to run the programs in this chapter. If you don’t have one, you can follow along using the Google Colab notebook (<https://mng.bz/Jw9Z>).

### 9.2.1 Downloading the pretrained VAE

Follow this code cell to download the three files, vocab.json, merges.txt, and hollie-mengert.ckpt:

```
import requests, os

urls=(("https://huggingface.co/ogkalu/Illustration-Diffusion"
      "/resolve/main/hollie-mengert.ckpt?download=true",
      "https://huggingface.co/stable-diffusion-v1-5/"
      "stable-diffusion-v1-5/resolve/main/tokenizer/merges.txt",
      "https://huggingface.co/stable-diffusion-v1-5/"
      "stable-diffusion-v1-5/resolve/main/tokenizer/vocab.json")

files=["files/hollie-mengert.ckpt",
       "files/merges.txt",
       "files/vocab.json"]

if all(os.path.exists(file) for file in files):
    print("files have already been downloaded")
else:
    print("downloading files")
    for url, file in zip(urls, files):
        fb=requests.get(url)
        with open(file,"wb") as f:
            f.write(fb.content)
    print("download complete")
```

**URLs to download the three files**

**Where to save the three files on your computer**

**Checks if the three files already exist on your computer**

**If not, downloads the three files**

The links to download the three files are provided in the tuple `urls`. Check to see if the three files already exist on your computer. If not, use the `requests` library to download the files and save them on your computer.

Run the following code cell to clone Umar Jamil's GitHub repository:

```
!git clone https://github.com/hkproj/pytorch-stable-diffusion
```

You'll find a folder `pytorch-stable-diffusion` containing all files in Umar Jamil's GitHub repository on your computer. Next, we load the pretrained VAE.

### Listing 9.1 Loading the pretrained VAE

```
import sys
sys.path.append("pytorch-stable-diffusion/sd")

from model_converter import load_from_standard_weights
from encoder import VAE_Encoder
from decoder import VAE_Decoder
import torch

model_file = "files/hollie-mengert.ckpt"
state_dict = load_from_standard_weights(model_file,
                                       "cpu")
encoder = VAE_Encoder()
encoder.load_state_dict(state_dict['encoder'],
                       strict=True)
decoder = VAE_Decoder()
decoder.load_state_dict(state_dict['decoder'],
                       strict=True)
```

← Accesses the subdirectory in the cloned GitHub repository

← Loads the pretrained weights in the VAE

← Creates the VAE encoder

← Creates the VAE decoder

The `sys.path.append()` method allows us to access a subdirectory on your computer. We use the method here to access the module files in the cloned GitHub repository. We then instantiate a VAE and load the pretrained weights to the encoder and decoder.

Download the two image files, `beaver.jpg` and `dolphin.png`, from the book's GitHub repository (<https://github.com/markhliu/txt2img>), and place them in the `files` folder on your computer (or simply clone the repository to your computer). If you're working in Google Colab, you can quickly set up the repository and add it to your working directory by running

```
!git clone https://github.com/markhliu/txt2img
import sys
sys.path.append("/content/txt2img")
```

We'll load the two images and resize them so that we can feed them to the VAE encoder later.

### Listing 9.2 Loading and resizing two example images

```
from PIL import Image
from torchvision import transforms
```

```
def center_crop(img):
    width, height = img.size
    new=min(width,height)
    left = (width - new)/2
    top = (height - new)/2
    right = (width + new)/2
    bottom = (height + new)/2
    im = img.crop((left, top, right, bottom))
    return im
image1=center_crop(Image.open(
    "files/beaver.jpg")).resize((512,512))
image2=center_crop(Image.open(
    "files/dolphin.png")).resize((512,512))
transform=transforms.ToTensor()
image1=transform(image1)
image2=transform(image2)
print(image1.shape, image2.shape)
```

← Defines a function to center crop an image

← Loads the beaver image and resizes it to  $3 \times 512 \times 512$

← Loads the dolphin image and resizes it to  $3 \times 512 \times 512$

← Prints out the shapes of the resized images

We define a `center_crop()` function to crop the center square of an image. Doing so will ensure that the image has the same width and height; otherwise, the resized image may appear distorted and disproportional. We then load the beaver and dolphin images, center crop them, and resize them to a shape of  $3 \times 512 \times 512$ . After running the code in listing 9.2, you'll see the following output:

```
torch.Size([3, 512, 512]) torch.Size([3, 512, 512])
```

The output shows that both images are PyTorch tensors with a size of  $3 \times 512 \times 512$ . Next, we'll use the VAE encoder to compress them into representations in the lower-dimensional latent space.

### Exercise 9.1

Download the two image files, `bird.png` and `cat.png`, from the book's GitHub repository (<https://github.com/markhliu/txt2img>), and place them in the `files` folder on your computer. Apply the `center_crop()` function on them, and then resize them to a size of  $3 \times 512 \times 512$ . Name them `image3` and `image4`, respectively. Print out the shape of the two processed images.

## 9.2.2 Encoding and decoding images with the pretrained VAE

The VAE encoder will compress images of size  $3 \times 512 \times 512$  to a lower dimension of  $4 \times 64 \times 64$ . Doing so greatly reduces the pixel count of the images. We can compress the two example images we just prepared into the latent space with the pretrained VAE encoder:

```
N=torch.distributions.Normal(0, 1)
noise=N.sample(torch.Size([1,4,64,64]))
latent1=encoder(image1.unsqueeze(0),
```

← Sample from the standard normal distribution

```

noise)[0]
latent2=encoder(image2.unsqueeze(0),
noise)[0]
print(latent1.shape,latent2.shape)

```

Compresses the beaver and dolphin images to the latent space

Prints out the shape of the compressed images

We first sample a noise tensor from the standard normal distribution. We then use the pretrained VAE encoder to sample latent representations for the two images in the lower-dimensional space. We call them `latent1` and `latent2`, respectively. The output from the preceding code is

```
torch.Size([4, 64, 64]) torch.Size([4, 64, 64])
```

This indicates that the two images are now compressed to a size of  $4 \times 64 \times 64$ .

### Exercise 9.2

Use the VAE encoder to compress the two images, `image3` and `image4`, to the latent space so that they have a shape of  $4 \times 64 \times 64$ . Name the two latent images `latent3` and `latent4`.

Now let's visually compare the original images with the compressed lower-dimensional images.

### Listing 9.3 Visually comparing original and compressed images

```

import matplotlib.pyplot as plt
import numpy as np

imgs=[image1.permute(1,2,0),
      torch.clip(latent1.permute(1,2,0),-1,1).detach().cpu().numpy(
          ).repeat(8, axis=0).repeat(8,axis=1)/2+0.5,
      image2.permute(1,2,0),\
      torch.clip(latent2.permute(1,2,0),-1,1).detach().cpu().numpy(
          ).repeat(8, axis=0).repeat(8,axis=1)/2+0.5]
plt.figure(figsize=(12,3),dpi=100)
for i in range(4):
    plt.subplot(1,4, i+1)
    plt.imshow(imgs[i])
    if i%2==0:
        plt.title("original image")
    else:
        plt.title("latent image")
    plt.axis('off')
plt.tight_layout()
plt.show()

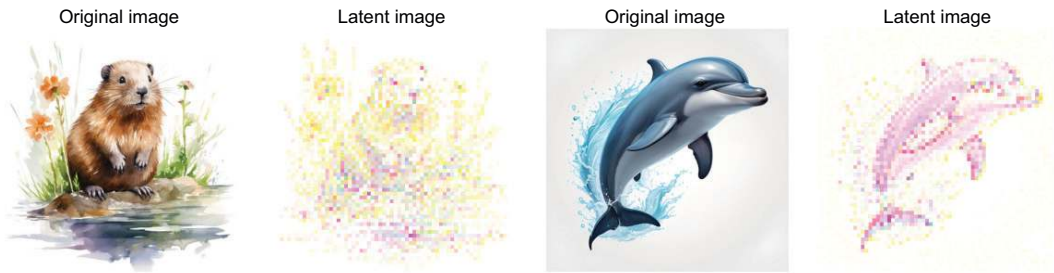
```

Places the original images and compressed images in one list

Plots the four images in a  $1 \times 4$  grid

Labels each image as either original or latent

We place the two original images and the two compressed images in the same list `imgs`. We then plot the four images in a  $1 \times 4$  grid. We label each image as either “original image” or “latent image.” The result is shown in figure 9.5.



**Figure 9.5** Comparing the original images with latent images. The first and third images are original images of a beaver and a dolphin, respectively, with a shape of  $3 \times 512 \times 512$ . We use a pretrained VAE encoder to compress the images to a lower-dimensional space. The compressed latent images have a shape of  $4 \times 64 \times 64$ , with only about 2% of the number of pixels of the original images. However, we can still tell that the second image is a beaver and the last image is a dolphin.

Figure 9.5 compares the two original images with the two latent ones. The original images have 48 times more pixels than the latent images have. However, we can still tell that the second image is a beaver and the last image is a dolphin, indicating that the VAE encoder has captured the essential features of the two images.

### Exercise 9.3

Plot the two original images, `image3` and `image4`, and the two latent images, `latent3` and `latent4`, in a  $1 \times 4$  grid to compare them.

Next, we feed the two latent images, `latent1` and `latent2`, to the pretrained VAE decoder to convert them back to the original high-dimensional space:

```
reconstruct1=decoder(latent1.unsqueeze(0))
reconstruct2=decoder(latent2.unsqueeze(0))
print(reconstruct1.shape,reconstruct2.shape)
```

The two decoded images are saved as `reconstruct1` and `reconstruct2`, respectively. We printed out the shapes of the two reconstructed images, and the output is

```
torch.Size([1, 3, 512, 512]) torch.Size([1, 3, 512, 512])
```

The output shows that the two latent images are converted to the original high-dimensional space, with shapes of  $3 \times 512 \times 512$ . We plot both the original images and the reconstructed images to compare them.

### Listing 9.4 Comparing the original and reconstructed images

```
imgs=[image1.permute(1,2,0),
      torch.clip(reconstruct1[0].permute(1,2,0),
```

```

    0,1).detach().cpu().numpy(),
    image2.permute(1,2,0),
    torch.clip(reconstruct2[0].permute(1,2,0),
               0,1).detach().cpu().numpy()])
plt.figure(figsize=(12,3),dpi=100)
for i in range(4):
    plt.subplot(1,4, i+1)
    plt.imshow(imgs[i])
    if i%2==0:
        plt.title("original image")
    else:
        plt.title("reconstructed image")
    plt.axis('off')
plt.tight_layout()
plt.show()

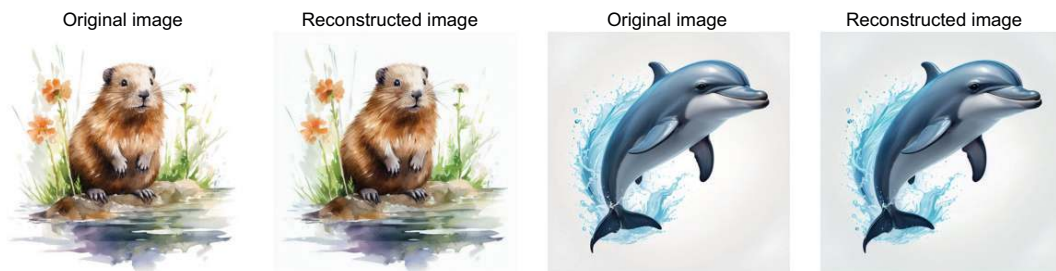
```

Places original and reconstructed images in a list

Plots the four images in a 1 × 4 grid

Labels each image as either original or reconstructed

We first place the two original and the two reconstructed images in a list. We then plot them in a 1 × 4 grid and label each image as either original or reconstructed. The output is shown in figure 9.6.



**Figure 9.6** Comparing the original images with reconstructed images. The first and third images are original images of a beaver and a dolphin, respectively, with a shape of  $3 \times 512 \times 512$ . We use a pretrained VAE encoder to compress the images into lower-dimensional latent images with a shape of  $4 \times 64 \times 64$ . We then use the pretrained VAE decoder to convert the latent images to the original high-dimensional space, as shown in the second and fourth images.

The first and third images in figure 9.6 are the original images, while the second and fourth images are the reconstructed ones. The reconstructed images closely resemble the original ones, which indicates that the VAE decoder can successfully reconstruct an image.

#### Exercise 9.4

Use the pretrained VAE decoder to convert the two latent images, `latent3` and `latent4`, into two high-dimensional images. Name them `reconstruct3` and `reconstruct4`, respectively. Then, plot the two original images, `image3` and `image4`, and the two reconstructed images in a 1 × 4 grid to compare them.

### 9.3 Text-to-image generation with latent diffusion

Now that you understand how the pretrained VAE decoder converts low-resolution images in the latent space into high-resolution ones, we'll discuss how to conduct diffusion in the low-dimensional latent space. The CLIP model will guide this process.

#### 9.3.1 Guidance by the CLIP model

We'll use a pretrained CLIP model to guide the diffusion model to generate latent images that match the descriptions in the text prompt. We'll first use the pretrained CLIP tokenizer from the transformers library to tokenize the text prompt (make sure you have the two files `vocab.json` and `merges.txt` saved on your computer):

```
from transformers import CLIPTokenizer
from clip import CLIP

device="cuda" if torch.cuda.is_available() else "cpu"
decoder.to(device)
tokenizer = CLIPTokenizer("files/vocab.json",
                        merges_file="files/merges.txt")
clip = CLIP().to(device)
clip.load_state_dict(state_dict['clip'],
                    strict=True)
```

Instantiates the `CLIPTokenizer()` class

Creates a CLIP model

Loads pretrained weights to the CLIP model

The `CLIPTokenizer()` class is used to tokenize text, and the vocabulary (i.e., collection of tokens) comes from the two files you just downloaded. We also import the CLIP model from Umar Jamil's GitHub repository directly and load it with the pretrained weights. From now on, we'll move different parts of the model (e.g., the VAE decoder, the CLIP tokenizer, and later the diffusion model) and all tensors to the GPU if your computer has a CUDA-enabled GPU. We do this to speed up the image-generation process.

We'll use "a banana riding a motorcycle, wearing sunglasses and a straw hat, highly detailed, ultra sharp, cinematic, 100 mm lens, 8k resolution" as our prompt. Because the final output is a combination of conditional and unconditional generation, as we explained in chapter 6, we need an unconditional prompt as well. An unconditional prompt is essentially an empty string passed to the text encoder. It represents the absence of any specific condition or descriptive information.

We'll use the classifier-free guidance (CFG) parameter, `cfg_scale`, to steer the generation process toward outputs that better match the provided conditional prompt, while reducing artifacts. We do this by combining the predictions obtained from both the conditional prompt and the unconditional prompt, as shown in listing 9.5. This mechanism pushes the generation closer to what the conditional prompt describes while using the unconditional output as a baseline to maintain diversity and stability in the output.

## Listing 9.5 Tokenizing and encoding prompts

```

prompt = '''A banana riding a motorcycle, wearing sunglasses and
a straw hat, highly detailed, ultra sharp, cinematic,
100mm lens, 8k resolution.'''

uncond_prompt=""

seed=42
with torch.no_grad():
    generator=torch.Generator(device=device)
    generator.manual_seed(seed)
    cond_tokens=tokenizer.batch_encode_plus(
        [prompt], padding="max_length", max_length=77).input_ids
    cond_tokens=torch.tensor(cond_tokens,
        dtype=torch.long,device=device)
    print("conditional tokens are\n", cond_tokens)
    cond_context = clip(cond_tokens).to(device)
    uncond_tokens=tokenizer.batch_encode_plus(
        [uncond_prompt], padding="max_length",
        max_length=77).input_ids
    uncond_tokens=torch.tensor(uncond_tokens,
        dtype=torch.long,device=device)
    print("unconditional tokens are\n", uncond_tokens)
    uncond_context = clip(uncond_tokens).to(device)
    clip.to("cpu")
    context=torch.cat([cond_context,uncond_context])

```

**Tokenizes the conditional prompt into a 77-token tensor**  
**Uses the CLIP text encoder to encode the conditional prompt**  
**Does the same for the unconditional prompt**  
**Concatenates encoded conditional and unconditional prompts**

The second half of the prompt, “highly detailed, ultra sharp, cinematic, 100 mm lens, 8k resolution,” provides the image generator with more specific instructions about the style, quality, and technical aspects of the image. This helps the model focus on producing a more refined and polished image. We fix the random state at 42 so that results are reproducible. We convert both conditional and unconditional prompts into 77-element tensors. The CLIP text encoder is designed to accept inputs of a fixed length (77 tokens). This design decision was made to provide enough context for a wide range of inputs while keeping the computational requirements manageable. If a prompt is shorter than 77 tokens, it gets padded; if longer, it’s truncated. This standardization ensures that every input, whether conditional or unconditional, fits the same 77-token structure.

We then use the CLIP text encoder to convert the tokenized conditional prompt into a text embedding. We do the same for the tokenized unconditional prompt. Finally, we combine the two embeddings in the same tensor named context.

The output from listing 9.5 is

```

conditional tokens are
tensor([[49406, 320, 8922, 6765, 320, 10297, 267, 3309,
12906, 537, 320, 16438, 3801, 267, 5302, 12609, 267, 8118,
8157, 267, 25602, 267, 272, 271, 271, 2848, 8666,
267, 279, 330, 9977, 269, 49407, 49407, 49407, 49407, 49407,

```



As you can see from the preceding output, we use the integer 49406 to signal the start of the sequence, and we use the integer 49407 to pad the sequence so it has 77 elements.

To conduct diffusion in the latent space, we need to first initiate a diffusion model and create a noise scheduler. The following code listing accomplishes those tasks.

**Creates a noise scheduler and sets the number of inference time steps to 50**

## Listing 9.7 Conducting reverse diffusion in the latent space

```

from tqdm import tqdm
from copy import deepcopy

cfg_scale=8
latent_shape=(1,4,64,64)
noisy_latents=[]
with torch.no_grad():
    latents=torch.randn(latent_shape,
        generator=generator,device=device)
    timesteps=tqdm(sampler.timesteps)
    for i, timestep in enumerate(timesteps):
        time_embedding=get_time_embedding(timestep).to(device)
        model_input=latents.to(device)
        model_input=model_input.repeat(2,1,1,1)
        model_output=diffusion(model_input,context,time_embedding)
        output_cond,output_uncond=model_output.chunk(2)
        model_output=cfg_scale*(output_cond-output_uncond)+\
            output_uncond
        latents=sampler.step(timestep,latents,model_output)
        if i%10==9:
            noisy_latents.append(deepcopy(latents))
diffusion.to("cpu")

```

Start with an image consisting of pure noise.

Iteratively denoise for 50 inference steps.

The model output is a combination of conditional and unconditional generation.

Save the intermediate images to the list noisy\_latents.

We set the CFG guidance parameter to 8 for the moment. We'll explore how different values of this parameter affect the output later. The shape of the latent image is (1, 4, 64, 64), meaning 1 image in the batch, and the image has a shape of  $4 \times 64 \times 64$ . We start with an image consisting of pure noise (corresponding to  $t = 1,000$ ) and iteratively remove noise from the image, guided by the text prompt. Note that the `diffusion()` function in the listing 9.7 takes three arguments, which guide the diffusion model to predict the noise in the image so we can remove noise effectively:

- `model_input`—The noisy image
- `context`—Encodes both the conditional and unconditional text prompts
- `time_embedding`—Embeds the time step so the model knows which time step the noisy image is in

Because we use 50 inference time steps, we generate 50 images corresponding to time steps  $t = 980, 960, \dots, 20$ , and 0. We saved 5 noisy images at time steps  $t = 800, 600, 400, 200$ , and 0, respectively, for later use.

### 9.3.3 Converting latent images to high-resolution ones

The images generated in the latent space have a shape of  $4 \times 64 \times 64$ . We'll use the pre-trained VAE to convert them into high-resolution images with a shape of  $3 \times 512 \times 512$  in the high-dimensional space. We first convert the final, clean image at time step 0 in the latent space into a high-resolution image:

```

final_latent=noisy_latents[-1]
final_output=decoder(final_latent)
img=torch.clip(final_output[0].permute(1,2,0)/2+0.5,0,1)
plt.figure(dpi=100)
plt.imshow(img.detach().cpu().numpy())
plt.axis("off")
plt.tight_layout()
plt.show()

```

Retrieves the final latent image

Uses the pretrained VAE decoder to convert the latent image to a high-resolution image

Changes the image to the channel-last format

Displays the final output

We retrieve the final latent image from the list `noisy_latents` that we generated in the previous subsection. We then use the pretrained VAE decoder to convert the latent image into a high-resolution image. The output is shown in figure 9.7.



**Figure 9.7** An image generated by a pretrained LDM with a text prompt “a banana riding a motorcycle, wearing sunglasses and a straw hat, highly detailed, ultra sharp, cinematic, 100 mm lens, 8k resolution”

As you can see, the image shows a human-shaped banana riding a motorcycle. The rider wears a straw hat and sunglasses, as described in the text prompt. This shows the ability of the trained LDM to generate images that closely match the text prompt. To get a better understanding of how the latent diffusion processes work, we can plot the five latent images in the list `noisy_latents`:

```

plt.figure(figsize=(10,3),dpi=100)
for i in range(5):

```

Plots the five images in a 1 × 5 grid

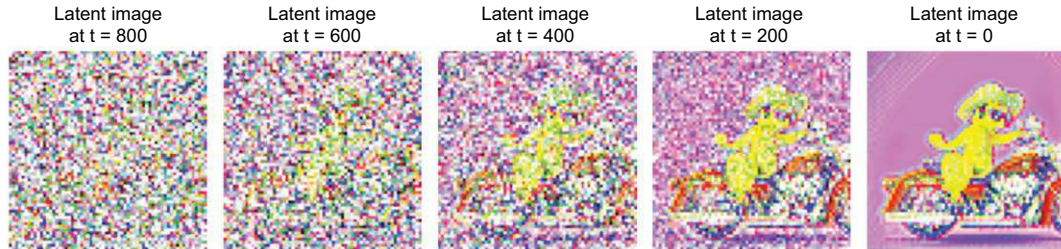
```

im=torch.clip(noisy_latents[i][0].permute(1,2,0)/2+0.5,0,1)
plt.subplot(1,5, i+1)
plt.imshow(im.detach().cpu().numpy())
plt.title(f"latent image\nat t={800-i*200}")
plt.axis('off')
plt.tight_layout()
plt.show()

```

Shows which time step the latent image is in

We plot the five images in a  $1 \times 5$  grid. The images are shown in figure 9.8.



**Figure 9.8** Generated latent images in time steps 800, 600, . . . , and 0. These images are generated with a pretrained LDM. We start with an image with pure random noise at  $t = 1,000$ . We use the diffusion model to denoise it iteratively and obtain the noisy image at  $t = 800$ , which is shown in the far left of the figure. We continue to denoise the image and obtain images at  $t = 600, 400$ , and  $200$ . Finally, we obtain the image at  $t = 0$ , which is shown at the far right of the figure. The images have a shape of  $4 \times 64 \times 64$ .

The far-left image in figure 9.8 is at time step  $t = 800$ . The image is indistinguishable from random noise. The middle three images are at time steps  $t = 600, 400$ , and  $200$ , respectively. They start to show a banana-shaped rider on a motorcycle. The far-right image is the clean image at  $t = 0$ . Even though the resolution is low, the image fits the text prompt well.

All the images in figure 9.8 are low-resolution images with a shape of  $4 \times 64 \times 64$ . We can use the pretrained VAE decoder to convert them into high-resolution images, which are shown as output in figure 9.9:

```

plt.figure(figsize=(10,3),dpi=100)
for i in range(5):
    im=decoder(noisy_latents[i])
    im=torch.clip(im[0].permute(1,2,0)/2+0.5,0,1)
    plt.subplot(1,5, i+1)
    plt.imshow(im.detach().cpu().numpy())
    plt.title(f"decoded image\nat t={800-i*200}")
    plt.axis('off')
plt.tight_layout()
plt.show()

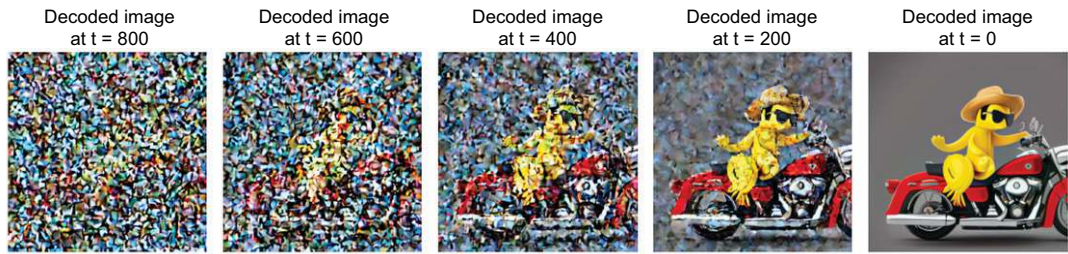
```

Uses the VAE decoder to convert latent images to high resolution ones

Plots the decoded image

Indicates which time step the image is in

The five images in the latent space are now converted into high-resolution images, with a shape of  $3 \times 512 \times 512$ . As you go from left to right in figure 9.9, the decoded



**Figure 9.9** Converting the intermediate latent images into the high-dimensional space. The five latent images in figure 9.8, each with a shape of  $4 \times 64 \times 64$ , are converted by the pretrained VAE decoder into the high-dimensional space, with a shape of  $3 \times 512 \times 512$ .

image becomes clearer and clearer. The far-right image is the final output from the LDM model that you saw earlier in figure 9.7.

## 9.4 Modifying existing images with text prompts

The LDM can also modify an existing image based on a text prompt. The process of modifying an existing image using an LDM works very much like generating an image from scratch, but with a key difference: you start from an encoded version of the image rather than pure noise.

The process begins by taking your existing image and encoding it into a latent representation using the pretrained VAE encoder. The VAE compresses the image into a lower-dimensional latent space that captures its essential features. To allow the model to modify the image based on a text prompt, a controlled amount of noise is introduced to the latent representation. The level of noise is governed by a “strength” parameter, which determines how much the final image will deviate from the original. Less noise preserves more of the original image, while more noise allows for greater modifications.

Download the image `horse.png` from the book’s GitHub repository. It’s an image of a white horse walking on white, snow-covered ground, as shown in figure 9.10.



**Figure 9.10** An image of a horse. We’ll use a pretrained LDM to modify this image based on text prompts.

In the following listing, we modify the image to show a white horse walking on a sunny, sandy beach.

#### Listing 9.8 Modifying an existing image with the LDM

```
import pipeline
import model_loader

models=model_loader.preload_models_from_standard_weights(model_file,
    device)
prompt="A white horse walks on the sunny sandy beach."
uncond_prompt=""
input_image = Image.open("files/horse.png")

output_image = pipeline.generate(
    prompt=prompt,
    uncond_prompt=uncond_prompt,
    input_image=input_image,
    strength=0.8,
    do_cfg=True,
    cfg_scale=8,
    sampler_name="ddpm",
    n_inference_steps=50,
    seed=42,
    models=models,
    device=device,
    idle_device="cpu",
    tokenizer=tokenizer,)
```

**The prompt used to modify the image**

**The existing image**

**Uses the generate() function in the pipeline module in Umar Jamil's GitHub repository**

**Sets the strength parameter to 0.8**

The generate() function in the pipeline module in Umar Jamil's GitHub repository can be used to modify an existing image or generate an image from scratch. The strength argument in the function ranges from 0 to 1, and it controls how much to modify the image: a higher value means the output deviates more from the original image. The generated image is shown in figure 9.11.

Next, we use the prompt "a zebra walks on the white, snow-covered ground" to modify the existing horse image:

```
prompt="A zebra walks on the white, snow-covered ground."
output_image = pipeline.generate(
    prompt=prompt,
    uncond_prompt=uncond_prompt,
    input_image=input_image,
    strength=0.5,
    do_cfg=True,
    cfg_scale=8,
    sampler_name="ddpm",
    n_inference_steps=50,
    seed=42,
    models=models,
    device=device,
    idle_device="cpu",
    tokenizer=tokenizer,)
```

**The new text prompt used to modify the image**

**Sets the strength parameter to 0.5**





**Figure 9.11** A modified horse image with the LDM. We use the text prompt “a white horse walks on the sandy sunny beach” to modify the horse image shown in figure 9.10 using a pretrained LDM.

We again use the `generate()` function in the `pipeline` module to modify the horse image. This time, we set the `strength` argument in the function to `0.5`. The generated image looks very similar to the original image, except that the horse now becomes a zebra (see figure 9.12).



**Figure 9.12** A modified image with a pretrained LDM based on the text prompt “a zebra walks on the white, snow-covered ground.” The original image is a white horse, as shown earlier in figure 9.10.

**Exercise 9.5**

Use “a dog walks on the white, snow-covered ground” as the prompt to modify the existing horse image. Set the strength parameter to 0.5, the `cfg_scale` parameter to 8, and the random state number (i.e., the seed argument in the `generate()` function) to 42. Plot the generated image.

We’ve set the `cfg_scale` parameter to a value of 8 thus far when generating images. If we set the parameter to a lower value, say, 5, the output is more diverse but may not align perfectly with the text prompt. On the other hand, if you set the parameter to a higher value, say, 12, the output is more aligned with the text prompt. However, setting the parameter too high may lead to overfitting the prompt or a reduction in the natural diversity of the output.

Next, we use the same text prompt, “a banana riding a motorcycle, wearing sunglasses and a straw hat, highly detailed, ultra sharp, cinematic, 100 mm lens, 8k resolution,” that we used before, but set the `cfg_scale` parameter to 12.

**Listing 9.9 Text-to-image generation with a high `cfg_scale`**

```
prompt = '''A banana riding a motorcycle, wearing sunglasses and
a straw hat, highly detailed, ultra sharp, cinematic,
100mm lens, 8k resolution.'''
uncond_prompt=""
output_image12 = pipeline.generate(
    prompt=prompt,
    uncond_prompt=uncond_prompt,
    input_image=None,
    strength=1,
    do_cfg=True,
    cfg_scale=12,
    sampler_name="ddpm",
    n_inference_steps=50,
    seed=42,
    models=models,
    device=device,
    idle_device="cpu",
    tokenizer=tokenizer,)
plt.figure(figsize=(5,5),dpi=100)
plt.imshow(output_image12)
plt.axis('off')
plt.tight_layout()
plt.show()
```

← Uses the `generate()` function in the pipeline module

← Sets the `input_image` argument to `None`

← Sets the `cfg_scale` argument to 12

← Plots the generated image

When we set the `input_image` argument to `None`, the `generate()` function in the pipeline module generates an image from scratch instead of modifying an existing image. The `generate()` function implements the steps we went through in the last section: it tokenizes and encodes the text prompt, generates an image in the latent space, and converts the latent image into a high-resolution image as the output. We set



the `cfg_scale` argument to 12, higher than the value we set before (8). The output is shown in figure 9.13.



**Figure 9.13** An image generated by a trained LDM based on a text prompt when we set the `cfg_scale` parameter to 12. The text prompt is “a banana riding a motorcycle, wearing sunglasses and a straw hat, highly detailed, ultra sharp, cinematic, 100 mm lens, 8k resolution.” The CFG parameter, `cfg_scale`, steers the generation process toward outputs that better match the provided conditional prompt, while reducing artifacts. A low value of the `cfg_scale` parameter (e.g., 5) leads to a more diverse output, and a high value (e.g., 12) leads to an output that aligns more closely with the text prompt.

The image in figure 9.13 aligns with the text prompt more closely compared to the image in figure 9.7, when we set the `cfg_scale` parameter to 8. The rider resembles more of a human rider: you can clearly see the mouth and ear in the final output.

### Exercise 9.6

Change the `cfg_scale` parameter to 5 in listing 9.9, and check to see what the output looks like.

One of the state-of-the-art text-to-image models, Stable Diffusion, is based on the LDM. In the next chapter, you’ll explore various components of Stable Diffusion and use it to generate high-resolution images with text prompts.

## Summary

- Generating high-resolution images with diffusion models conditional on text prompts is computationally costly, making training and inference inefficient and slow. To overcome the challenge, Rombach et al. proposed a latent diffusion model (LDM) in 2022 [1]. Instead of conducting forward and reverse diffusion processes on high-resolution images, the authors propose conducting the diffusion processes in a lower-dimensional space (i.e., the latent space).
- Because images in the latent space are much smaller in size, the diffusion processes are much faster. However, this means lower quality of the generated images. To compensate for this, a variational autoencoder (VAE) is used to convert the low-resolution latent images back into high-resolution ones as the final output.
- A VAE is a generative model composed of an encoder and a decoder. The encoder transforms high-resolution images into compact, lower-dimensional representations, which the decoder uses to reconstruct the original image. The decoder's ability to convert low-resolution images into high-resolution output is an essential part of the LDM.
- An LDM can also modify an existing image based on a text prompt, similar to how it generates an image from scratch, but with a key difference: you start from an encoded version of the image rather than pure noise.
- The classifier-free guidance (CFG) parameter, `cfg_scale`, controls the diversity versus fidelity of generated images. A low `cfg_scale` value leads to more diverse output. A high `cfg_scale` value leads to an output that is more aligned with the text prompt.
- One of the state-of-the-art text-to-image generators, Stable Diffusion, which we'll cover in the next chapter, is based on LDMs.

# 10

## *A deep dive into Stable Diffusion*

---

### ***This chapter covers***

- Differences between Stable Diffusion and the latent diffusion model
- Accessing Stable Diffusion with the `diffusers` library
- Components of Stable Diffusion and their roles in text-to-image generation
- Interpolating text embeddings to generate a series of images

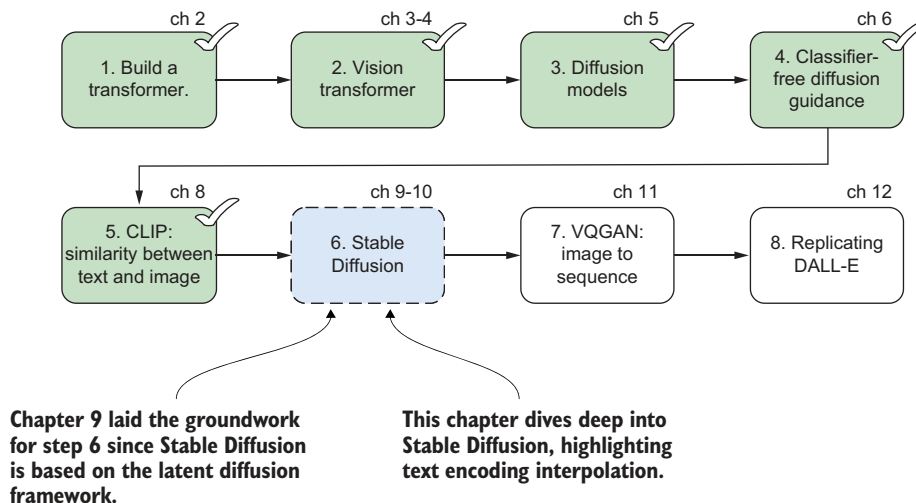
In the world of text-to-image generation, Stable Diffusion stands out as one of the most powerful and accessible models. Building on the latent diffusion architecture discussed in the previous chapter, Stable Diffusion pushes the boundaries of image quality, flexibility, and speed, all while remaining open source and free for anyone to use.

Developed collaboratively by CompVis, Stability AI, and LAION, Stable Diffusion draws on the breakthrough techniques introduced by Rombach et al. [1], combining

them with advanced optimizations and a vastly expanded training dataset. While the original latent diffusion model (LDM) was trained on the 400-million image LAION-400M dataset, Stable Diffusion uses a curated subset of the even larger LAION-5B, unlocking a richer diversity of visual concepts and styles.

What truly sets Stable Diffusion apart is its commitment to democratizing generative AI. The model's integration into user-friendly applications and open source libraries, such as Hugging Face's *diffusers*, means that anyone, regardless of technical background, can generate stunning images from simple text prompts. You no longer need a deep learning lab or specialized hardware to bring your ideas to life.

Figure 10.1 is a diagram of how this chapter fits into your journey of building a text-to-image generator. We zero in on step 6, Stable Diffusion, which is based on the LDM we discussed in chapter 9.



**Figure 10.1** Eight steps for building a text-to-image generator from scratch. Here, we focus on step 6, Stable Diffusion, showing how this state-of-the-art generator builds directly on the LDMs introduced in chapter 9. You'll not only learn how to use Stable Diffusion but also how to manipulate the text embeddings to create novel images.

This chapter dives deep into Stable Diffusion and highlights how Stable Diffusion differentiates from earlier LDMs. You'll use the Hugging Face *diffusers* library to quickly generate high-quality images. You'll also learn the model's core components, including the VAE, the denoising U-Net, and the contrastive language-image pretraining (CLIP) text encoder, and how each contributes to translating text prompts into coherent images. Interpolating between different text embeddings allows you to smoothly morph one generated image into another and gain insights into the underlying generative process.

## 10.1 Generating images with Stable Diffusion

In this section, you'll learn how to create high-resolution images by using a version of the Stable Diffusion that's integrated into the Hugging Face `diffusers` library. To install the library, run the following line of code in Jupyter Notebook:

```
!pip install diffusers
```

The `diffusers` library provides a standardized, user-friendly interface that abstracts away much of the underlying complexity of the diffusion process. The integration makes this state-of-the-art technology more approachable to the general public. The `StableDiffusionPipeline` package in the `diffusers` library allows you to use Stable Diffusion as an off-the-shelf tool to generate extraordinary images in just a few lines of code.

**NOTE** The Python programs for all chapters in this book can be accessed from the GitHub repository (<https://github.com/markhliu/txt2img>). Ideally, you'll want a CUDA-enabled GPU to run the programs in this chapter. If you don't have one, you can follow along using the Google Colab notebook (<https://mng.bz/wZX5>).

Let's use the prompt "an astronaut in a spacesuit riding a unicorn" to generate an image. The following listing shows how.

**Listing 10.1** Image generation with the `diffusers` library

```
import torch
from diffusers import StableDiffusionPipeline

torch.manual_seed(42)
prompt="an astronaut in a spacesuit riding a unicorn"
pipe = StableDiffusionPipeline.from_pretrained(
    "CompVis/stable-diffusion-v1-4",
    variant="fp16",
    torch_dtype=torch.float16).to("cuda")
image = pipe(prompt).images[0]
image
```

Uses the `StableDiffusionPipeline` package in the `diffusers` library

Selects the version of the Stable Diffusion model

Chooses the half-precision floating-point format

The `from_pretrained()` method in the `StableDiffusionPipeline` package loads a pre-trained version of the Stable Diffusion model. You can check out different versions of the model in Hugging Face at <https://huggingface.co/CompVis/stable-diffusion>. Here, we'll use the version `stable-diffusion-v1-4`. The first time you use it, Python downloads the pretrained weights to your computer. Because the weights are large (more than 1 GB), we'll use this version for all examples in this chapter. If you use a different version of the pretrained model, simply change the version name in the code cell.

The arguments `variant="fp16"` and `torch_dtype=torch.float16` mean that we load a model checkpoint using the half-precision floating-number format. The default data type in PyTorch tensors is `float32`, a 32-bit floating-point number. A half-precision

floating-point number, `float16`, provides less precise results than `float32` but also takes less time to compute, thereby speeding up the image-generation process. After running the code in listing 10.1, you'll see an image as shown in figure 10.2.



**Figure 10.2** An image generated by the `StableDiffusionPipeline` package of the `diffusers` library using the text prompt “an astronaut in a spacesuit riding a unicorn”

The generated image clearly shows an astronaut riding a unicorn on an alien planet. This indicates that Stable Diffusion can convert various concepts (e.g., “astronaut,” “spacesuit,” and “unicorn”) in natural language into visual concepts and generate a coherent image.

### Exercise 10.1

Change the text prompt to “a futuristic city at sunrise” in listing 10.1. Rerun the code listing to see what image you get.

You can generate multiple images at once. Here, we generate three images using three different prompts:

```

from PIL import Image

torch.manual_seed(0)
prompts = ["a panda eating a bowl of noodles",
           "a dog with sunglasses and a straw hat",
           "a cat stretching on the floor"]
images = pipe(prompts).images
grid = Image.new('RGB', size=(3*images[0].size[0],
                             images[0].size[1]))
for i, img in enumerate(images):
    grid.paste(img, box=(i*images[0].size[0], 0))
grid

```

Provides a list of three different text prompts

Places the three output images in a 1 × 3 grid

We use three different prompts in a list. The generated images are displayed in a 1 × 3 grid, as shown in figure 10.3.



**Figure 10.3** Three images generated by Stable Diffusion. The three text prompts are “a panda eating a bowl of noodles,” “a dog with sunglasses and a straw hat,” and “a cat stretching on the floor,” respectively.

So far, we’ve used the default values of image resolution ( $512 \times 512$ ) and the classifier-free guidance (CFG) parameter (set to 7.5) in the `StableDiffusionPipeline` package. However, you can change these parameter values, as in this example:

```

torch.manual_seed(42)
prompt = "dogs running on a sandy beach under blue sky"
images = pipe(prompt,width=768,height=512,
              guidance_scale=12).images
images[0]

```

Sets the image resolution to  $768 \times 512$

Changes the CFG parameter to 12

In this example, we’ve changed the image resolution to  $768 \times 512$  and set the CFG parameter, `guidance_scale`, to 12. The output is shown in figure 10.4.

The output shows four dogs running on a sandy beach under a blue sky. The generation takes just a few seconds.





**Figure 10.4** An image generated by Stable Diffusion using the prompt “dogs running on a sandy beach under blue sky.” The image resolution is 768 × 512, and the CFG parameter is set to 12.

**NOTE** Keep in mind that generating multiple high-resolution images can quickly use up large amounts of GPU memory, especially when working with larger image sizes or batch generation. If your GPU runs out of memory, you may need to reduce the resolution or generate images one at a time. The `guidance_scale` parameter (often referred to as the CFG parameter) controls how strongly the generated image is steered by your text prompt versus being allowed to “free-associate” without guidance. A higher `guidance_scale` encourages the model to adhere more closely to your prompt, while a lower value allows for more creativity and randomness in the output. For a deeper explanation of conditional versus unconditional generation and how CFG works, refer to chapter 6.

Now that you know what Stable Diffusion can accomplish, let’s explore various components of the model and how they contribute to the text-to-image generation process. We’ll start with its architecture.

## 10.2 *The Stable Diffusion architecture*

As mentioned earlier, Stable Diffusion relies on latent diffusion models (LDMs), a method we explained previously in chapter 9. While diffusion techniques perform very well for image generation, they normally require adding and removing noise over many iterations (often more than 1,000), which makes the process very computationally demanding. In addition, handling high-resolution images directly in pixel form uses a lot of memory, complicating both training and running the model.

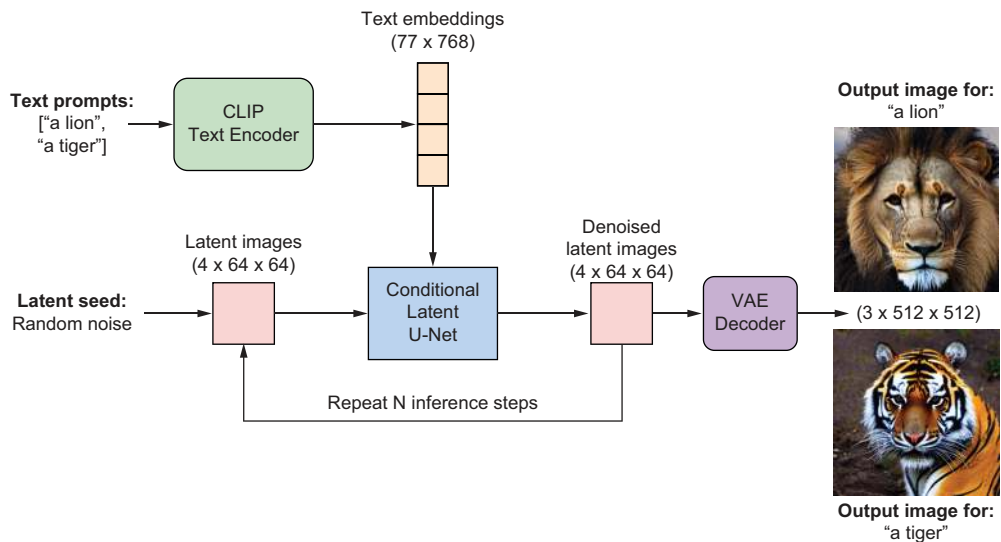


To overcome these issues, latent diffusion shifts most of the computation from the high-dimensional pixel space to a smaller, more efficient latent space. This approach significantly lowers memory usage and computational costs. Because operating in a lower-dimensional space reduces image quality, a variational autoencoder (VAE) is used to upscale and refine the latent representations, ensuring the final images are detailed and match the text prompts well.

In this section, we explain how different components of Stable Diffusion work together to transform a text prompt into a clear image. We also introduce a method for blending text embeddings from two separate prompts, creating a smooth transition between the resulting images. This experiment offers further insight into how Stable Diffusion operates.

### 10.2.1 Generating images from text with Stable Diffusion

Stable Diffusion has three main components: a text-encoder, a conditional denoising U-Net, and an autoencoder (VAE). Consider a scenario where you want to generate two images based on two different text prompts: “a lion” and “a tiger” (see top left of figure 10.5). The process involves several steps using the three main components.



**Figure 10.5** How Stable Diffusion converts text prompts to images. Stable Diffusion has three main components: a text encoder, a U-Net, and a VAE decoder. The text encoder converts text prompts such as “a lion” or “a tiger” into text embeddings. The U-Net conducts the reverse diffusion process in the latent space, using the text embeddings as the conditional information. The generated latent images are then converted by the VAE decoder into high-resolution images as outputs.

The first step is to convert the text prompts into a format that the image generation model can understand. This is achieved by using the CLIP model’s text encoder.

Here's how it works. Each text prompt is broken down into individual tokens. For instance, the prompt "a lion" is split into the tokens [a, lion]. The CLIP text encoder is designed to work with a fixed sequence length of 77 tokens. If a prompt contains fewer than 77 tokens (as in our case), it's padded with special tokens to reach the required length. On the other hand, if the prompt is too long, it will be truncated to fit into the 77-token limit. Each token is then transformed into a 768-dimensional vector via a word embedding process. This embedding captures the semantic meaning of the token, resulting in a text embedding matrix of shape  $77 \times 768$  for each prompt.

The second step involves image generation in the latent space. The generation starts with an image represented by a tensor of shape (4, 64, 64). This tensor is initialized with pure random noise, serving as the starting point for the diffusion process. The model then enters a reverse diffusion process, where it iteratively refines the noisy image. At each iteration, noise is gradually removed from the image, with the refinement process being conditioned on the text embeddings. This ensures that the refined image increasingly aligns with the semantic content of the text prompt. Typically, this process is repeated for a fixed number of iterations (e.g., 50 inference steps), each progressively enhancing the image quality and fidelity.

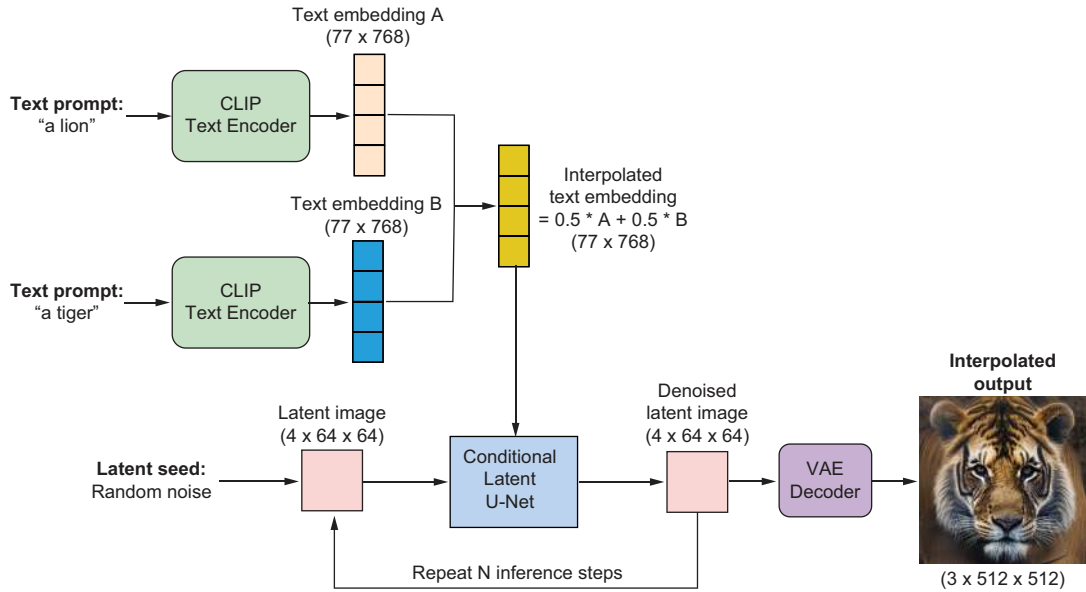
The third step is to convert latent images into high-resolution output. The denoised latent image is passed through a pretrained VAE decoder. The VAE is trained to reconstruct detailed images from compact latent representations, thus producing the final high-resolution images that reflect the input prompts. As you may recall, we've briefly discussed how VAEs work in chapter 9 and will provide a more formal introduction to VAEs in chapter 11.

The complete pipeline, from text tokenization and embedding via CLIP, through the iterative conditional reverse diffusion process, to the final VAE decoding, produces images that closely match the original text descriptions. As shown on the right of figure 10.5, the generated images clearly show a lion and a tiger, demonstrating the ability of this multistep approach to create images directly from textual input.

### 10.2.2 *Text embedding interpolation*

In the latent space, the image generation process begins by feeding an initial noise tensor into a U-Net. This network progressively refines the random noise into a structured, detailed image, all while being guided by a specific text embedding. Because the final image is conditioned on the text embedding, changing the embedding produces different outcomes. This dependency enables us to explore and understand the relationship between the text embedding and the resulting image.

We investigate this relationship through interpolation between different text embeddings. By blending two embeddings in various proportions, we can create a series of intermediate embeddings. Each of these blended embeddings, when used to condition the reverse diffusion process, generates a unique image that gradually shifts from one concept to another. Figure 10.6 demonstrates this technique within the context of Stable Diffusion.



**Figure 10.6** Using text embedding interpolation to create hybrid images in Stable Diffusion. Suppose you have two different prompts, “a lion” and “a tiger.” We first use the CLIP text encoder to convert them into two text embeddings, A and B. We can blend A and B and create an interpolated text embedding, which is then used to guide the U-Net in the reverse diffusion process in the latent space. The generated latent images are then converted into high-resolution images as the final output, exhibiting characteristics of both a lion and a tiger (bottom right).

Consider the two prompts: “a lion” and “a tiger.” As shown in the top left of figure 10.6, the “a lion” prompt is encoded by the CLIP text encoder into a text embedding (denoted as A), while “a tiger” is encoded into another embedding (denoted as B).

To interpolate between these two embeddings, we use a blending equation. Let  $z_A$  represent the embedding for “a lion” and  $z_B$  for “a tiger.” An interpolated embedding  $z_\alpha$  is defined as

$$z_\alpha = \alpha z_A + (1 - \alpha) z_B \quad (10.1)$$

where  $\alpha$  is a blending coefficient that varies between 0 and 1, allowing for a gradual transition from one image to another.

In the middle of figure 10.6, we interpolate the text embeddings A and B by placing equal weights on them (i.e., setting  $\alpha = 0.5$  in equation 10.1). We then use the interpolated text embedding as the conditioning factor in the reverse diffusion process in the latent space, and the output is an image that has characteristics of both a lion and a tiger (see bottom right in the figure).

By generating mixed text embeddings at different interpolation levels (e.g.,  $\alpha = 0.9, 0.7, \dots, 0.1$ ), we can produce a sequence of images that transition smoothly from one

concept to another. This technique may offer practical benefits for various users who can fine-tune the output by adjusting the interpolation parameter, creating hybrid images that seamlessly merge features from multiple prompts. Using interpolated embeddings, it's possible to generate animations that transition fluidly between different visual concepts, opening up new avenues for motion graphics. Artists and designers can harness text embedding interpolation to experiment with creative blends, generating novel variations that merge characteristics in unexpected and inspiring ways.

### 10.3 Creating text embeddings

As we mentioned earlier, Stable Diffusion consists of three main parts: a text encoder, a U-Net, and a VAE decoder. The first step in text-to-image generation is to convert the text prompts to text embeddings. These embeddings are used as the conditioning factors when generating latent images.

In this section, we discuss how text embeddings are generated, using the two prompts, “a lion” and “a tiger,” as our examples. To that end, we first create a `Config()` class to store the hyperparameters used in the model. We also define the two example prompts:

```
class Config:
    height = 512
    width = 512
    guidance_scale = 7.5
    num_inference_steps=50
config=Config()

prompts1 = ["a lion"]
prompts2 = ["a tiger"]
```

The height and width of the generated images are 512 pixels. We’ll use 50 inference steps. The CFG scale is set at 7.5, which is the default value of the parameter in the `StableDiffusionPipeline` package in the `diffusers` library. We import the pretrained CLIP tokenizer and text encoder from the `transformers` library:

```
from transformers import CLIPTextModel, CLIPTokenizer
import torch

device = "cuda" if torch.cuda.is_available() else "cpu"
tokenizer = CLIPTokenizer.from_pretrained(
    "openai/clip-vit-large-patch14")
text_encoder = CLIPTextModel.from_pretrained(
    "openai/clip-vit-large-patch14").to(device)
```

Now we can tokenize the two text prompts and convert them into text embeddings:

```
tokens1=tokenizer(prompts1,padding="max_length",
                  max_length=tokenizer.model_max_length,
```

```

        truncation=True, return_tensors="pt")
tokens2=tokenizer(prompts2,padding="max_length",
                  max_length=tokenizer.model_max_length,
                  truncation=True, return_tensors="pt")
with torch.no_grad():
    text_embeddings1=text_encoder(
        tokens1.input_ids.to(device))[0]
    text_embeddings2=text_encoder(
        tokens2.input_ids.to(device))[0]
print("text embedding 1 shape:", text_embeddings1.shape)
print("text embedding 2 shape:", text_embeddings1.shape)

```

← Converts the two prompts into sequences of indices

← Generates two text embeddings

The output is

```

text embedding 1 shape: torch.Size([1, 77, 768])
text embedding 2 shape: torch.Size([1, 77, 768])

```

Both text embeddings have a shape of (1, 77, 768): one prompt in each batch, with 77 tokens for each prompt, and each token represented by a 768-value tensor. As we explained in chapter 9, the CLIP text encoder is designed to accept inputs of a fixed length (77 tokens). If a prompt is shorter than 77 tokens, it gets padded.

### Exercise 10.2

Define `prompt3` and `prompt4` as “a leopard” and “a lioness,” respectively. Then tokenize the two prompts, and convert them to text embeddings, `text_embedding3` and `text_embedding4`.

We also create an unconditional text embedding for CFG:

```

unconditional_prompt = [""]
uncond_input = tokenizer(unconditional_prompt,
                        padding="max_length",
                        max_length=tokenizer.model_max_length,
                        return_tensors="pt")
with torch.no_grad():
    uncond_embeds=text_encoder(
        uncond_input.input_ids.to(device))[0]

```

← The unconditional prompt is an empty string.

← Tokenizes the unconditional prompt

← Converts the unconditional prompt into a text embedding

The unconditional prompt is nothing more than an empty string that we tokenize and convert into a text embedding. When we use it to guide image generation in the latent space, the text embedding leads to unconditional image generation. We'll concatenate the unconditional text embedding and the prompt text embeddings to form the final text embeddings. As we explained in chapter 6, the final output is a combination of conditional generation and unconditional generation. The CFG parameter controls how much the final output relies on conditional generation as opposed to unconditional generation.

## 10.4 Image generation in the latent space

As we discussed previously, Stable Diffusion performs image generation in a lower-dimensional latent space to dramatically reduce memory requirements and computation time. In this section, we'll walk through the practical steps of generating images in the latent space using the K-LMS scheduler from the K-Diffusion models and explain several subtleties that are important for high-quality results.

A crucial aspect of diffusion-based image generation is the choice of scheduler, which controls how noise is removed from the latent image over a sequence of inference steps. Different schedulers, such as Denoising Diffusion Implicit Model (DDIM) and Linear Multi-Step (LMS; used here), implement different numerical solvers for the reverse diffusion process.

Schedulers can significantly influence both the inference speed (how fast an image is generated) and the visual fidelity (how realistic and detailed the output appears). For instance, the DDIM scheduler is generally faster and allows for fewer steps but may sometimes produce slightly lower-quality images compared to LMS. The LMS scheduler often yields higher visual fidelity, especially when generating complex or high-resolution images, though it may take a bit more time per image.

**NOTE** If you want quicker results, try using a scheduler such as DDIM with fewer steps. For the highest quality, LMS with more steps can be preferable, at the cost of increased computation time.

We'll use the K-LMS scheduler, a type of sampling scheduler used in the context of K-Diffusion models. The LMS methods are a class of numerical solvers for differential equations, which are integral to diffusion models. They aim to estimate the value of the solution at a new point by using a linear combination of previously computed points. This makes LMS methods well-suited for modeling the iterative nature of diffusion processes. In diffusion models, the goal is to learn a reverse diffusion process that starts from pure noise and progressively denoises it to generate realistic samples. K-Diffusion models incorporate a series of steps, each refining the generated sample further. The K-LMS scheduler specifically focuses on efficiently handling these iterative steps. First, we import the K-LMS scheduler and the pretrained conditional denoising U-Net from the `diffusers` library:

```
from diffusers import LMSDiscreteScheduler
from diffusers import UNet2DConditionModel

scheduler = LMSDiscreteScheduler(beta_start=0.00085,
                                  beta_end=0.012,
                                  beta_schedule="scaled_linear",
                                  num_train_timesteps=1000)

UNET_PATH = "CompVis/stable-diffusion-v1-4"
UNET_SUBFOLDER = "unet"
UNET_DEVICE = "cuda:0"

unet = UNet2DConditionModel.from_pretrained(
    UNET_PATH, subfolder=UNET_SUBFOLDER)
unet = unet.to(UNET_DEVICE)
```

Creates the K-LMS scheduler

Creates the pretrained denoising U-Net

We'll start with pure noise tensors. These noise tensors have the same shape, (4, 64, 64), as the final clean image in the latent space. We then denoise the image for 50 inference steps. These steps are implemented in the `gen_latents()` function.

### Listing 10.2 Generating images in the latent space

```
from tqdm import tqdm

def gen_latents(text_embeddings, seed):
    torch.manual_seed(seed)
    scheduler.set_timesteps(config.num_inference_steps)
    latents = torch.randn((1, unet.config.in_channels,
                           config.height // 8, config.width // 8)).to(device)
    latents = latents * scheduler.init_noise_sigma
    text_embeddings=torch.cat([uncond_embeds, text_embeddings])
    with torch.autocast(device):
        for i, t in tqdm(enumerate(scheduler.timesteps)):
            latent_model_input = torch.cat([latents] * 2)
            sigma = scheduler.sigmas[i]
            latent_model_input = scheduler.scale_model_input(
                latent_model_input, t)
            with torch.no_grad():
                noise_pred = unet(latent_model_input, t,
                                encoder_hidden_states=text_embeddings).sample
                noise_pred_uncond, noise_pred_text = noise_pred.chunk(2)
                noise_pred=noise_pred_uncond+config.guidance_scale*(
                    noise_pred_text - noise_pred_uncond)
            latents = scheduler.step(noise_pred,
                                    t, latents).prev_sample
    return 1 / 0.18215 * latents
```

The starting image consists of pure random noise.

Iteratively denoises the image for 50 inference steps

Divides the output by 0.18215 to undo a previous scaling during training

The `gen_latents()` function takes two arguments, `text_embeddings` and `seed`. The first argument is the text embedding of the text prompt, while the `seed` argument ensures that the output is reproducible.

You may be wondering why we scale the final output by 0.18215 in the `gen_latents()` function. In Stable Diffusion and other LDMs, the latent images are divided by a factor of 0.18215 to reverse a scaling applied during training. When these models are trained, an input image is first encoded into a latent representation, and then this representation is multiplied by 0.18215 to normalize its range to ensure training stability and convergence. When a new latent sample is generated, dividing by 0.18215 undoes the scaling, returning the data to the original image space.

**NOTE** The value 0.18215 is not chosen at random. During VAE training, researchers found that encoding real images into the latent space produced latent vectors with a standard deviation of about 0.18215, rather than 1. By dividing the latent vectors by 0.18215, we normalize their standard deviation to 1. This normalization ensures that the decoder receives inputs consistent

with what it was exposed to during training, which is crucial for producing high-quality image reconstructions.

Now we can apply the `gen_latents()` function on the two text embeddings we generated in the previous section to obtain two latent images for the text prompts “a lion” and “a tiger”:

```
latents1=gen_latents(text_embeddings1,seed=5)
latents2=gen_latents(text_embeddings2,seed=5)
print(f"latent image size is {latents1[0].shape}")
```

The output is

```
latent image size is torch.Size([4, 64, 64])
```

The two latent images have a shape of (4, 64, 64). We’ll convert them into high-resolution images in the next section.

### Exercise 10.3

Apply the `gen_latents()` function on the two text embeddings `text_embedding3` and `text_embedding4`, which you generated earlier. Set the seed value to 8. Name the latent images `latents3` and `latents4`.

## 10.5 *Converting latent images to high-resolution ones*

We’ll use the pretrained VAE to convert the low-resolution images in the latent space to high-resolution ones as the final output of the Stable Diffusion model. In the following code cell, we import the pretrained VAE from the `diffusers` library:

```
from diffusers import AutoencoderKL

vae = AutoencoderKL.from_pretrained(
    "CompVis/stable-diffusion-v1-4",
    subfolder="vae").to(device)
```

Now we can use the VAE decoder to convert latent images to high-resolution ones:

```
with torch.no_grad():
    images1 = vae.decode(latents1).sample
    images2 = vae.decode(latents2).sample
print(f"final image size is {images2[0].shape}")
```

The output is

```
final image size is torch.Size([3, 512, 512])
```

After VAE decoding, each image has a shape of  $3 \times 512 \times 512$ .



**Exercise 10.4**

Convert the two latent images, `latents3` and `latents4`, into high-resolution images. Name them `images3` and `images4`, respectively.

We can plot the two latent images and the two final high-resolution images in a  $1 \times 4$  grid. The following listing shows how this is done.

**Listing 10.3 Visualizing latent images and final outputs**

```
import matplotlib.pyplot as plt
import numpy as np

def latent_to_display_image(latent, upsample_factor=8):
    image = latent[0].permute(1, 2, 0).detach().cpu()
    image = torch.clamp(image, -1, 1)
    image = image.repeat_interleave(upsample_factor,
                                    dim=0).repeat_interleave(upsample_factor, dim=1)
    image = (image + 1) / 2
    return image.numpy()

def output_to_display_image(output):
    image = output[0].permute(1, 2, 0).detach().cpu()
    image = (image + 1) / 2
    image = torch.clamp(image, 0, 1)
    return image.numpy()

display_imgs = [
    latent_to_display_image(latents1),
    output_to_display_image(images1),
    latent_to_display_image(latents2),
    output_to_display_image(images2)]

captions = [
    "latent image\n a lion",
    "final output\n a lion",
    "latent image\n a tiger",
    "final output\n a tiger"]

plt.figure(figsize=(8, 5), dpi=100)
for i in range(4):
    plt.subplot(1, 4, i + 1)
    plt.imshow(display_imgs[i])
    plt.title(captions[i], fontsize=15)
    plt.axis('off')
plt.tight_layout()
plt.show()
```

← Changes the latent image to a displayable format

← Changes the final output image to a displayable format

← Places the four images in a list

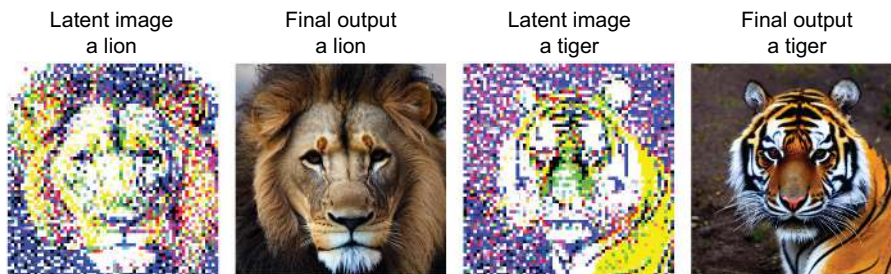
← Creates four captions

← Plots the four images in a  $1 \times 4$  grid

We define the `latent_to_display_image()` function to convert a latent image tensor into a format suitable for visualization. This function performs several operations: it rearranges the tensor from channels-first to channels-last order, clips the values to the

range  $[-1, 1]$ , detaches the tensor from the computation graph, and moves it from the GPU to the CPU for compatibility with visualization libraries. Additionally, it upsamples the latent image so that all displayed images have the same height and width, ensuring they align properly in the same row. The `output_to_display_image()` function follows a similar process, but because the output image is already at the desired resolution, it doesn't require upsampling.

We place the formatted latent images and final output images in the list `display_imgs`. We also create four captions, one for each image. After running the code listing, you'll see the image shown in figure 10.7.



**Figure 10.7** Latent images and final outputs from Stable Diffusion. We use “a lion” and “a tiger” as the text prompts. The Stable Diffusion model first converts the two prompts into two text embeddings. The model then conducts the reverse diffusion process in a latent space, guided by the text embeddings. The generated latent images are shown as the first and the third image in this figure. The latent images are converted to high-resolution ones by using a VAE decoder. The high-resolution final outputs are shown as the second and the fourth images in this figure.

The first and the third images in figure 10.7 are the latent images. Even though they have low resolution, we can tell that they are a lion and a tiger, respectively. This indicates that the latent images have captured the essence of the two text prompts. The VAE decoder then converts these two latent images into high-resolution ones, as shown in the second and fourth subplot in figure 10.7.

### Exercise 10.5

Plot the two latent images, `latents3` and `latents4`, and the two outputs, `images3` and `images4`, in a  $1 \times 4$  grid. Label the four images accordingly.

We can interpolate the two text embeddings, `text_embeddings1` and `text_embeddings2`, to create new embeddings with various weights:

```
images=[]
for weight in [0.9, 0.7, 0.5, 0.3, 0.1]:
    text_embeddings = weight * text_embeddings1 + \
```

← Chooses five different values of weights

```

(1-weight) * text_embeddings2
latents=gen_latents(text_embeddings,seed=5)
with torch.no_grad():
    images.append(vae.decode(latents).sample)

```

Creates an interpolated text embedding based on the weight

Generates a latent image based on the text embedding

Converts the latent image into a high-resolution image

The variable weight is the weight we place on the text embedding for the “a lion” prompt. It takes five values, 0.9, 0.7, 0.5, 0.3, and 0.1. The complementary weight is placed on the text embedding for the “a tiger” prompt. The interpolated text embedding is fed to the `gen_latents()` function to generate a latent image, which is then converted into a high-resolution image using the VAE decoder. The five high-resolution images are placed in a list `images`. We plot the five interpolated images in a  $1 \times 5$  grid, and the output is shown in figure 10.8:

```

plt.figure(figsize=(8,5),dpi=100)
for i in range(5):
    plt.subplot(1,5,i+1)
    img=torch.clip(images[i][0].detach().cpu(
        ).permute(1,2,0)/2+0.5,0,1)
    plt.imshow(img)
    plt.title(f"{90-20*i}% lioness\n {10+20*i}% leopard",
        fontsize=15)
    plt.axis('off')
plt.tight_layout()
plt.show()

```

Plots the five images in a  $1 \times 5$  grid

Labels the images



**Figure 10.8** The text embeddings for “a lion” and “a tiger” are combined with different weights to form a series of interpolated text embeddings. The Stable Diffusion model then conducts the reverse diffusion process in the latent space, guided by the interpolated text embeddings. The latent images are converted to high-resolution ones by using a VAE decoder. The high-resolution final outputs are shown in this figure, with different weights placed on the two text embeddings.

The first image shows the final output when we use 90% of the text embedding for “a lion” and 10% of the text embedding for “a tiger” to create the interpolated embedding. As we move to the right, the weight on the text embedding for “a lion” decreases

and that on the text embedding for “a tiger” increases. As a result, when you move from left to right, the images show more characteristics of a tiger and fewer characteristics of a lion.

### Exercise 10.6

Let the variable `weight` take five values, 0.9, 0.7, 0.5, 0.3, and 0.1. Create five interpolated text embeddings of `text_embeddings3` and `text_embeddings4` based on these weights. Feed the interpolated text embeddings to the `gen_latents()` function to generate five latent images. Convert the five latent images to high-resolution ones using the VAE decoder. Plot the five high-resolution images in a  $1 \times 5$  grid, and label them accordingly.

By now, you should have a thorough understanding of text-to-image generation with diffusion models. In the next chapters, you’ll learn how to replicate DALL-E, a transformer-based text-to-image generation model.

### Summary

- Stable Diffusion is based on the latent diffusion model (LDM) that we discussed in the previous chapter. It’s one of the state-of-the-art text-to-image models and the only one that is open source and free for anyone to use.
- Stable Diffusion is developed jointly by CompVis, Stability AI, and LAION. Stable Diffusion has incorporated improvements that enhance both the model’s stability and performance, such as advanced training techniques and optimizations. While the original LDM was trained on the LAION-400M dataset (with 400 million image-text pairs), Stable Diffusion was trained on a subset of the even larger LAION-5B database.
- The `StableDiffusionPipeline` package in the `diffusers` library allows us to use Stable Diffusion as an off-the-shelf tool to generate extraordinary images in just a few lines of code.
- By blending the embeddings of two different text prompts, we can create an interpolated text embedding. If we use this embedding to guide the latent diffusion process, we can obtain an interpolated latent image, which can in turn be converted to an interpolated final output image. The output image will have characteristics of the two original text prompts. For example, interpolating between the embeddings for “a lion” and “a tiger” will produce an image with characteristics of both a lion and a tiger.

# *Text-to-image generation with transformers*

**W**hile diffusion models dominate today's landscape, transformer-based approaches to text-to-image generation remain highly influential and offer a different perspective. In chapter 11, we dive deep into the vector quantized generative adversarial network, which transforms images into discrete sequences of integers, enabling transformers to handle images in the same way they process text.

Chapter 12 implements a minimal version of DALL-E, a groundbreaking transformer-based model that first demonstrated the power of generating images directly from natural language descriptions.



# 11

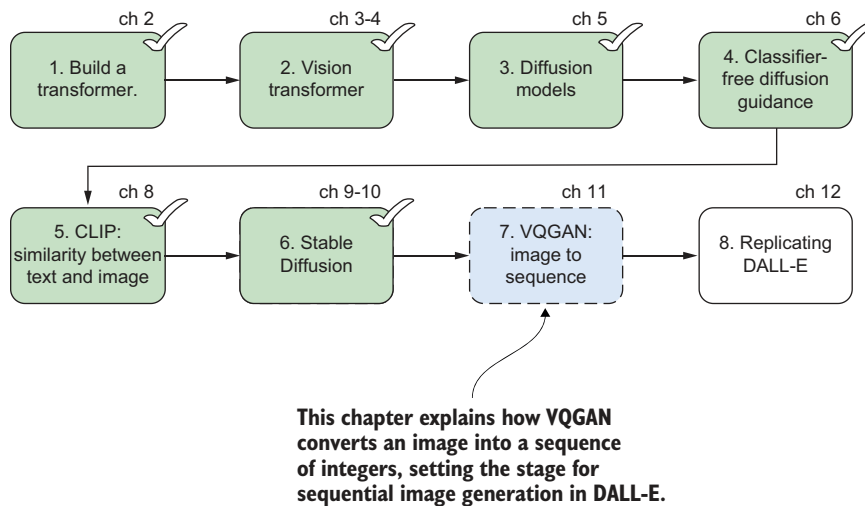
## *VQGAN: Convert images into sequences of integers*

---

### ***This chapter covers***

- Encoding images into continuous latent representations
- Quantizing latent representations into discrete codes using a codebook
- Reconstructing images from discrete sequences
- Understanding perceptual loss, adversarial loss, and quantization loss

Modern transformer-based text-to-image models, such as DALL-E, rely on a crucial step: transforming images into sequences of discrete tokens, just as language models treat text as sequences of word tokens. Figure 11.1 shows how this step fits into the larger journey of building a text-to-image generator. In this chapter, we zero in on step 7, where vector quantized generative adversarial network (VQGAN) bridges the gap between images and language-like data, making images accessible to transformers.



**Figure 11.1** Eight steps for building a text-to-image generator from scratch. This chapter focuses on step 7: transforming an image into a sequence of integers using VQGAN. By achieving this, we unlock the ability to generate images sequentially with transformer models, a critical advance that powers state-of-the-art systems such as DALL-E.

Transformers revolutionized language by breaking text into sequences of integers (tokens), enabling powerful generative models. To bring this power to images, we need a method to convert visual data into an analogous sequence of discrete codes. This is exactly what the VQGAN accomplishes. By encoding images into compact latent representations and quantizing these representations into codebook indices, VQGAN enables images to be processed and generated *token by token*, setting the foundation for models such as DALL-E, which you’ll tackle in the next chapter.

In this chapter, you’ll learn how VQGAN works. We’ll break down VQGAN into its components: an encoder that maps images into latent space, a quantizer that assigns each patch to a codebook entry, and a decoder that reconstructs images from sequences of code indices.

Essentially, VQGAN is a generative adversarial network (GAN) where the generator is a vector quantized variational autoencoder (VQ-VAE). Therefore, you’ll first learn how GANs and VQ-VAEs work, mastering the needed context to understand VQGAN. You’ll then explore a pretrained VQGAN model, learning step-by-step how images are encoded into integer sequences and how these sequences are decoded back into high-resolution images. This discrete representation enables powerful downstream applications, particularly in transformer-based text-to-image models such as DALL-E, where images are treated analogously to sequences of words or tokens.

By the end of this chapter, you’ll have a solid understanding of the inner workings of VQGANs and the practical skills to use pretrained models for image reconstruction



and transformation. This knowledge sets the stage for the subsequent chapter, where you'll apply VQGAN to replicate DALL-E, turning text into images that match the description.

## 11.1 Converting images into sequences of integers and back

VQGAN, introduced by Esser et al. (2021), merges the strengths of GANs and VQ-VAEs [1]. GANs excel at producing realistic and high-quality images through adversarial training between two neural networks: a generator and a discriminator. However, GANs often face training instabilities and produce latent spaces that lack interpretability. VQ-VAEs, on the other hand, address these issues by quantizing continuous latent representations into discrete, interpretable codes via a learned codebook. This discretization process results in more stable training and structured latent spaces that are particularly suitable for symbolic reasoning tasks, such as transformer-based modeling. That said, pure GANs can still be preferred in scenarios where maximum photorealism is the primary goal and where interpretability or latent-space structure is less critical.

The ability of VQGAN to convert an image into a sequence of integers and then back into an image is crucial for transformer-based text-to-image generation models. To give you a taste of what a VQGAN model can accomplish, let's first treat it as a black box. We'll use a pretrained VQGAN to convert an image to a sequence of integers.

To get started, run the following command in a new Jupyter Notebook cell to install the required libraries:

```
!pip install omegaconf pytorch-lightning
```

**NOTE** The Python programs for this chapter can be accessed from the GitHub repository (<https://github.com/markhliu/txt2img>) and the Google Colab notebook (<https://mng.bz/qR6x>). Don't just copy and paste the book's code snippets into a new Colab notebook and run them. Instead, use the Colab notebooks I've provided, as Google Colab has a unique filesystem that requires specific setup.

Go to the book's GitHub repository, download the image `fish.png`, and place it in the `/files/` folder on your computer. In addition, download the local module `VQGAN-util.py`, and place it in the `/utils/` folder (or simply clone the repository to your computer). The module defines a few helper functions that will be implemented and explained later in this chapter.

We'll use the pretrained VQGAN model developed by the Computer Vision and Learning research group at Ludwig Maximilian University of Munich through its GitHub repository (<https://github.com/CompVis/taming-transformers>). Due to recent changes in PyTorch, the repository requires small updates to remain compatible. Rather than asking every one of you to edit the source code manually, I've created a fork of the repository with these compatibility fixes already applied.

To clone the modified GitHub repository that contains the VQGAN implementation and copy all of its files to a local folder, run the following command in a new cell:

```
!git clone https://github.com/markhliu/taming-transformers
```

After this completes, you should see a new folder named `/taming-transformers/` in your working directory. To make sure Python can access and import modules from this repository, add this folder to Python's search path using the `sys` library:

```
import sys
sys.path.append("./taming-transformers")
```

The following code listing loads the pretrained VQGAN model and converts the fish image into a sequence of integers. The model then converts the sequence back into a fish image.

#### Listing 11.1 Converting an image into a sequence of integers

```
import matplotlib.pyplot as plt
from utils.VQGANutil import (load_model, process_image,
                             image_to_sequence, sequence_to_image)

model=load_model()
image=process_image("files/fish.png")
sequence=image_to_sequence(model,image)
print("the image is converted to:\n", sequence)
reconstruct=sequence_to_image(model,sequence)
imgs=[image[0].cpu().permute(1,2,0)*0.5+0.5,
      reconstruct[0].cpu().permute(1,2,0)*0.5+0.5]
captions=["original","reconstructed"]
plt.figure(figsize=(8,5),dpi=100)
for i in range(2):
    plt.subplot(1,2,i+1)
    plt.imshow(imgs[i].clamp(0,1))
    plt.title(captions[i],fontsize=25)
    plt.axis('off')
plt.tight_layout()
plt.show()
```

← Loads the pretrained VQGAN (may take a few minutes)

← Converts the image into a sequence of integers

← Prints out the sequence

← Reconstructs the image based on the sequence

← Displays the original and reconstructed images

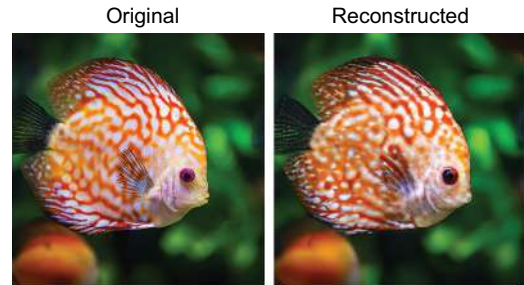
We start by loading a pretrained VQGAN model using the `load_model()` function provided in the local module `VQGANutil.py`. This initial step may require several minutes, as it involves downloading a large pretrained weights file exceeding 1 GB in size.

Next, the model converts the input fish image into a discrete sequence of integers. The resulting sequence is illustrated here (some integers are omitted for brevity):

```
the image is now represented by this:
tensor([ 508,  343,  261,  508,  156,  596,  634,  534,  508,  501,
        253,  970, ...,  47,  187,  313,  596,  564,  261,   96,   29,  299,
        534,  330,  542],device='cuda:0')
```

This sequence consists of 576 integers (I'll explain the reason behind this specific number shortly). The pretrained VQGAN model then reconstructs a high-resolution image from this integer sequence, closely resembling the original fish image, as shown in figure 11.2.

You may be wondering how the VQGAN encoder transforms an image into a sequence of integers, what these integers represent, and how the decoder reconstructs a high-resolution image from them. In the remainder of this chapter, we dive deeper into these questions, examining the VQGAN model in detail and illustrating precisely how each component works.



**Figure 11.2** A pretrained VQGAN model reconstructs an image. The left image shows the original fish image. The VQGAN encoder transforms this image into a sequence of 576 integers, representing its compressed latent code. The VQGAN decoder then reconstructs the image from this sequence, producing the version shown on the right. The reconstructed image closely resembles the original, demonstrating the effectiveness of the VQGAN in capturing and preserving key visual features through discrete latent representations.

## 11.2 Variational autoencoders

To understand VQGAN, it's important to first become familiar with VQ-VAE, a specialized type of variational autoencoder (VAE). VAEs are foundational to many state-of-the-art text-to-image generation models. In chapters 9 and 10, we used pretrained VAEs to decode latent representations into high-resolution images, serving as the final step in latent diffusion models (LDMs). In this chapter, we formally introduce VAEs, which are crucial for grasping more advanced architectures such as VQGAN.

To build a solid understanding of VAEs, we must first explore autoencoders, as VAEs represent a significant advance over traditional autoencoders. While standard autoencoders excel at learning compact representations of data, VAEs incorporate probabilistic modeling, enabling not only efficient encoding but also powerful generative capabilities. This section explains the evolution from autoencoders to variational autoencoders, highlighting their differences, advantages, and practical uses.

Due to space constraints, we won't cover building and training autoencoders or VAEs from scratch in this book. If you're interested in a comprehensive, hands-on introduction to implementing these models, I recommend chapter 7 of my book *Learn Generative AI with PyTorch* (Manning, 2024), which offers step-by-step instructions and examples.

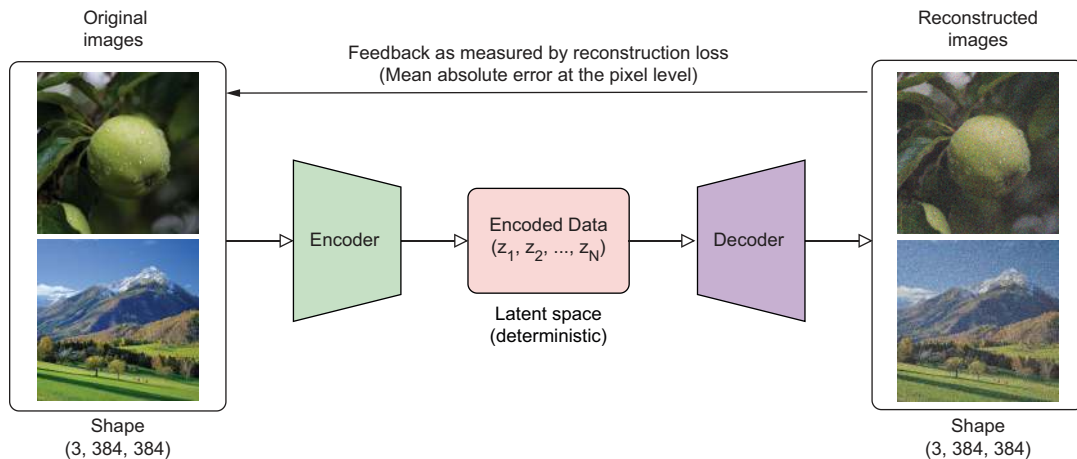
### 11.2.1 What is an autoencoder?

Autoencoders are neural networks used primarily in unsupervised learning and are particularly effective for tasks such as image compression, denoising, and representation learning. An autoencoder consists of two main components:

- *Encoder*—Compresses the input data into a lower-dimensional representation in the latent space
- *Decoder*—Reconstructs the original input from this compressed representation

The latent space captures essential features of the input data. In image-generation tasks, for example, it encodes critical visual characteristics learned during training. Autoencoders efficiently process unlabeled data, making them valuable for dimensionality reduction and feature extraction. However, they face challenges such as potential information loss during encoding, leading to less accurate reconstructions.

Consider building an autoencoder designed to reconstruct high-resolution color images (e.g., with a size of  $3 \times 384 \times 384$ , representing three color channels, 384 pixels in height and width). Figure 11.3 illustrates the architecture of an autoencoder and the associated training steps. During training, input images are first passed through the encoder, which compresses them into deterministic latent representations (latent vectors). These latent vectors are then passed through the decoder, which reconstructs the original images. During training, the autoencoder continually adjusts its parameters by minimizing the reconstruction loss, calculated as the difference between the original images and their reconstructed versions.



**Figure 11.3** The architecture and training workflow of an autoencoder designed to generate high-resolution color images by using its two primary components: an encoder (shown on the left) and a decoder (shown on the right)

As you can see in the diagram, the encoder compresses color images into latent vectors, and the decoder reconstructs images from these vectors. Both encoder and decoder are deep neural networks, potentially including layers such as dense, convolutional, and transposed convolutional layers.

When an autoencoder is built, model parameters are initialized randomly. During training, images from a training set are passed through the encoder, producing

compressed latent vectors. These vectors are then decoded back into images. The reconstruction loss (mean absolute error across all pixels between the original and reconstructed images) is calculated. This loss is propagated back through the network to update encoder and decoder parameters, minimizing reconstruction loss iteratively over multiple epochs.

Once trained, the autoencoder can compress and reconstruct unseen images. However, autoencoders can't reliably generate novel images beyond their training data. Their latent spaces lack interpretability and continuity, preventing meaningful interpolation between encoded points. Thus, we turn to VAEs to address these limitations.

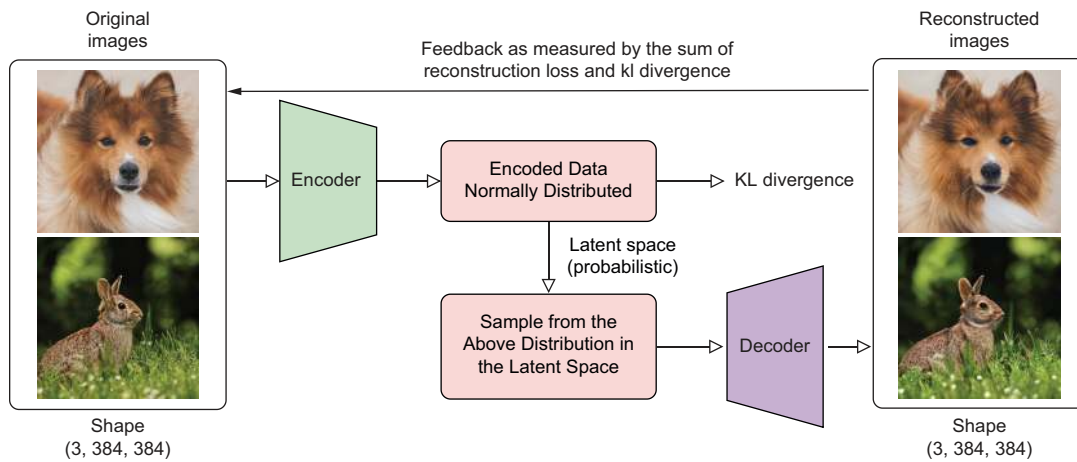
### 11.2.2 The need for VAEs and their training methodology

VAEs, introduced by Diederik Kingma and Max Welling in 2013 [2], enhance traditional autoencoders by adopting a probabilistic framework. Like autoencoders, VAEs contain an encoder and a decoder, but with two fundamental differences. The first difference lies in the latent space: whereas traditional autoencoders map an input image into a deterministic latent vector, VAEs map inputs into probabilistic distributions in the latent space. Each input image is encoded as a distribution (typically a normal distribution) parameterized by a mean ( $\mu$ ) and standard deviation ( $\sigma$ ). Sampling from these distributions generates variations in reconstructed outputs, making VAEs powerful for generative tasks.

The second difference is related to the loss function: whereas autoencoder training focuses solely on reconstruction loss, training VAEs involves minimizing both the reconstruction loss and the Kullback-Leibler (KL) divergence. *KL divergence* measures how much the learned distribution diverges from a predefined prior distribution (usually a standard normal distribution). Minimizing KL divergence ensures meaningful, generalized latent representations, preventing overfitting.

Figure 11.4 shows the architecture and training process of a VAE designed for generating high-resolution color images (with a size of  $3 \times 384 \times 384$ ). During each training iteration, original images are fed into the encoder, which compresses them into probabilistic representations characterized by mean and variance vectors in the latent space. Encodings are then sampled from these distributions and passed to the decoder. The decoder reconstructs images based on these sampled latent vectors. The VAE optimizes its parameters by minimizing a loss function composed of two terms: the reconstruction loss, which quantifies differences between the original and reconstructed images, and the KL divergence, which measures how much the encoder's latent distribution diverges from a predefined standard normal distribution.

During training, the combined reconstruction and KL divergence losses are calculated and propagated back to update model parameters. After training, the encoder can produce meaningful latent representations from new images, while the decoder can generate novel images by sampling random points from the latent space. Additionally, VAEs allow interpolation between latent vectors, enabling the creation of smooth transitions and meaningful manipulations between images.



**Figure 11.4** The architecture of a variational autoencoder (VAE) and the training steps involved in generating high-resolution color images via the VAE's two primary components: an encoder (upper-middle left) and a decoder (lower-middle right)

### 11.3 Vector quantized variational autoencoders

VQ-VAEs represent a significant advancement over traditional VAEs, primarily addressing certain shortcomings inherent to VAEs, such as blurry reconstructions and inefficient use of the latent space. Originally introduced by van den Oord et al. in 2017 [3], VQ-VAEs have become foundational in modern generative models, particularly within image synthesis. In this section, we discuss the main idea behind VQ-VAEs. We explain the motivation behind their development, highlight how they differ from traditional VAEs, and outline the step-by-step process for training a VQ-VAE.

#### 11.3.1 The need for VQ-VAEs

Traditional VAEs encode inputs into a continuous latent space characterized by probabilistic distributions, typically normal distributions. During decoding, samples from this continuous space often lead to blurry or imprecise outputs because the decoder averages over multiple possible reconstructions. This uncertainty results in images lacking sharp details and defined structures. Additionally, the continuous latent space tends to be underused, with significant regions remaining unused or sparsely populated, making the generative capability less efficient.

VQ-VAEs address these challenges by discretizing the latent space. Rather than mapping inputs to continuous distributions, a VQ-VAE maps inputs to discrete representations, or embeddings, within a finite codebook. This discretization allows the model to produce sharper reconstructions and makes the latent representations highly efficient and interpretable, significantly improving generative performance. The main difference between VAEs and VQ-VAEs lies in how latent representations are handled:



- VAEs produce continuous latent vectors drawn from probabilistic distributions (based on the mean and variance of a normal distribution), making sampling inherently stochastic and continuous.
- VQ-VAEs, in contrast, produce discrete latent vectors from a predefined codebook (vector quantization). Each encoded input is represented by its nearest embedding in this finite set, eliminating continuous uncertainty.

This difference drastically improves reconstruction sharpness and makes VQ-VAEs more suitable for downstream generative tasks, such as image synthesis, text-to-image generation, and conditional generation. Later in this chapter, you'll learn precisely what discrete latent vectors are and how encoded inputs are mapped to their nearest representations within the codebook.

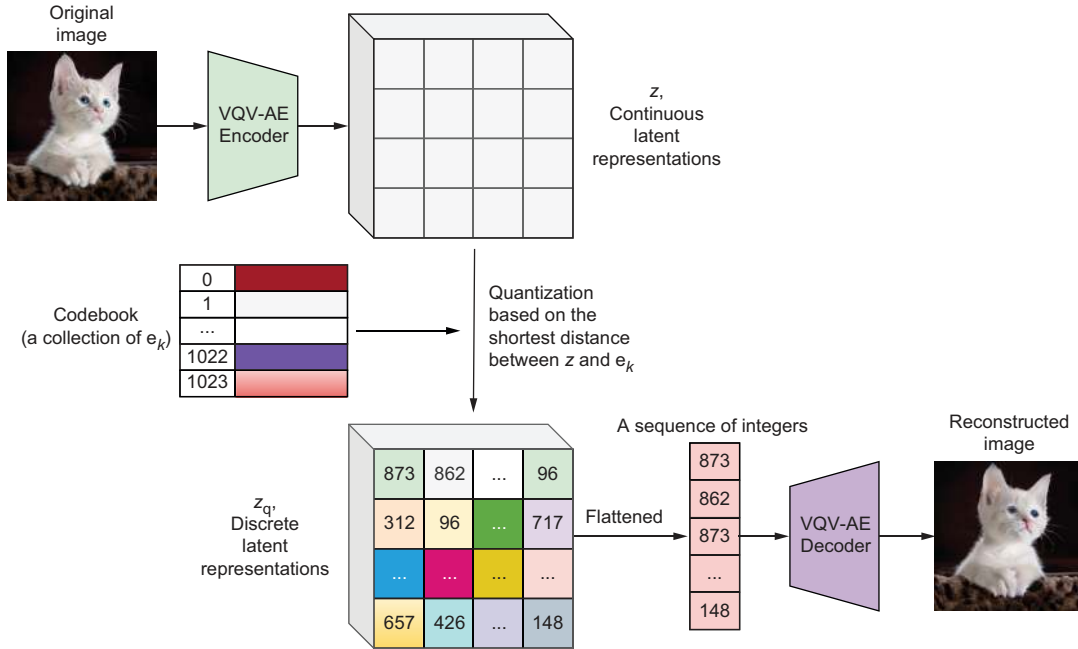
### 11.3.2 The VQ-VAE model architecture and training process

A VQ-VAE typically consists of three core components:

- *Encoder*—Similar to the encoder in a traditional VAE, the encoder in a VQ-VAE compresses the input data (e.g., images) into a latent representation. However, instead of encoding into a continuous distribution, the encoder outputs continuous latent vectors that are subsequently quantized.
- *Codebook*—A codebook is similar to an embedding space but serves a distinct purpose. While a standard embedding maps inputs to continuous vectors, a codebook defines a finite set of discrete embedding vectors in the latent space. During encoding, each latent vector is quantized (replaced by the nearest vector from this codebook). This quantized vector acts as the discrete latent code for the input. Unlike typical embeddings, the vectors in the codebook are explicitly learned to represent a compact set of prototypes, enabling the model to work with discrete representations rather than continuous ones.
- *Decoder*—The decoder in a VQ-VAE reconstructs the original input from the quantized embeddings. By working from discrete codes, the decoder achieves much clearer and more accurate reconstructions compared to the decoder in a traditional VAE.

Figure 11.5 is a diagram of the structure and training process for VQ-VAE. During training, we pass images through the encoder to obtain latent representations,  $z$ . We replace the latent vectors in  $z$  with the nearest vectors in the codebook,  $z_q$ . Therefore, the image can be represented by a sequence of integers, each pointing to the index of the feature vector in the codebook. The decoder then reconstructs the image based on the discrete latent representation  $z_q$ . We train the model to minimize the reconstruction loss and the quantization loss.

To train a VQ-VAE, we begin by assembling a large collection of high-resolution color images to serve as the training dataset. The training process starts by encoding these input images, such as the example cat image shown at the top-left corner of figure 11.5, into continuous latent representations. Specifically, each input image is compressed



**Figure 11.5** The architecture of a VQ-VAE and its three components—encoder, codebook, and decoder—in the training process

into a grid of feature vectors within the latent space. In the case of our cat image, this results in  $24 \times 24 = 576$  feature vectors, with each vector containing 256 numerical values.

Once the image is represented in this continuous latent space, the next step is to quantize these feature vectors by mapping them to discrete embeddings from a predefined codebook. In our example, this codebook contains 1,024 embedding vectors, each also consisting of 256 values. For each feature vector from the encoded image, we identify the nearest embedding vector in the codebook and replace the feature vector with this closest match. This quantization step transforms the continuous latent representations into a sequence of discrete codes.

To illustrate, let's label the 576 latent feature vectors derived from the image as  $z_0, z_1, \dots, z_{575}$ , and the 1,024 embedding vectors in the codebook as  $e_0, e_1, \dots, e_{1023}$ . For each latent vector  $z_p$ , we compute its distance to every embedding vector  $e_k$  in the codebook using the following formula:

$$\text{distance}(z_i, e_k) = \sum_{n=0}^{255} \left( z_i^{(n)} - e_k^{(n)} \right)^2 \quad (11.1)$$

Here,  $z_i^{(n)}$  and  $e_k^{(n)}$  represent the  $n$ -th component of vectors  $z_i$  and  $e_k$ , respectively. We calculate this squared Euclidean distance for all codebook vectors  $e_k$ , and replace  $z_i$



with the embedding vector that yields the smallest distance. This process is repeated for every latent vector  $z_p$ , converting the entire latent representation of the image into a sequence of 576 discrete codes.

**DEFINITION** *Quantization loss* refers to the difference between the continuous latent vectors produced by the encoder and their nearest discrete vectors from a learned codebook. Specifically, it measures how closely the encoder's output matches the chosen discrete embeddings. Minimizing this loss ensures that the encoder outputs representations that closely align with discrete codes, helping the model form stable, meaningful, and compact latent representations.

After quantization, these discrete embeddings are passed into the decoder, which reconstructs the image. The result is shown at the bottom-right corner of figure 11.5, shown previously, where the decoder outputs a version of the original cat image based on its discrete latent representation.

The VQ-VAE model is trained to minimize two key loss functions: the reconstruction loss and the vector quantization loss. The *reconstruction loss* measures the difference between original inputs and reconstructed outputs, while the *vector quantization loss* encourages encoder outputs to remain close to their assigned discrete embeddings. The VQ-VAE model parameters, including the embeddings within the codebook, are updated during training to minimize a weighted average of the two losses. This enables the model to learn both an effective discrete representation of the data and a reliable decoding process.

A significant advantage of VQ-VAE lies in its ability to convert an image into a sequence of integers, which represent the indices of the codebook embeddings. Returning to the cat image example, after quantization, the image is encoded as 576 discrete codes, where each code corresponds to an index in the codebook. For instance, the first latent vector might be replaced by the embedding at index 873, the second by index 862, and so on, resulting in a sequence of (873, 862, . . . , 148). This discrete sequence can fully represent the cat image in a compressed form. During decoding, this sequence of integers is fed back into the VQ-VAE decoder, which retrieves the corresponding embeddings from the codebook and reconstructs the image.

The ability of VQ-VAE to map images to sequences of discrete tokens, and vice versa, has profound implications for large-scale generative models such as DALL-E. In such models, images are treated analogously to text, as sequences of tokens. This enables the use of transformers, originally designed for text, to model images.

In the next chapter, you'll see how this mechanism is used in DALL-E. Specifically, DALL-E breaks images into sequences of discrete tokens (integer indices). A transformer model is trained to predict these tokens one at a time, conditioned on a given text prompt and previously generated tokens. Once the full sequence of image tokens is generated, it's passed to the pretrained VQ-VAE decoder, which converts the sequence back into an image, completing the text-to-image generation process. This approach allows DALL-E to seamlessly bridge language and vision, generating novel images that align with textual descriptions.

## 11.4 Vector quantized generative adversarial networks

GANs are among the most powerful frameworks for high-quality data generation. Introduced by Ian Goodfellow and collaborators in 2014 [4], GANs have gained widespread popularity due to their ability to create realistic and diverse outputs across domains such as image synthesis, music generation, and more. Despite their success, however, GANs face significant challenges, including training instability, difficulty in controlling output representations, and mode collapse, that limit their reliability and flexibility in practical applications.

To address these limitations, researchers have proposed integrating GANs with VQ-VAEs, leading to the development of VQGAN. VQGAN combines the strengths of both architectures: it uses the discrete latent spaces of VQ-VAEs for stable, structured encoding while using adversarial training to boost the perceptual fidelity of the generated outputs. This hybrid approach bridges the gap between the generative power of GANs and the robust, discrete representation learning of VQ-VAEs, paving the way for major advances in fields such as text-to-image generation. In this section, we first introduce the fundamentals of GANs, including their architecture and training dynamics, before diving into VQGAN, its architectural design, training procedure, and the pivotal role of its loss function.

### 11.4.1 Generative adversarial networks

GANs are a class of generative models built on a game-theoretic framework between two competing neural networks:

- *Generator*—Takes random noise, typically sampled from a normal distribution, and transforms it into synthetic data samples that aim to resemble the real data distribution
- *Discriminator*—Receives both real samples from the training set and generated samples from the generator, and then attempts to correctly classify them as real or fake

The generator and discriminator engage in a two-player minimax game:

- The discriminator seeks to maximize its ability to distinguish real samples from fakes.
- The generator aims to minimize the discriminator's ability to correctly identify fake samples.

Training proceeds iteratively, with both networks improving their abilities over time. Ideally, the generator becomes so proficient that the discriminator can no longer tell real from fake samples, achieving approximately 50% classification accuracy.

The appeal of GANs lies in their simplicity and their ability to generate highly realistic outputs. However, GANs face the challenges of training instability, mode collapse, and lack of structured representations. These limitations have motivated the search for hybrid architectures that can combine the generative power of GANs with more

structured and stable representation learning techniques, ultimately leading to the creation of VQGAN.

### 11.4.2 VQGAN: A GAN with a VQ-VAE generator

At its core, VQGAN is a GAN where the generator is replaced with a VQ-VAE. This integration allows VQGAN to harness the discrete latent representations of VQ-VAEs while retaining the powerful generative capacity of GANs. As we explained earlier, a VQ-VAE encodes input images into discrete latent codes, which are then decoded back into images. In VQGAN, these reconstructions are further refined through adversarial training with a discriminator.

This VQGAN design merges the structured, quantized latent space of VQ-VAEs (promotes semantic consistency and efficient compression) with the high-fidelity image generation capability of GANs (improves perceptual quality). VQGAN consists of three main components:

- *VQ-VAE generator*—As we mentioned in the previous section, it has an encoder, a codebook, and a decoder. The encoder maps the input image  $x$  to a continuous latent representation  $z$ . The model replaces  $z$  with the nearest vectors from a learned codebook, resulting in discrete latent codes  $z_q$ . The decoder takes  $z_q$  and reconstructs the image  $\hat{x}$ .
- *Discriminator*—This operates similarly to traditional GAN discriminators by evaluating whether images (either real or generated) are real or fake.
- *Codebook*—This is a collection of discrete embeddings  $e_k$ , where  $k = 1, 2, \dots, K$ . Here,  $K$  is the size of the codebook (e.g., 1,024). During quantization, the encoder outputs are mapped to the nearest codebook vectors.

The adversarial training with the discriminator ensures that the decoded images are not only close to the original images (in pixel space) but also perceptually realistic. The VQGAN loss function is a weighted combination of three main components: reconstruction loss, adversarial loss, and quantization loss. State-of-the-art VQGANs usually use Learned Perceptual Image Patch Similarity (LPIPS) to calculate the reconstruction loss. LPIPS is a perceptual loss function that measures the perceptual similarity between two images—not just pixel-wise differences, but how similar they look to a human observer. The adversarial loss encourages the generator to produce perceptually convincing images that fool the discriminator. The quantization loss encourages the encoder output  $z$  to stay close to the quantized vectors  $z_q$ , ensuring that the encoder consistently uses the codebook.

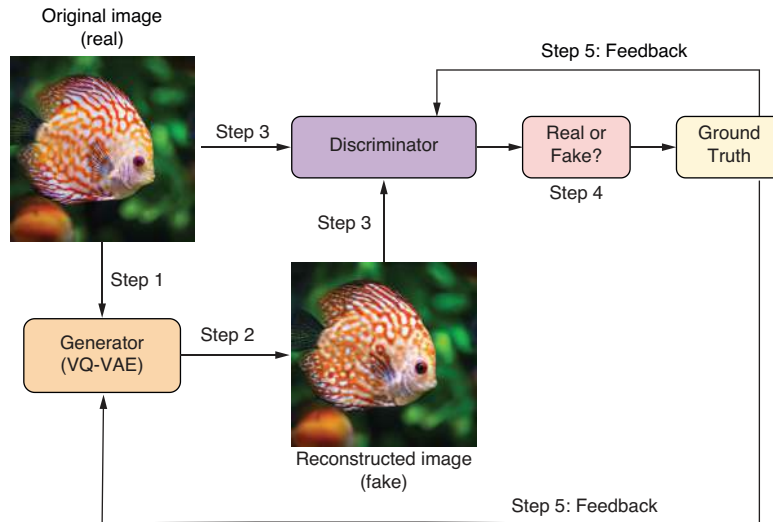
#### Perceptual loss

Perceptual loss measures the visual similarity between generated and real images based on human perception, rather than simple pixel-by-pixel differences. LPIPS is a common example: it uses deep neural network features from pretrained models (e.g., VGG or AlexNet) to compare images.

(continued)

By measuring differences in learned representations rather than raw pixels, LPIPS aligns more closely with human judgments of image quality, effectively capturing perceptual realism in generated images.

Figure 11.6 is a graph showing the architecture and training process of the VQGAN. The generator aims to produce images that are visually indistinguishable from real samples in the training set. Meanwhile, the discriminator attempts to determine whether a given image is “real” (from the training data) or “fake” (produced by the generator). During training, the generator reconstructs images from real inputs, producing synthetic (fake) samples, which are then mixed with real images and fed into the discriminator. The discriminator predicts whether each image is real or fake, and its predictions are compared against the ground truth labels. Based on the results, both the generator and discriminator update their parameters to improve their respective performance, making the generator better at fooling the discriminator and the discriminator better at detecting fakes.



**Figure 11.6**  
VQGAN  
architecture  
and training  
process in which  
VQGAN combines  
a VQ-VAE  
generator with a  
discriminator in a  
GAN framework

The training process of the VQGAN model involves multiple iterations, shown as steps in the preceding figure. In each iteration, we use the VQ-VAE generator to encode and quantize images in the training dataset:

- *Step 1*—The input image  $x$  is encoded into a continuous latent representation  $z$  and quantized into discrete codes  $z_q$ .

- *Step 2*—The quantized latent codes  $z_q$  are decoded into a reconstructed image  $\hat{x}$ .
- *Step 3*—Both the real image  $x$  and the generated image  $\hat{x}$  are evaluated by the discriminator.
- *Step 4*—The discriminator tries to classify each sample as either real or fake. It then compares the classification with the actual labels, the ground truth.
- *Step 5*—Both the discriminator and the VQ-VAE generator receive feedback from the classification and improve their capabilities: while the discriminator adapts its ability to identify fake samples, the generator learns to enhance its capacity to generate convincing samples to fool the discriminator.

As training advances, an equilibrium is reached when neither network can further improve. At this point, the generator becomes capable of producing data instances that are practically indistinguishable from real samples.

VQGAN elegantly integrates the expressive power of GANs with the discrete, structured latent space of VQ-VAEs. This combination allows VQGAN to generate high-resolution images with rich details and achieve stable training compared to traditional GANs. You'll learn to use VQGAN in the min-DALL-E project (a PyTorch replication of DALL-E) firsthand in the next chapter.

## 11.5 A pretrained VQGAN model

In this section, we'll work with a pretrained VQGAN model from the GitHub repository (<https://github.com/CompVis/taming-transformers>). You'll start by learning how to reconstruct a simple image of a cat using the model. From there, we'll dive deeper into the reconstruction process.

First, you'll learn the encoding process: the cat image is transformed into a set of latent vectors. We'll then map these latent vectors to discrete representations by finding the nearest feature vectors in a codebook, a process known as *quantization*. This allows us to represent the original image as a sequence of integers, where each integer corresponds to the index of a feature vector in the codebook. Finally, we'll reverse the process by decoding this sequence of integers back into a full image, demonstrating how VQGAN enables efficient compression and reconstruction of visual data.

### 11.5.1 Reconstructing images with the pretrained VQGAN

The modified GitHub repository you cloned earlier contains the VQGAN implementation. First, we'll download the pretrained weights and model configuration files for VQGAN. This will allow us to reconstruct images and explore how VQGAN encodes them into latent representations:

```
import requests, os

url1=('https://heibox.uni-heidelberg.de/f/'
      '140747ba53464f49b476/?dl=1')
url2=('https://heibox.uni-heidelberg.de/f/'
      '6ecf2af6c658432c8298/?dl=1')
```

```

file1="files/vqgan_imagenet_f16_1024.ckpt"
file2="files/vqgan_imagenet_f16_1024.yaml"

if not os.path.exists(file1):
    fb=requests.get(url1)
    with open(file1,"wb") as f:
        f.write(fb.content)
if not os.path.exists(file2):
    fb=requests.get(url2)
    with open(file2,"wb") as f:
        f.write(fb.content)

```

Defines URLs and filenames for the model weights and configuration

If the weights file is missing, download it from the specified URL.

If the configuration file is missing, download it from the specified URL.

The code block first checks whether the pretrained weights and model configuration files already exist in your local directory. If they are missing, the code will automatically download them from the provided URLs. Once these files are in place, you can import the VQGAN model directly from the cloned GitHub repository at <https://github.com/markhliu/taming-transformers>.

### Listing 11.2 Importing a pretrained VQGAN model

```

import torch
torch.set_grad_enabled(False)

device = "cuda" if torch.cuda.is_available() else "cpu"

from omegaconf import OmegaConf
from taming.models.vqgan import VQModel

config = OmegaConf.load(file2)
model = VQModel(**config.model.params).to(device)
sd = torch.load(file1)["state_dict"]
missing, unexpected = model.load_state_dict(sd,
                                             strict=False)

```

Disables gradient computation to speed up inference

Loads model configuration

Imports the VQGAN model from the GitHub repository

Loads pretrained weights to the VQGAN model

We import the VQGAN model from the GitHub repository and instantiate a model based on the model specifications in the file `vqgan_imagenet_f16_1024.yaml` that we downloaded earlier. We then load the pretrained weights into the model so that we can reconstruct images using the model later.

To understand how the VQGAN model reconstructs images step by step, we'll use two images as our examples. Go to the book's GitHub repository (<https://github.com/markhliu/txt2img>), and download the two images, `cat.png` and `fish.png`. Place the two images in the `/files/` folder on your computer.

### Listing 11.3 Reconstructing images using VQGAN

```

import PIL
import torchvision.transforms as T
import torchvision.transforms.functional as TF

```

```
def process_image(file_name):
    size=384
    img=PIL.Image.open(file_name)
    s = min(img.size)
    r = size / s
    s = (round(r * img.size[1]), round(r * img.size[0]))
    img = TF.resize(img, s, interpolation=PIL.Image.LANCZOS)
    img = TF.center_crop(img, output_size=2 * [size])
    return torch.unsqueeze(T.ToTensor()(img), 0)*2-1

image1=process_image("files/cat.png")
image2=process_image("files/fish.png")
print(f"the shape of the original image is {image1.shape}")
reconstruct1=model((image1).to(device))[0]
reconstruct2=model((image2).to(device))[0]
print(f"reconstructed image's shape is {reconstruct2.shape}")
```

Defines a process\_image() function to center crop images and resize them

Applies the process\_image() function on the cat and fish images

Reconstructs the cat and fish images using the pretrained VQGAN

The output is

```
the shape of the original image is torch.Size([1, 3, 384, 384])
reconstructed image's shape is torch.Size([1, 3, 384, 384])
```

We first define a process\_image() function to process images. The function loads an image from the local folder, center-crops the image, resizes it to  $3 \times 384 \times 384$ , and converts it into a PyTorch tensor. We apply the function to the cat and fish images and name the processed images image1 and image2, respectively. We feed the two processed images to the pretrained VQGAN to obtain two reconstructed images, reconstruct1 and reconstruct2.

### Exercise 11.1

Go to the book's GitHub repository (<https://github.com/markhliu/txt2img>), and download the two images, bird.png and bunny.png. Place the two images in the /files/ folder on your computer. Use the process\_image() function to process the two images, and name the processed images image3 and image4, respectively. Feed the two processed images to the pretrained VQGAN to obtain two reconstructed images, reconstruct3 and reconstruct4.

We plot the two original images and the two reconstructed ones in a  $1 \times 4$  grid, as shown here.

#### Listing 11.4 Comparing reconstructed images with originals

```
import matplotlib.pyplot as plt

imgs=[image1[0].cpu().permute(1,2,0)*0.5+0.5,
       reconstruct1[0].cpu().permute(1,2,0)*0.5+0.5,
       image2[0].cpu().permute(1,2,0)*0.5+0.5,
       reconstruct2[0].cpu().permute(1,2,0)*0.5+0.5]
```

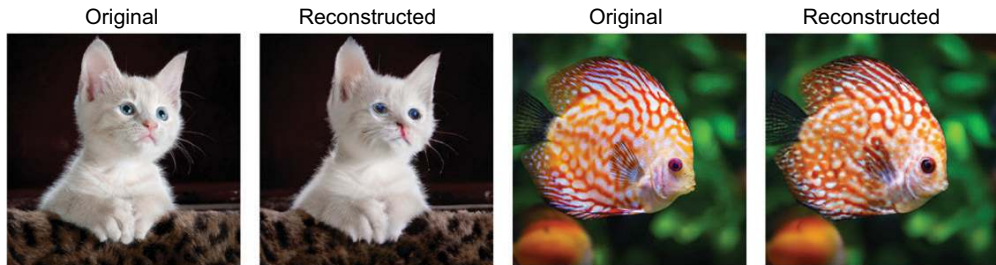
Places the two originals and the two reconstructed images in a list



```
captions=["original","reconstructed"]*2  ← Prepares four captions

plt.figure(figsize=(8,5),dpi=100)
for i in range(4):  ← Plots the four images in a 1 × 4 grid
    plt.subplot(1,4,i+1)
    plt.imshow(imgs[i].clamp(0,1))
    plt.title(captions[i],fontsize=15)
    plt.axis('off')
plt.tight_layout()
plt.show()
```

We place the two original images and the two reconstructed ones in a list `imgs`. We then plot the four images in a  $1 \times 4$  grid and put a caption of either Original or Reconstructed above each image. The output is shown in figure 11.7.



**Figure 11.7** Comparing reconstructed images with originals. A cat image and a fish image are processed and resized to  $3 \times 384 \times 384$ . We then feed the two processed images to the pretrained VQGAN model to obtain two reconstructed images.

The two reconstructed images in figure 11.7 are similar to the two originals, but you can easily see the difference. Some information is lost in the reconstruction process. Part of the reason is the quantization process: we use 1,024 discrete latent vectors in the codebook to approximate the actual latent vectors. Because the two sets of vectors aren't identical, some information is lost in the quantization process.

### Exercise 11.2

Place the two original images, `image3` and `image4`, and the two reconstructed ones, `reconstruct3` and `reconstruct4`, in a list `imgs`. Plot the four images in a  $1 \times 4$  grid, and add a caption of either Original or Reconstructed above each image.

Next, we'll dive deeper and examine how the model encodes the images and quantizes the latent vectors. We'll also investigate how the model translates the quantized latent vectors into a sequence of integers and converts the sequence back into an image.



### 11.5.2 Converting images into sequences of integers

We'll use the cat image you downloaded earlier as an example to show, step-by-step, how the VQGAN model reconstructs an image. The VQGAN model first encodes the input image into a latent representation. This is done using the model's encoder, followed by a linear projection layer (`quant_conv`). The process looks like this:

```
z=model.encoder(image1.to(device))
z=model.quant_conv(z)
print(f"the shape of the latent representation is {z.shape}")
```

The output is

```
the shape of the latent representation is torch.Size([1, 256, 24, 24])
```

This output tensor, `z`, is the latent representation of the image. The output shows that `z` has a shape of `[1, 256, 24, 24]`:

- The first dimension (1) indicates a single image in the batch.
- The second dimension (256) represents the number of feature channels.
- The last two dimensions ( $24 \times 24$ ) represent the spatial resolution of the latent image.

You may be wondering why the latent representation has a resolution of  $24 \times 24$ . This is because the encoder uses four strided convolutional layers, and each of these layers reduces the spatial resolution by a factor of 2. Therefore, the encoder has a total downsampling factor of  $2^4 = 16$ . The original image has a resolution of  $384 \times 384$ ; as a result, the encoded image has a resolution of  $24 \times 24$  because  $384 \div 16 = 24$ . Thus, the encoded image now has  $24 \times 24 = 576$  patches. Each patch is represented by a 256-dimensional feature vector, explaining the overall shape of `[1, 256, 24, 24]`.

The next step is to quantize the latent vectors. VQGAN maintains a codebook with 1,024 learnable vectors. During quantization, each feature vector in the latent space is replaced with the nearest vector from this codebook. This discretization allows the model to compress continuous latent vectors into discrete representations. The quantization process is performed as follows:

```
z_q, _, indices= model.quantize(z)
int_sequence = indices[2]
print(int_sequence)
```

Quantizes the latent vectors by  
replacing each one with its  
nearest neighbor in the codebook

Extracts the sequence of integers (indices  
of the selected codebook vectors)

Now the latent representation (`z`) are represented by the nearest vectors in the codebook, `z_q`. Further, the latent representation can be converted into a sequence of 576 integers, shown here (some numbers are omitted for brevity):

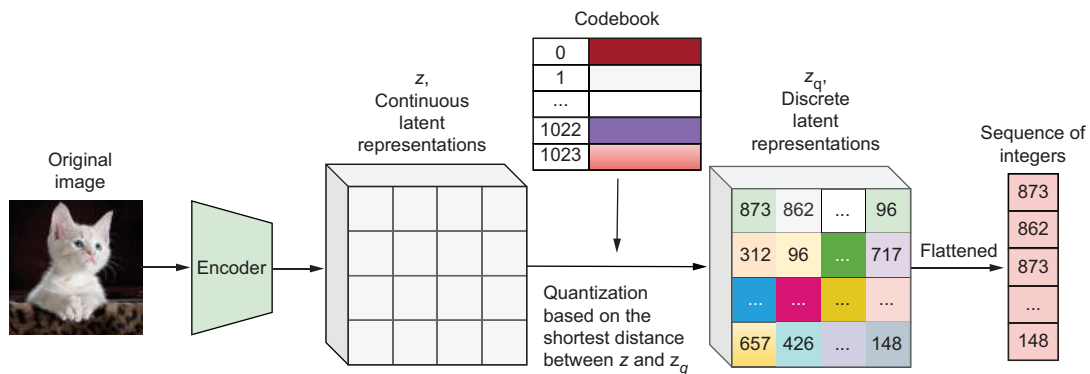
```

tensor(
  [ 873,  862,  873,  862,  862,   63,  110,  455,  590,  481,   29,  813,
    455,  813,  812,  813,  862,  862,   96,  862,  862,   63,   96,   96,
    ...,
    657,  426,  830,  717,  313,  941,  710,  172,  276,  551,  772,  717,
    339,  590,  328,  913,  623,  780,  990,  276,    4,  595,  894,  148],
  device='cuda:0')

```

The shape of the latent representation  $z$  is (256, 24, 24), meaning the image is represented by 576 patches in a  $24 \times 24$  grid with each patch represented by a 256-value feature vector. We then match each of the 576 vectors with the nearest vector in the codebook. Therefore, each of the 576 vectors can be assigned an index, where the index is the corresponding discrete vector in the codebook. The result is a sequence of 576 integers, each representing a specific vector in the codebook that approximates the original latent vector.

Figure 11.8 illustrates how the VQGAN model converts an image into a sequence of integers.



**Figure 11.8** Converting an image into a sequence of integers. The VQGAN model first encodes an image into a continuous latent representation,  $z$ , in the latent space, which consists of 576 feature vectors. Each of these vectors is matched with one of the 1,024 discrete feature vectors in the codebook. Therefore, the image can be represented by a sequence of integers, with each integer representing the index of the feature vector in the codebook.

As shown in figure 11.8, the first two integers in the sequence are 873 and 862. This indicates the following:

- The first feature vector in the latent grid is closest to vector 873 in the codebook.
- The second feature vector is nearest to the codebook vector 862.

With this sequence of integers and access to the codebook, you can reconstruct the quantized latent representations, denoted as  $z_q$ . Each integer simply serves as a

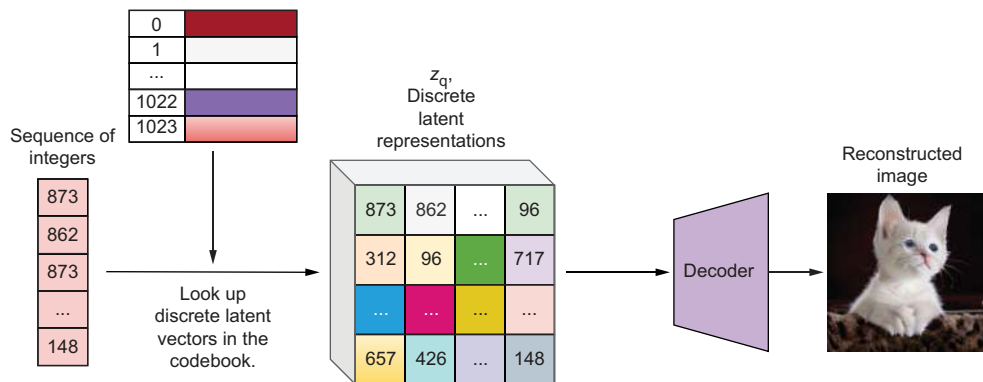
pointer to a specific vector in the codebook, allowing you to rebuild the entire quantized latent grid.

### Exercise 11.3

Find out the quantized latent representation for the fish image, and name it `z2_q`. Find out the sequence of integers that represent the fish image, and call it `int_sequence2`. Print out the 576 values in `int_sequence2`.

Importantly, the VQGAN decoder is capable of performing the reverse operation: it takes this sequence of integers (representing codebook indices) and reconstructs the corresponding image. This ability to map discrete sequences back into visual data is foundational for models such as DALL-E, which generate images from sequences of integers derived from text prompts. We'll explore this concept further in the next chapter.

Figure 11.9 illustrates how VQGAN reconstructs an image from a sequence of integers, showcasing the power of this discrete latent representation. Given a sequence of 576 integers, we can reconstruct the corresponding image using the VQGAN codebook and decoder. The process begins by mapping each integer in the sequence to its corresponding feature vector in the codebook. For instance, if the first two integers in the sequence are 873 and 862, they point to vectors 873 and 862 in the codebook, respectively. By retrieving all 576 feature vectors in this way, we assemble a complete quantized latent representation, which we'll call `z_q2`.



**Figure 11.9** Reconstructing an image from a sequence of integers. Each integer in the sequence corresponds to the index of a feature vector in the VQGAN codebook, allowing the model to retrieve the associated feature vectors. These 576 vectors collectively form the quantized latent representation,  $z_q$ . The VQGAN decoder then uses  $z_q$  to reconstruct the original image.

Once `z_q2` is formed, we can pass it through the VQGAN decoder to reconstruct the image. The process is implemented as follows:

```

z_q2 = model.quantize.embedding(int_sequence)
z_q2 = z_q2.permute(1,0).view(z_q.shape)
z_post2 = model.post_quant_conv(z_q2)
z_recon2 = model.decoder(z_post2)

```

Builds a quantized latent representation, **z\_q2**, based on the sequence of integers

Converts the latent representation into an image using the VQGAN decoder

We use the VQGAN decoder to convert the quantized latent representation **z\_q2** into a reconstructed image, **z\_recon2**.

#### Exercise 11.4

Map each integer in **int\_sequence2** to its corresponding feature vector in the codebook to assemble a complete quantized latent representation **z2\_q2**. Pass **z2\_q2** through the VQGAN decoder to reconstruct the fish image, and name the reconstructed image **z2\_recon2**.

We can compare **z\_recon2** with the one reconstructed based on **z\_q**:

```

z_post = model.post_quant_conv(z_q)
z_recon = model.decoder(z_post)

```

Reconstructs the cat image using the quantized latent representation, **z\_q**

```

imgs=[z_recon[0].cpu().permute(1,2,0)*0.5+0.5,
      z_recon2[0].cpu().permute(1,2,0)*0.5+0.5]

captions=["reconstructed based on\nquantized vectors",
          "reconstructed based on \na sequence of indices"]

plt.figure(figsize=(8,5),dpi=100)
for i in range(2):
    plt.subplot(1,2,i+1)
    plt.imshow(imgs[i].clamp(0,1))
    plt.title(captions[i],fontsize=20)
    plt.axis('off')
plt.tight_layout()
plt.show()

```

Compares the reconstructed image based on **z\_q** to the image based on the sequence of indices

The image **z\_recon** is reconstructed based on the quantized representation **z\_q**. We compare it with the reconstructed image based on the sequence of integers. Both images are shown in figure 11.10.

#### Exercise 11.5

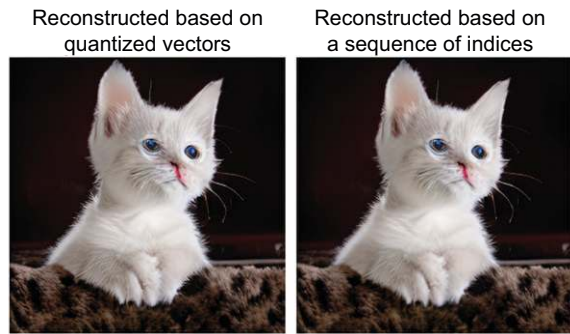
First, reconstruct the fish image based on the quantized latent representation you first generated, **z2\_q**, in exercise 11.3, and name the reconstructed fish image **z2\_recon**. Plot **z2\_recon** and **z2\_recon2** side by side in a  $1 \times 2$  grid to compare them.

Figure 11.10 displays two identical reconstructed images, demonstrating VQGAN's capability to regenerate an image solely from a sequence of integers. This ability to convert discrete token sequences back into high-quality images is a foundational feature in state-of-the-art text-to-image models such as DALL-E.

In the next chapter, you'll explore the min-DALL-E project, an open source PyTorch implementation that replicates key aspects of DALL-E. The VQGAN model you studied in this chapter serves as a crucial component within min-DALL-E, enabling the translation of text-driven sequences into coherent visual outputs.

## Summary

- Traditional variational autoencoders (VAEs) encode images into probabilistic latent distributions and reconstruct them by sampling from these distributions. This process often leads to blurry outputs due to the averaging effect of sampling.
- Vector-quantized VAEs (VQ-VAEs) improve upon traditional VAEs by replacing continuous latent distributions with discrete codes from a learned codebook. This leads to sharper reconstructions and more interpretable, efficient latent representations.
- Generative adversarial networks (GANs) consist of a generator and a discriminator trained adversarially: the generator learns to create realistic images, while the discriminator learns to distinguish real from fake.
- VQGAN integrates the discrete latent structure of VQ-VAEs with the adversarial training of GANs. This hybrid model benefits from both stability and semantic structure, enabling the generation of high-resolution, perceptually realistic images.
- The VQGAN loss function blends three key components: adversarial loss for realism, perceptual loss (e.g., LPIPS) for visual fidelity, and quantization loss for effective discretization of the latent space.
- VQGAN encodes images as sequences of discrete tokens, making it a foundational component in models such as DALL-E, where images are generated from symbolic sequences using transformer architectures.



**Figure 11.10** Comparing the image reconstructed based on quantized vectors with the one reconstructed based on the sequence of integers. The left image is reconstructed based on the quantized latent representation generated by the VQGAN encoder,  $z_q$ . The right image is reconstructed based on the sequence of 576 integers. We look up feature vectors in the codebook based on the integers in the sequence. These feature vectors form a quantized latent representation,  $z_{q2}$ , which is then fed to the VQGAN decoder to reconstruct the image.

# 12

## *A minimal implementation of DALL-E*

---

### ***This chapter covers***

- How DALL-E is trained to generate images from text descriptions
- How a pretrained BART encoder transforms a text prompt into dense embeddings
- How a BART decoder uses those embeddings to predict image tokens
- How a VQGAN decoder converts image tokens into a high-resolution image

OpenAI's DALL-E is one of the earliest and most influential text-to-image generators. It's a large-scale transformer that can generate high-resolution images from natural language prompts. DALL-E stands at the intersection of two technologies in modern AI: powerful autoregressive language models and vector-quantized representations of visual information. Yet, for many learners, practitioners, and researchers, the full DALL-E model remains out of reach due to its proprietary nature.

To foster deeper understanding and democratize access, the open-source community has stepped up, replicating DALL-E’s core ideas with public tools and datasets. Notably, the DALL-E mini project and its streamlined PyTorch reimplementation, min-DALL-E, have made it possible for anyone to explore the mechanics of transformer-based text-to-image generation and even build their own models from scratch.

Here, you’ll learn to build a transformer-based text-to-image generator from scratch, not treating it as a black box, but instead working through every stage of the process. You’ll learn how a bidirectional and autoregressive transformer (BART) encoder transforms a text prompt into rich vector embeddings, how the BART decoder translates those embeddings into a sequence of image tokens, and finally, how a vector quantized generative adversarial network (VQGAN) decoder reconstructs those tokens into a complete high-resolution image.

To make the process tangible, we’ll use a simple, vivid prompt, “panda with top hat reading a book,” and go through each stage of the pipeline. Along the way, you’ll see not only the final result but also the intermediate outputs: what the model produces after generating just 32, 64, or 128 of the eventual 256 image tokens. This incremental visualization will give you an intuitive grasp of how the image gradually takes shape as the model “paints” patch by patch.

### Note

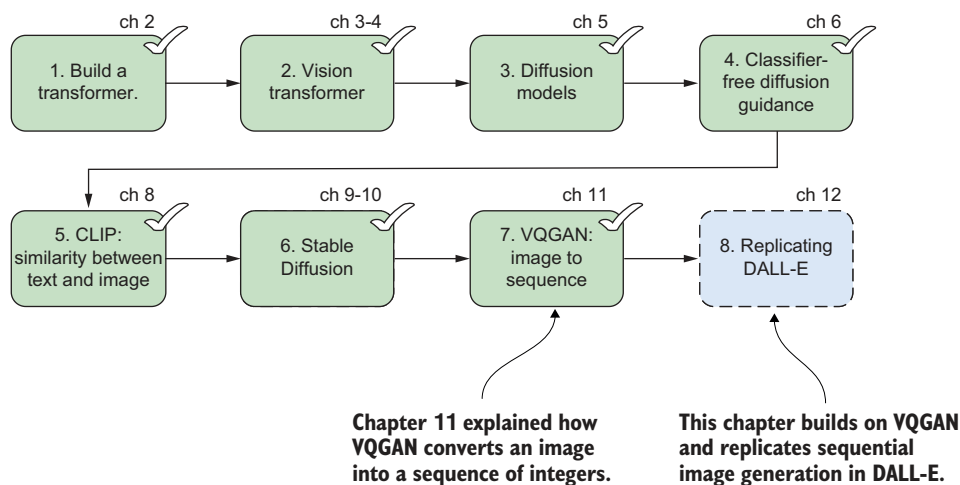
You may have noticed that the prompt “panda with top hat reading a book” is grammatically informal. Image-generation models, such as DALL-E or Stable Diffusion, prioritize the semantics and content of prompts rather than strict grammatical correctness. In practice, shorter and less grammatically formal prompts can often lead to clearer and more visually appealing results because the model easily identifies the main subjects and actions.

Figure 12.1 illustrates how this chapter fits into your journey so far. Throughout the book, you’ve built up the knowledge and tools needed for text-to-image generation from scratch. Now, in step 8, you’ll build on the VQGAN model you learned in the previous chapter to replicate a transformer-based generator similar to DALL-E.

By the end of the chapter, you’ll not only understand the architecture and training procedure of DALL-E-like models, but you’ll also gain hands-on experience generating images from text prompts using the min-DALL-E implementation. Along the way, you’ll deepen your understanding of key generative modeling concepts such as vector quantization, autoregressive decoding, temperature scaling, and top-K sampling.

## 12.1 How min-DALL-E works

To understand how min-DALL-E generates images from text, it’s important to have a grasp of the architecture and training process that inspired it: OpenAI’s DALL-E. Although the official DALL-E model remains closed-source, the open-source DALL-E mini project provides valuable insight by approximating its design. We’ll



**Figure 12.1** Eight steps for building a text-to-image generator from scratch. This chapter focuses on the last step: replicating a transformer-based text-to-image generator such as DALL-E.

reverse-engineer the high-level ideas behind DALL-E and explain how min-DALL-E replicates the text-to-image generation process using a transformer-based architecture and a VQGAN decoder.

### 12.1.1 *Training min-DALL-E*

DALL-E is a generative model that takes a natural language prompt, such as “horses walk on the beach,” and outputs a coherent image that represents the described scene. To accomplish this, it must learn a joint distribution between text and images. That means understanding both how text descriptions relate to images and how to generate visual content that reflects those descriptions.

This task can be viewed as a type of sequence-to-sequence generation problem. Instead of translating text from German to English as in traditional sequence-to-sequence generation (implemented in chapter 2), DALL-E “translates” text into images by converting a sequence of words into a sequence of image tokens. It does this using the following pipeline:

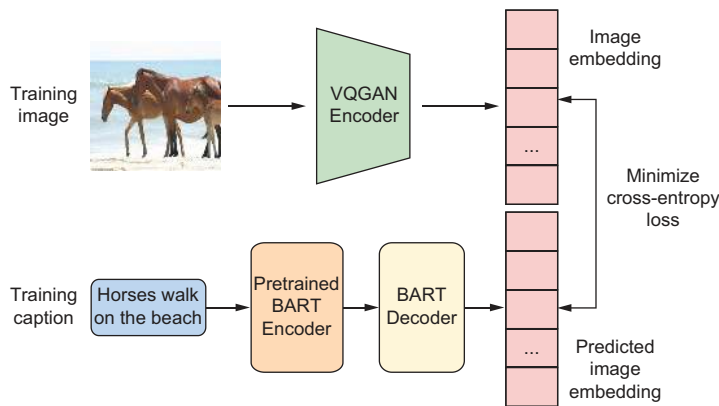
- *Text encoding*—The input prompt is tokenized and passed through a pretrained BART encoder. BART is used here because it combines the bidirectional understanding of bidirectional encoder representations from transformers (BERT; useful for capturing full context in the prompt) with the autoregressive decoding ability of generative pretrained transformers (GPTs). This makes it especially suitable for Seq2Seq tasks such as text-to-image generation.
- *Image embedding prediction*—A transformer-based BART decoder, trained from scratch, autoregressively predicts a sequence of image tokens conditioned on the encoded text.



- *Image reconstruction*—The sequence of image tokens is passed to a VQGAN decoder, which transforms them into image patches and stitches them into a complete image.

The elegance of this system lies in its ability to handle both modalities, text and image, within a unified transformer architecture, with training supervised by cross-entropy loss on paired text-image data. Because the original DALL-E model is proprietary, the open source DALL-E mini project was designed to mimic DALL-E using publicly available datasets and tools. min-DALL-E is the PyTorch implementation of this project.

The training steps in DALL-E mini are illustrated in figure 12.2, which depicts how the model learns to map textual descriptions to their corresponding images. The model is trained on a large dataset of image–caption pairs to replicate the functionality of OpenAI’s proprietary DALL-E. During training, each image is encoded into a sequence of discrete tokens using a pretrained VQGAN encoder. At the same time, the accompanying caption is tokenized and passed through a pretrained BART encoder to produce a sequence of text embeddings. These embeddings, along with the VQGAN-encoded image tokens, are used to train a BART decoder from scratch. The decoder learns to autoregressively predict the image tokens, and the training objective is to minimize the cross-entropy loss between the predicted and actual image token sequences.



**Figure 12.2** Training pipeline of the DALL-E mini model

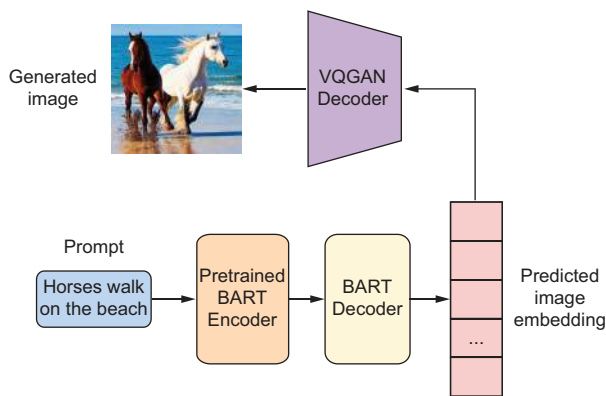
A dataset of image–caption pairs (about 15 million examples) is used to train the DALL-E mini model. Each image is first passed through a pretrained VQGAN encoder, which converts the image into a sequence of 256 discrete image tokens. At the same time, the associated caption is tokenized and encoded into a sequence of dense vectors using a pretrained BART encoder. The BART decoder is then trained to predict the image tokens autoregressively (one at a time, sequentially) conditioned on the output of the BART encoder. The training objective is to minimize the cross-entropy loss

between the predicted image tokens and the ground-truth tokens produced by the VQGAN encoder.

The BART encoder is frozen during training, while the BART decoder learns to bridge the gap between textual and visual representations. This design simplifies training and takes advantage of powerful pretrained language models, enabling the decoder to focus on learning how to “speak image.”

### 12.1.2 From prompt to pixels: Image generation at inference time

The PyTorch implementation of DALL-E mini by Brett Kuprel, min-DALL-E, doesn’t train a model from scratch. Instead, it reuses the pretrained weights from DALL-E mini, which was trained using Flax/JAX (JAX is a Python library developed by Google for high-performance numerical computing and machine learning research, and Flax is a deep learning library built on top of JAX). The min-DALL-E model converts those weights to work with a PyTorch implementation of the same architecture. Figure 12.3 depicts how min-DALL-E generates an image from a text prompt.



**Figure 12.3** The image generation process in min-DALL-E to generate an image from a text prompt

At inference time, we first tokenize and index the text prompt. These tokens are passed through the BART encoder to obtain text embeddings. The trained BART decoder then autoregressively generates image tokens. It starts with a special beginning-of-sequence (BOS) token and predicts the first image token. That token is added to the sequence, and the process repeats until all 256 image tokens are generated. Finally, these tokens are decoded into a high-resolution image using the VQGAN decoder. This step-by-step generation process mimics the way language models produce text, except that the output sequence represents patches of an image rather than words or characters.

One challenge in text-to-image generation is dealing with the high dimensionality of images. Images are large arrays of pixel values, and directly generating them using a transformer is computationally prohibitive. To address this, DALL-E mini and

min-DALL-E use VQGAN to compress images into discrete tokens from a learned codebook. Each token represents a small image patch. The decoder in VQGAN can reconstruct the full image from a sequence of these tokens.

Instead of predicting raw pixels directly, DALL-E mini (or min-DALL-E) is trained to predict sequences of discrete image tokens. Raw pixels are continuous values representing color intensities for every point in an image, which makes the prediction space extremely large and complex. In contrast, discrete image tokens come from a finite vocabulary produced by a pretrained image tokenizer, similar to how words are represented as tokens in language models. This transformation makes the generation process more efficient, scalable, and conceptually aligned with language modeling.

In summary, DALL-E mini is trained to map text descriptions to image tokens and then reconstruct those tokens into high-resolution images. It does this by combining the strengths of three powerful components:

- A pretrained BART encoder to convert text into embeddings
- A decoder-only transformer trained to predict image tokens autoregressively
- A pretrained VQGAN decoder to convert image tokens into visual content

Although it's a simplified version of OpenAI's DALL-E, min-DALL-E is remarkably capable and serves as a practical and educational foundation for understanding text-to-image models. In the next section, we'll begin building this system step-by-step, starting with tokenizing and encoding the text prompt.

## 12.2 Tokenizing and encoding the text prompt

Before we generate an image using a prompt, we must first convert the input text prompt into a form that the model understands. This process involves the following steps:

- 1 Download the tokenizer vocabulary files.
- 2 Initialize the tokenizer.
- 3 Tokenize the text prompt.
- 4 Pad and format the tokenized input.
- 5 Encode the tokens into text embeddings using a pretrained BART encoder.

Let's walk through each of these steps using the prompt "panda with top hat reading a book" as our running example. This simple, vivid example produces fun visual output. The Python programs in this chapter are adapted from the excellent GitHub repository by Brett Kuprel (<https://github.com/kuprel/min-dalle>).

### 12.2.1 Tokenizing the text prompt

First, you should clone the GitHub repository by Brett Kuprel to your computer by running the following line of code in a code cell in Jupyter Notebook:

```
!git clone https://github.com/kuprel/min-dalle
```

You'll see a folder `/min-dalle/` on your computer, which contains all files and modules in the min-DALL-E project.

**NOTE** The Python programs for this chapter can be accessed from the GitHub repository (<https://github.com/markhliu/txt2img>) and the Google Colab. Because the model in this chapter is quite large and memory-intensive, I used two separate Colab notebooks (<https://mng.bz/7Q6Q> and <https://mng.bz/mZD8>). This allows you to run everything efficiently without requiring a paid GPU plan.

Next, we create a local folder to store the pretrained models:

```
import os

folder="files/DALLE"
if not os.path.exists(folder):
    os.makedirs(folder)
```

← If the folder `/files/DALLE/` isn't on your computer, create the folder.

You should see a new directory `/files/DALLE/` created on your computer after running the preceding code cell. We'll place pretrained models in this folder later.

The tokenizer used by min-DALL-E is based on byte pair encoding (BPE), and it relies on two files: `vocab.json`, which maps tokens to IDs, and `merges.txt`, which defines how subword units are merged during tokenization. These files are hosted on Hugging Face and must be downloaded to initialize the tokenizer.

#### Listing 12.1 Downloading tokenizer files

```
import requests

url='https://huggingface.co/kuprel/min-dalle/resolve/main/'
vocab_path=folder+"/vocab.json"
merges_path=folder+"/merges.txt"

if not (os.path.exists(vocab_path)
        and os.path.exists(merges_path)):
    print("downloading tokenizer parameters")

    vocab = requests.get(url + 'vocab.json')
    merges = requests.get(url + 'merges.txt')
    with open(vocab_path, 'wb') as f:
        f.write(vocab.content)
    with open(merges_path, 'wb') as f:
        f.write(merges.content)
else:
    print("tokenizer parameters have already been downloaded")
```

← If either `vocab.json` or `merges.txt` isn't on your computer, starts downloading

← Downloads the two files and saves them on your computer

← Skips if the two files have already been downloaded

This code block checks whether the tokenizer files already exist on your computer. If not, it downloads them from the official min-DALL-E Hugging Face repository and saves them locally for later use. Go to the `/files/DALLE/` folder, and you'll see the two tokenizer files, `vocab.json` and `merges.txt`.

With the tokenizer files in place, we now initialize the tokenizer. The min-DALL-E implementation provides a custom tokenizer in the `min_dalle.text_tokenizer` module.

### Listing 12.2 Initializing the tokenizer

```
import sys
sys.path.append("min-dalle")

import json
from min_dalle.text_tokenizer import TextTokenizer

print("intializing TextTokenizer")
with open(vocab_path, 'r', encoding='utf8') as f:
    vocab = json.load(f)
with open(merges_path, 'r', encoding='utf8') as f:
    merges = f.read().split("\n")[1:-1]
tokenizer = TextTokenizer(vocab, merges)
```

← Adds the /min-dalle/ folder to Python's search path

← Imports the TextTokenizer() class from the min-DALL-E GitHub repository

← Loads the tokenizer files

← Initializes the tokenizer

To make sure Python can access and import modules from the min-DALL-E GitHub repository, we add the `/min-dalle/` folder to Python's search path using the `sys` library. We then import the `TextTokenizer()` class from the repository and load up the two tokenizer files. We initialize the tokenizer by using the two files as input in the `TextTokenizer()` class.

This tokenizer converts raw text into a sequence of token IDs using BPE. It also handles special tokens such as `<s>` (start of sequence), `</s>` (end of sequence), and `<pad>` (padding), which are required for the model to understand context and batch input correctly. Let's inspect the first few entries in the vocabulary to get a sense of how tokens are indexed:

```
print({k:v for k,v in vocab.items() if v<10})
```

The output is

```
{'<s>': 0, '<pad>': 1, '</s>': 2, '<unk>': 3, '<mask>': 4, '': 5,
 '$': 6, '%': 7, '&': 8, '"': 9}
```

As you can see, the start of the sentence token is mapped to index 0. The padding token is mapped to index 1. The end of the sentence token is mapped to index 2. These IDs are used internally by the model and must match those in the pretrained weights for inference to work properly. Now we'll tokenize our prompt.

### Listing 12.3 Tokenizing the text prompt

```
text = "panda with top hat reading a book"
token_ids = tokenizer.tokenize(text)
print("Token IDs:", token_ids)
```

← Tokenizes text to get token IDs

```
idx2token = {v: k for k, v in vocab.items()}
raw_tokens = [idx2token.get(i, "<unk>") for i in token_ids]
print("Raw Tokens:", raw_tokens)
```

← Reverses the vocab dictionary to map IDs to tokens

The `tokenizer.tokenize()` function in `min-DALL-E` returns token IDs, not raw token strings. However, we've recovered the raw subword tokens by reversing the vocab dictionary. The output is

```
Token IDs: [0, 8418, 208, 479, 2583, 4132, 58, 407, 2]
Raw Tokens: ['<s>', 'Ġpanda', 'Ġwith', 'Ġtop', 'Ġhat',
'Ġreading', 'Ġa', 'Ġbook', '</s>']
```

The token IDs start with 0, corresponding to the token '<s>', signaling the start of the sentence. The last token ID is 2, corresponding to the token '</s>', indicating the end of the sequence. The prompt itself is divided into seven raw tokens, ['Ġpanda', 'Ġwith', 'Ġtop', 'Ġhat', 'Ġreading', 'Ġa', 'Ġbook'], where Ġ indicates a word boundary (space). This is a convention in BPE tokenizers used by models such as GPTs and BART.

### Exercise 12.1

Convert the prompt “an underwater fantasy city” into token IDs and then raw tokens. Name the list of token IDs `token_ids2`.

Transformer models require fixed-length sequences for batching and efficient computation. We'll create a padded input of length 64 and prepare two sequences: one for the actual text and another unconditional one for classifier-free guidance (CFG; as discussed in chapter 6).

### Listing 12.4 Formatting and padding the tokenized input

```
import numpy, torch

device="cuda" if torch.cuda.is_available() else "cpu"

text_tokens = numpy.ones((2, 64), dtype=numpy.int32)
text_tokens[0, :2] = [token_ids[0], token_ids[-1]]
text_tokens[1, :len(token_ids)] = token_ids
text_tokens = torch.tensor(text_tokens,
                           dtype=torch.long, device=device)
print(text_tokens)
```

← Initializes a NumPy array filled with the <pad> token (index 1)

← Creates the unconditional prompt with just two tokens <s> and </s>

← Converts the input to a PyTorch long tensor

← Creates the padded conditional prompt

This creates a  $2 \times 64$  tensor: one row for the unconditional prompt and one for the real prompt, both padded to a uniform length. The output is

```

tensor([
  [ 0,  2,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
    1,  1,  1,  1],
  [ 0, 8418, 208, 479, 2583, 4132, 58, 407, 2, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1]]), device='cuda:0')

```

The first row in the output represents the unconditional prompt, which is essentially an empty string. The second row represents the conditional prompt. Both are padded to a length of 64 elements.

### Exercise 12.2

Use the list of token IDs `token_ids2` that you created in exercise 12.1 to create a PyTorch tensor `text_tokens2`. Make sure that `text_tokens2` has a shape of (2, 64), representing the padded token IDs for the unconditional prompt and the text prompt “an underwater fantasy city,” respectively. Print out the values of `text_tokens2`.

#### 12.2.2 Encoding the text prompt

The next step is to use the pretrained BART encoder to transform our tokenized input into high-dimensional vector embeddings. To that end, we download the pretrained weights for the BART encoder from Hugging Face like this:

```

encoder_path=folder+"/encoder.pt"

if not os.path.exists(encoder_path):
    print("downloading encoder parameters")
    ws = requests.get(url + 'encoder.pt')
    with open(encoder_path, 'wb') as f:
        f.write(ws.content)
else:
    print("encoder parameters have already been downloaded")

```

← If the weights aren't available locally, downloads them from Hugging Face

← Otherwise, skips downloading

This code block ensures that the pretrained BART encoder weights (used in the min-DALL-E model) are available locally. If they're not, it downloads them from Hugging Face. These weights are essential for converting text prompts into dense vector embeddings.

To create a BART encoder, we instantiate the `DalleBartEncoder()` class in the local module file `dalle_bart_encoder.py` in the `/min-dalle/min_dalle/models/` folder. The following listing shows how.

**Listing 12.5** Initializing the pretrained BART encoder

```

from min_dalle.models import DalleBartEncoder

torch.manual_seed(42)
dtype = "float16"
print("initializing DalleBartEncoder")
encoder = DalleBartEncoder(
    attention_head_count = 32,
    embed_count = 2048,
    glu_embed_count = 4096,
    text_token_count = 64,
    text_vocab_count = 50272,
    layer_count = 24,
    device=device
).to(getattr(torch, dtype)).eval()
params = torch.load(encoder_path)
encoder.load_state_dict(params, strict=False)
del params
encoder = encoder.to(device)

```

← Fixes the random state so results are reproducible

← Instantiates the `DalleBartEncoder()` class to create the BART encoder

← Text embeddings have a dimension of 2,048.

← Text prompts have 64 tokens.

← Loads the pretrained weights

This code block sets up and loads the pretrained BART encoder in min-DALL-E. We set a fixed random seed for reproducibility and use the data type `float16` for faster inference. Now we can encode the unconditional prompt and the text prompt:

```

with torch.amp.autocast("cuda", dtype=getattr(torch, dtype)):
    encoder_state = encoder.forward(text_tokens)
torch.cuda.empty_cache()
print("Shape of the text embeddings:", encoder_state.shape)

```

The output is

```
Shape of the text embeddings: torch.Size([2, 64, 2048])
```

The output tensor contains the encoded embeddings for both the unconditional and conditional prompts. Each of the 64 tokens is represented as a 2,048-dimensional vector. These embeddings will be used in the next step to condition the image token generation.

**Exercise 12.3**

Use the BART text encoder to generate text embeddings for the PyTorch tensor `text_tokens2` that you created in exercise 12.2. Name the text embeddings `encoder_state2`, and print out its shape.

**12.3 Iterative prediction of image tokens**

Now that we've obtained text embeddings using the BART encoder, it's time to generate image tokens, one of the most critical steps in the min-DALL-E pipeline. Next, we'll show you how to do the following:



- Load the pretrained BART decoder
- Configure sampling parameters
- Use the decoder to predict a sequence of image tokens autoregressively
- Capture intermediate token outputs for visualization later

This process effectively converts text into a “visual language” that the VQGAN decoder will later reconstruct into a high-resolution image.

### 12.3.1 Loading the pretrained BART decoder

Just like the encoder, the decoder in min-DALL-E is based on a transformer architecture. Its role is to generate a sequence of 256 image tokens, one at a time, conditioned on the text embeddings we computed in the previous section. We first download the pretrained weights if they’re not already saved locally:

```
decoder_path=folder+"/decoder.pt"

if not os.path.exists(decoder_path):
    print("downloading decoder parameters")
    weights = requests.get(url + 'decoder.pt')
    with open(decoder_path, 'wb') as f:
        f.write(weights.content)
else:
    print("decoder parameters have already been downloaded")
```

← Downloads the pretrained BART decoder weights if they aren't saved locally

Next, we create a BART decoder by instantiating the `DalleBartDecoder()` class from the local module.

#### Listing 12.6 Initializing the BART decoder

```
from min_dalle.models import DalleBartDecoder

print(„initializing DalleBartDecoder“)
decoder = DalleBartDecoder(
    image_vocab_count = 16415,
    attention_head_count = 32,
    embed_count = 2048,
    glu_embed_count = 4096,
    layer_count = 24,
    device=device
).to(getattr(torch, dtype)).eval()
params = torch.load(decoder_path)
decoder.load_state_dict(params, strict=False)
del params
decoder = decoder.to(device)
```

← Creates a BART decoder

← Loads the trained weights

We create the BART decoder and load the trained weights. The decoder is trained to predict image tokens autoregressively, that is, one token at a time, using all previously generated tokens and the encoded text as input.

### 12.3.2 Predicting image tokens using the BART decoder

We're now ready to generate image tokens. The image is composed of 256 discrete tokens, arranged in a  $16 \times 16$  grid, representing patches of the final image. These tokens are generated sequentially using the decoder. Before sampling, we need to set up several components:

```
attention_mask=text_tokens.not_equal(1)[:, None, None, :]
attention_state=torch.zeros(size=(24,4,256,2048),
                               device=device)
image_tokens=torch.full((1, 256 + 1),
                        2 ** 14 - 1, dtype=torch.long, device=device)
```

Creates an attention mask

Creates an attention\_state cache

Creates a sequence of 257 image tokens, filled with a special placeholder token

We create a tensor `attention_mask`, which is used to ensure that the model ignores padding tokens (index 1) when computing attention scores. It's shaped to match the decoder's expected input for causal self-attention. We then create the `attention_state` tensor to cache the decoder's internal attention activations across 24 layers and four attention heads. Finally, the tensor `image_tokens` holds the generated sequence of tokens. It's initialized with a special placeholder token (index 16383) and will be filled with valid image tokens as generation proceeds.

#### Exercise 12.4

Create three tensors, `attention_mask2`, `attention_state2`, and `image_tokens2`, to represent the attention mask, attention state cache, and image tokens for the text prompt "an underwater fantasy city."

To control the creativity and quality of the output, min-DALL-E uses a few key sampling strategies:

```
token_indices=torch.arange(256, device=device)
temperature=0.5
top_k=128
supercondition_factor=4
settings = torch.tensor(
    [temperature, top_k, supercondition_factor],
    dtype=torch.float32, device=device)
```

Sets the temperature parameter to 0.5

Sets the top-K sampling parameter to 128

Sets the supercondition factor to 4

The temperature parameter controls the randomness of the predictions of the trained model. A high temperature (greater than 1, e.g., 1.5) makes the generated output more creative, while a low temperature (less than 1, e.g., 0.6) makes the text more confident and predictable. Top-K sampling is a method where you select the

next token from the top K most probable tokens, rather than selecting from the entire vocabulary. A small value of K leads to the selection of highly likely tokens in each step, which, in turn, makes the generated text less creative and more coherent. The supercondition factor balances the influence of the unconditional and conditional prompts during CFG. Higher values increase the model's adherence to the input text.

Now comes the core loop. We generate tokens one at a time using the decoder, feeding in the current token and updating the internal state.

#### Listing 12.7 Predicting image tokens iteratively

```
from copy import deepcopy

imgtokens=[]
for i in range(256):
    torch.cuda.empty_cache()
    with torch.amp.autocast("cuda", dtype=getattr(torch, dtype)):
        image_tokens[:, i + 1], \
            attention_state = decoder.sample_tokens(
                settings=settings,
                attention_mask=attention_mask,
                encoder_state=encoder_state,
                attention_state=attention_state,
                prev_tokens=image_tokens[:, [i]],
                token_index=token_indices[[i]])
        imgtokens.append(deepcopy(image_tokens))
```

Creates a list `imgtokens` to store intermediate steps

Iterates 256 steps

Predicts the next token using previous tokens and the text embeddings

We iteratively predict the 256 image tokens. In the first iteration, we predict the image token for the patch in the top-left corner of the image. In the second iteration, we predict the image token for the patch in the top row, second column in the image grid, and so on. We've created a list `imgtokens` to contain all intermediate steps so that later we can visualize how the image is created step-by-step.

By autoregressively generating each token, min-DALL-E constructs an image in a step-by-step manner, similar to how GPT generates text. This gives us fine-grained control over the generation process and allows us to inspect intermediate outputs. For instance, you can visualize what the image looks like after 32, 64, or 128 tokens to understand how the image evolves as more information is added.

These image tokens are still abstract. They represent positions in a learned VQGAN codebook. Next, we'll decode them into visual patches and stitch them into a full image using the pretrained VQGAN detokenizer.

#### Exercise 12.5

Predict image tokens iteratively for the prompt "an underwater fantasy city." Save all intermediate image tokens in a list `imgtokens2` as we did earlier in listing 12.7.

## 12.4 Converting image tokens to high-resolution images

By now, we've taken a text prompt, encoded it using a BART encoder, and used a decoder to generate a sequence of 256 image tokens, each representing a patch in a  $16 \times 16$  image grid. However, these tokens aren't directly viewable. They are indices pointing to a codebook learned by a VQGAN. To visualize the final image, we must convert these tokens into actual pixels, a process known as detokenization.

Next, we'll use a pretrained VQGAN decoder, also known as the detokenizer, to convert the sequence of tokens into a high-resolution color image. We'll first generate the complete image, then visualize how the image evolves step-by-step, and finally create an animation that shows how a picture is literally built one token at a time.

### 12.4.1 Loading the pretrained VQGAN detokenizer

The VQGAN detokenizer is a convolutional neural network trained to map discrete token indices back to their corresponding image patches. When stitched together, these patches form the final image, as you've done in chapter 11.

Like the BART encoder and decoder, the VQGAN detokenizer must be initialized with pretrained weights. The detokenizer is defined in the `vqgan_detokenizer.py` module of the min-DALL-E repository. We begin by ensuring the detokenizer weights are available:

```
detokenizer_path=folder+"/detoker.pt"
if not os.path.exists(detokenizer_path):
    print("downloading detokenizer parameters")
    ws = requests.get(url + 'detoker.pt')
    with open(detokenizer_path, 'wb') as f:
        f.write(ws.content)
else:
    print("detokenizer parameters have already been downloaded")
```

← Downloads the pretrained weights for the VQGAN detokenizer

This code block checks whether the file `detoker.pt` exists locally. If not, it downloads the file from the min-DALL-E repository on Hugging Face. Next, we initialize the detokenizer and load its parameters:

```
from min_dalle.models import VQGANDetokenizer

print("initializing VQGANDetokenizer")
detokenizer = VQGANDetokenizer().eval()
params = torch.load(detokenizer_path)
detokenizer.load_state_dict(params)
del params
detokenizer = detokenizer.to(device)
```

← Initializes the VQGAN detokenizer and sets it to evaluation mode

← Loads the pretrained weights

The detokenizer is set to `.eval()` mode to ensure that dropout and other training behaviors are disabled during inference.

### 12.4.2 Visualizing the intermediate and final high-resolution outputs

Now that our detokenizer is loaded, we can convert the generated image tokens into an actual image. Let's first visualize the final image corresponding to all 256 tokens:

```
from PIL import Image

torch.cuda.empty_cache()
image = detokenizer.forward(True,
                             image_tokens[:, 1:])
image = image.to(torch.uint8).to('cpu').numpy()
image = Image.fromarray(image)
image.save("files/minDALLE.png")
image
```

Decodes image tokens to an image tensor (skips the BOS token)

Converts the image tensor to NumPy arrays

Converts NumPy arrays to an actual image using the library

The tensor `image_tokens` has 257 elements; we skip the first one because it represents the BOS token. We use the pretrained VQGAN detokenizer to decode the remaining 256 image tokens to an image tensor. We then convert the image tensor to NumPy arrays and use the Python Imaging Library (PIL) to display the final image. After running the preceding code cell, you'll see an image `minDALLE.png` in the local folder, which looks similar to figure 12.4.

Figure 12.4 shows that the panda, hat, book, and reading posture are clearly represented, even though the prompt is informal. The boundaries between patches are nearly invisible, highlighting the decoder's ability to preserve texture, structure, and style during reconstruction.



**Figure 12.4** An image generated by the minDALL-E model with the prompt “panda with top hat reading a book”

#### Exercise 12.6

Display the generated image for the prompt “an underwater fantasy city.”

To better understand how the image “grows” during generation, we can visualize snapshots of intermediate outputs. This gives us a more intuitive understanding of what the model is doing at each decoding step.

#### Listing 12.8 Visualizing intermediate steps of image generation

```
from torchvision.utils import make_grid
from torchvision.transforms import ToPILImage
```

```

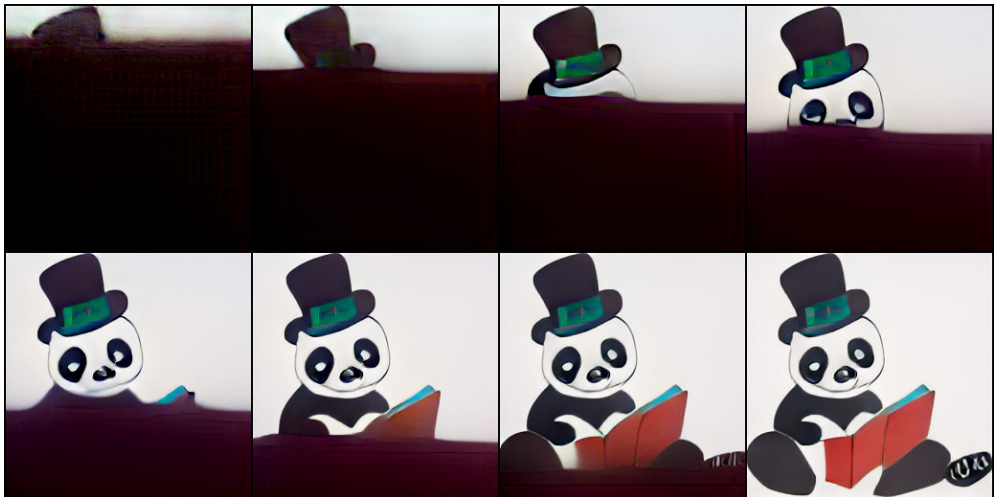
imgs=[]
for i in range(8):
    torch.cuda.empty_cache()
    images = detokenizer.forward(True,
        imgtokens[32*(i+1)-1][:, 1:])
    imgs.append(images.permute(2,0,1).to(torch.uint8))
grid=make_grid(imgs,nrow=4)
img=ToPILImage()(grid)
img.save("files/minDALLEsteps.png")
img

```

Selects images at steps 32, 64, 96, 128, 160, 192, 224, and 256

Places the eight images in a  $2 \times 4$  grid

The image generation takes 256 steps, with each step predicting one image token. We select the intermediate image after every 32 steps, and this results in eight intermediate images, at steps 32, 64, . . . , 224, and 256, respectively. We then place these eight images in a  $2 \times 4$  grid, as shown in figure 12.5. The model divides an image into 256 patches, organized in a  $16 \times 16$  grid. When generating an image based on a text prompt, the model first predicts the top-left patch. In the next iteration, the model predicts the patch next to it, based on the first patch and the prompt. The process is repeated until we have all the needed patches in the image. In this figure, the top-left subplot shows the output when 32 image patches are generated. The second subplot in the top row shows the output when 64 patches are generated. The rest of the images show the outputs when 96, 128, . . . , and 256 patches are generated.



**Figure 12.5** How the min-DALL-E model generates an image based on the prompt “panda with top hat reading a book”

Figure 12.5 shows the step-by-step progression of image generation. Each subplot corresponds to an intermediate image. After 32 tokens, only the top 1/8 of the image

is present; by 128 tokens, half of the image is recognizable. The final subplot (256 tokens) shows the complete result.

### Exercise 12.7

Display the 32nd, 64th, . . . , and 256th images in the list `imgtokens2` that you created in exercise 12.5 in a  $2 \times 4$  grid.

To wrap up, we'll animate the full token-by-token generation process using all 256 intermediate outputs.

### Listing 12.9 Animating the image-generation process

```
vids=[]
for i in range(len(imgtokens)):
    torch.cuda.empty_cache()
    images = detokenizer.forward(True,
    imgtokens[i][:, 1:])
    image = images.to(torch.uint8).to('cpu').numpy()
    vids.append(Image.fromarray(image))

import imageio
imageio.mimsave("files/minDALLE.gif",
    vids, fps=30)
```

Creates a list `vids` to store all intermediate images

Generates an image in each of the 256 intermediate steps

Uses the `imageio` library to create an animation

This will save a graphics interchange format (GIF) file named `minDALLE.gif` on your computer. You can watch the image form in real time. Alternatively, you can view it on the Google drive (<https://mng.bz/5vJ7>).

The animation serves as an illustration of autoregressive decoding in the generation process. Unlike traditional generative adversarial networks (GANs), which generate images in a single pass, models such as DALL-E allow us to peek inside the creative process, one token at a time.

### Exercise 12.8

Generate an animation using the 256 images in the list `imgtokens2` that you created in exercise 12.5.

This marks the completion of the min-DALL-E pipeline: from a natural language prompt, through encoding and decoding, all the way to pixel-level image generation.

In the next chapter, we'll explore how deep learning can be used to detect AI-generated images, focusing on methods to differentiate deepfakes from real photographs. While models such as DALL-E open up incredible creative potential, they also pose risks, and it's just as important to understand how to identify AI-generated content as it is to create it.

## Summary

- DALL-E mini is a project to replicate the proprietary DALL-E model, and min-DALL-E is the PyTorch reimplementation of DALL-E mini, which was built using the FLAX/JAX deep learning framework.
- To convert natural language to images, min-DALL-E first tokenizes and encodes the text prompt using a pretrained BART encoder, which converts the prompt into dense vector embeddings that condition the image-generation process.
- The min-DALL-E model then autoregressively generates 256 image tokens using a trained BART decoder, where each token corresponds to an image patch in a  $16 \times 16$  grid.
- A pretrained VQGAN decoder then converts the sequence of image tokens into a high-resolution color image by looking up visual features from a learned codebook.
- Building the min-DALL-E model from scratch allows us to understand how state-of-the-art text-to-image generators such as DALL-E work. The project provides hands-on experience with encoding, decoding, detokenization, and sampling strategies.



## *Part 5*

# *New developments and challenges*

**I**n the final chapter of this book, we step back and discuss the latest advances, challenges, and open questions in text-to-image generation. We begin with an overview of how state-of-the-art models such as OpenAI's DALL-E series, Google's Imagen, and Stable Diffusion (and its derivative, Midjourney) have revolutionized the field by translating natural language prompts into detailed, high-fidelity images. Despite these breakthroughs, the chapter emphasizes persistent challenges, such as geometric inconsistency, high computational and environmental costs, misuse through deepfakes, intellectual property disputes, and embedded social biases.

The second half of the chapter focuses on developing a defense mechanism against AI-generated fakes through deep learning. It provides a detailed, hands-on guide to fine-tuning ResNet-50 to classify real versus fake images. The chapter concludes by reinforcing the need for a balanced view, embracing the creative potential of generative AI while responsibly addressing the risks it poses to authenticity, ownership, and trust.



# 13

## *New developments and challenges in text-to-image generation*

---

### ***This chapter covers***

- How state-of-the-art text-to-image generators work
- Text-to-image model challenges and concerns
- Creating a model to distinguish real images from deepfakes
- Preparing a large-scale dataset of real and fake images for fine-tuning
- Testing fine-tuned models on unseen images

By now, we've explored two methods of text-to-image generation: building models from scratch and unlocking the creative potential of modern AI. From early transformer-based generators to cutting-edge diffusion models, we've seen how machines can now turn simple prompts into breathtaking images based on text prompts.

Yet, with these advances come profound new challenges. As the quality and realism of generated images have skyrocketed, so too have the risks. *Deepfakes*, AI-generated images and videos designed to deceive, are now increasingly indistinguishable from

real photographs. This presents not only technical hurdles but also ethical, legal, and societal dilemmas. How do we ensure these powerful tools are used responsibly? Can we reliably detect AI-generated images, and what are the broader consequences when detection fails?

This chapter brings together the state of the art in text-to-image generation with the urgent need for practical safeguards and critical reflection. We begin by looking at how the latest models, including OpenAI's DALL-E series, Google's Imagen, and innovations such as Midjourney, are pushing the boundaries of image generation. We then turn to the challenges facing the field, such as geometric inconsistency and the societal risks posed by misinformation, bias, and copyright concerns.

In direct response to these issues, we take a hands-on approach to defending against one of the most pressing threats: deepfakes. We show how a fine-tuned ResNet50 can be used to distinguish real images from those generated by AI. By using deep-learning skills you've learned so far, this chapter closes the loop: we not only generate images but also critically evaluate and defend against the risks these models pose.

As you work through this chapter, you'll gain a dual perspective: both the excitement of new creative possibilities and the responsibility to anticipate and address the challenges that follow. This holistic view is essential for anyone hoping to innovate and safeguard the future of generative AI.

## 13.1 *State-of-the-art text-to-image generators*

The burgeoning field of text-to-image generation now boasts sophisticated models capable of generating highly detailed images purely from text prompts. This section discusses the main ideas behind three leading models: OpenAI's DALL-E series, Google's Imagen, and the powerful latent diffusion models (notably Stable Diffusion and its derivative, Midjourney).

### 13.1.1 *DALL-E series*

By harnessing the combined strengths of transformer architectures and advanced image-generation techniques, the DALL-E series of models has pushed the boundaries of what's achievable in generating images from text. The original DALL-E model, introduced by OpenAI researchers in the seminal paper "Zero-Shot Text-to-Image Generation" in 2021 [1], showcased the transformative potential of transformer-based architectures in multimodal generation tasks. Using an architecture derived from GPT-3, DALL-E was trained to generate images directly from descriptive natural language prompts. DALL-E used a combination of two key strategies (as we discussed in chapter 12):

- *Image tokenization*—Using vector quantized variational autoencoders (VQ-VAE), images were compressed into discrete visual tokens, effectively representing images as sequences of integers.
- *Joint embedding space*—A transformer-based language model then mapped textual prompts into the same latent space, enabling the model to learn correspondences between text embeddings and image tokens.

During inference, DALL-E generated images by first translating the input text prompt into embeddings, predicting corresponding image tokens, and finally reconstructing these tokens back into coherent images using a pretrained VQGAN decoder. Despite being an initial attempt, DALL-E effectively demonstrated a remarkable ability to capture the essence of complex textual concepts, sparking widespread interest and enthusiasm in both academic and industry communities.

Building on the foundation set by its predecessor, DALL-E 2 introduced substantial improvements, especially in the areas of image quality, coherence, and semantic alignment with textual prompts. Detailed in OpenAI's paper "Hierarchical Text-Conditional Image Generation with CLIP Latent" in 2022 [2], DALL-E 2 incorporated the contrastive language-image pretraining (CLIP) model (discussed in chapter 8) to significantly enhance image-text alignment.

In the most recent advancement, DALL-E 3, presented in the paper "Improving Image Generation with Better Captions in 2023 [3], further refined and expanded the capabilities of its predecessor. DALL-E 3 used a more advanced generative pretraining paradigm, incorporating improved caption-aware contextual modeling to maintain close adherence to complex textual instructions. As a result, the model could accurately depict complex scenes, subtle visual relationships, and abstract concepts with far greater fidelity than previous versions. Its ability to generate images that accurately reflect detailed and sometimes abstract descriptions has expanded the creative ability of text-to-image technologies, demonstrating their potential to serve as indispensable tools across various professional domains.

### 13.1.2 Google's Imagen

Introduced by researchers at Google in 2022, Imagen represents a significant leap forward in text-to-image generation, distinguished by its extraordinary photorealism and attention to detail [4]. Central to Imagen's exceptional capability is its use of pretrained text encoders, specifically Google's T5-XXL model, a massive transformer architecture originally designed for comprehensive natural language understanding and generation tasks. When provided with a textual prompt, Imagen first uses T5-XXL to transform the text into rich, contextually dense embeddings. These embeddings capture the meaning of the prompt, effectively encoding the underlying semantics necessary for precise image generation.

Imagen further distinguishes itself through a unique multistage cascaded diffusion model. Unlike simpler diffusion approaches, which directly generate images at a single resolution, Imagen uses a sequential refinement strategy across multiple resolutions, beginning with a low-resolution image (typically  $64 \times 64$  pixels) and progressively enhancing image detail and clarity through iterative refinement.

Imagen's training relies on massive datasets containing millions of image-caption pairs. The pretrained T5-XXL model is used to generate text embeddings that align with visual semantics (though in most implementations, T5 remains frozen rather than fine-tuned). The cascaded diffusion U-Nets are then trained sequentially while the T5

encoder stays fixed. During inference, Imagen follows an intuitive and structured workflow to generate high-resolution images from user-supplied prompts:

- 1 *Textual embedding generation*—The input text prompt is encoded using the pre-trained T5-XXL model, producing context-rich text embeddings.
- 2 *Low-resolution image generation*—The initial diffusion model generates a preliminary low-resolution image ( $64 \times 64$  pixels), capturing essential semantic features and composition dictated by the prompt.
- 3 *Cascaded refinement process*—Successive diffusion models iteratively upsample and refine this initial image, incrementally increasing image resolution and fidelity. Each refinement stage ensures consistency with both the original textual embedding and previously generated lower-resolution outputs.
- 4 *Final high-resolution output*—After the cascaded refinement, the output is a high-resolution image (typically  $1,024 \times 1,024$  pixels), characterized by exceptional photorealism, clarity, and detailed semantic accuracy.

Google's Imagen significantly elevated the state-of-the-art in text-to-image generation through its innovative integration of pretrained textual encoders and a cascaded diffusion architecture. By effectively combining language understanding capabilities with progressive image refinement, Imagen achieves an unprecedented balance between semantic precision and photorealistic detail.

### 13.1.3 *Latent diffusion models: Stable Diffusion and Midjourney*

Latent diffusion models, as we explored in detail in chapter 9, have emerged as a powerful paradigm in text-to-image generation. Unlike conventional diffusion models, which operate directly on pixel space, latent diffusion models first encode images into a lower-dimensional latent representation. This approach substantially reduces computational costs while maintaining, or even enhancing, visual quality and semantic fidelity. Among the most influential latent diffusion-based models are Stability AI's Stable Diffusion, which we discussed extensively in chapter 10, and its derivative, Midjourney, which extends the core capabilities of Stable Diffusion into user-friendly, creatively empowering applications.

Stable Diffusion's innovative approach significantly lowers computational barriers compared to previous generation models, making it accessible to a broader audience of artists, researchers, and enthusiasts. The model's open source release has further accelerated its widespread adoption and creative exploration, profoundly impacting generative AI communities and fostering rapid innovation across diverse industries.

Midjourney represents a compelling application built on Stable Diffusion. Although technically rooted in the same latent diffusion framework, Midjourney distinguishes itself through substantial enhancements aimed specifically at user interaction, creative exploration, and real-time collaboration. Unlike Stable Diffusion, which emphasizes fundamental generative capabilities, Midjourney's core contributions lie in advanced user-centric innovations:

- *Intuitive prompt engineering*—Midjourney integrates sophisticated natural language processing (NLP) features, enabling users to craft detailed prompts with ease. The platform offers guided prompt suggestions, adaptive semantic interpretations, and interactive refinement options, significantly enhancing the creative flexibility and accuracy of generated images.
- *Real-time generation and refinement*—Midjourney incorporates real-time feedback and iterative refinement capabilities, allowing users to adjust their prompts dynamically, instantly observing visual outcomes. This iterative workflow transforms text-to-image generation into an intuitive, iterative process, closely resembling traditional artistic methods.
- *Collaborative and interactive features*—Recognizing the inherently collaborative nature of creative endeavors, Midjourney provides a shared, collaborative workspace where multiple users can simultaneously generate, refine, and critique images. This interactive platform promotes collective creativity, facilitating inventive dialogues and collaborative design processes previously unfeasible with traditional generative models.
- *Advanced artistic and stylistic controls*—Beyond mere prompt-based generation, Midjourney offers explicit stylistic and compositional controls. Users can effortlessly specify artistic styles, color schemes, lighting conditions, and compositional constraints, effectively turning generative AI into an adaptable, personalized creative assistant.

Midjourney likely uses diffusion-style methods, though its exact architecture, training data, and implementation details remain undisclosed. By extending the capabilities of generative models through an accessible, user-friendly interface, it demonstrates how such technologies can become powerful tools for practical artistic and professional workflows.

## 13.2 Challenges and concerns

The state-of-the-art text-to-image generative models also face many challenges and concerns. Four main issues are highlighted here:

- *Geometric inconsistency problem*—Text-to-image models, whether based on transformers or diffusion models, often struggle with maintaining geometric consistency because they are typically trained on large, diverse datasets of 2D images without a deep understanding of 3D structure or physical rules governing spatial relationships. Therefore, these models don't inherently understand 3D geometry. They learn patterns and associations between pixels and the semantic meaning of text, but they lack an internal representation of the three-dimensional world, which is essential for maintaining consistent spatial relationships across objects and viewpoints.

Transformer-based models use positional encodings to understand spatial relationships, but these are limited when it comes to enforcing strict geometric

constraints, especially across complex 3D objects. Current text-to-image models typically lack a robust mechanism for encoding depth and perspective consistently. These models don't understand the rules of perspective, lighting, or physics like humans do. As a result, they might generate objects that defy physical laws or appear distorted when viewed from certain angles, which breaks geometric consistency. Achieving geometric consistency would likely require incorporating models that explicitly consider 3D structure, physics simulations, or other methods that enforce spatial coherence.

- *Computational costs and environmental impacts*—The development and deployment of all the state-of-the-art text-to-image models involve high computational costs. Training these models requires significant resources, including large datasets and powerful hardware. The energy consumption associated with training can be substantial. The environmental impact of these models is a serious concern. While techniques such as pruning, quantization, and transfer learning have been used to mitigate these costs, the scalability of text-to-image models remains a pressing issue. Reducing the computational footprint while maintaining high-quality output is a major area of ongoing research.
- *Misuse of the output*—The capacity to generate realistic images opens up possibilities for misuse, such as deepfakes or misinformation. These generative models can be exploited for creating and spreading misinformation, propaganda, or manipulative content. This poses significant risks to public discourse, democracy, and trust in information. The solution to this concern lies in developing robust content-moderation systems. Establishing guidelines for responsible use and fostering collaborations between AI developers, policymakers, and media organizations are crucial steps in combating misinformation. Later in this chapter, we'll provide a deep learning approach for differentiating real images from deepfakes using a fine-tuned ResNet architecture.
- *Intellectual property rights*—The training data used in these models is often scraped from the web without the explicit consent of the original creators, raising questions about the ownership of AI-generated images. Supporters argue that the way data is used to train these generative models is transformative: the model doesn't merely regurgitate the data but uses it to generate new, original content. This transformation could qualify under the *fair use doctrine*, which allows limited use of copyrighted material without permission if the use adds new expression or meaning. Critics argue that these generative models are trained on vast amounts of copyrighted images and texts without permission, which goes beyond what might be considered fair use. The scale of data used and the direct ingestion of copyrighted material without transformation during training could be seen as infringing. The debate is ongoing. It's a debate that likely needs to be resolved by legislative and judicial bodies to provide clear guidelines and ensure that the interests of all parties are fairly represented.



Finally, text-to-image models trained on broad datasets often incorporate inherent biases related to race, gender, or culture. For example, generating images from a prompt like “a doctor” or “a leader” might disproportionately yield stereotypical representations. This bias is not just a technical problem but a social one, as it reflects and amplifies existing inequalities in society. Developing methods to audit and mitigate bias in these models is an ongoing challenge, requiring a concerted effort from both AI developers and social scientists. However, we must keep in mind not to overcorrect. A counterexample is that Google’s Gemini overcorrected the stereotypes in image generation by including people of color in groups of Nazi-era German soldiers (<https://mng.bz/645R>).

Text-to-image generative AI models have made tremendous progress in the past few years, becoming transformative tools in art, design, and AI. However, as these models become more integrated into society, addressing the challenges of resource consumption, ethical use, and bias will be essential for their responsible and sustainable development.

### 13.3 A blueprint to fine-tune ResNet50

We currently live in a world where digital content is rapidly generated and shared. The ability to distinguish between real images and deepfakes (AI-generated images that convincingly mimic real people or objects) has become crucial. Deepfakes are increasingly prevalent, raising concerns about misinformation, privacy, and security. As these deepfakes grow more sophisticated, traditional methods of image verification struggle to keep up. This challenge demands innovative solutions grounded in deep learning and computer vision techniques.

This chapter aims to provide a robust approach for differentiating real images from deepfakes. Instead of training a new model from scratch, we’ll fine-tune a ResNet50 model that’s already learned general visual features from a large dataset. By replacing and retraining only the final layers, we can quickly adapt ResNet50 to classify images as real or fake, saving time and resources.

We’ll introduce the key features and strengths of ResNet50 and explain the advantages of fine-tuning over training from scratch. We’ll also outline the steps to modify and prepare ResNet50 for our deepfake-detection task. Finally, we’ll explore how to use a pretrained ResNet50 to classify some example images, demonstrating its powerful feature extraction abilities and showing its workflow.

The Python program you’re about to use is adapted from two wonderful GitHub repositories: <https://mng.bz/oZBD> and <https://github.com/cyb0rg14/fake-image-detector>. Before you start, run the following code cell in Jupyter Notebook to install the kagglehub library for this chapter:

```
!pip install kagglehub
```

#### 13.3.1 The history and architecture of ResNet50

ResNet50 is a landmark architecture in deep learning, introduced in 2015 by Kaiming He et al. at Microsoft Research [5]. Its key innovation, the residual connection,

transformed how deep neural networks are built and trained, allowing them to go much deeper than before without suffering from the vanishing gradient problem.

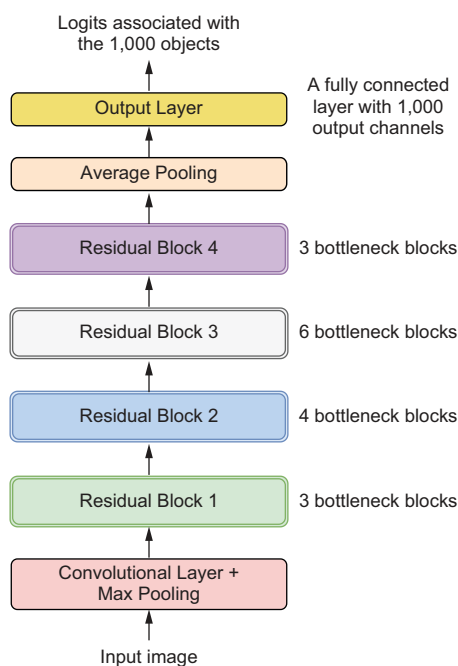
Traditional deep networks struggled as layers increased, often failing to learn meaningful features. Residual Network (ResNet) tackled this by adding shortcut, or “skip,” connections that allow the input of a layer to bypass one or more layers and be added directly to the output. This simple yet powerful idea helps preserve information and enables the effective training of much deeper models. In fact, the “residual connection” concept has become foundational, now appearing in everything from vision models to transformers for NLP.

ResNet50 is one of the most popular variants, featuring 50 layers organized into modular blocks. Its architecture enables the network to learn hierarchical and abstract representations of images, which is why it has performed exceptionally well on large-scale datasets such as ImageNet (over 1 million images and 1,000 object classes). The availability of high-quality pretrained weights makes ResNet50 a go-to choice for transfer learning in many real-world applications.

Figure 13.1 illustrates the main building blocks of ResNet50, structured as follows: (1) an initial convolutional layer followed by max pooling to reduce spatial dimensions, (2) four sequential “residual blocks” composed of bottleneck units with residual connections, (3) an average pooling layer, and (4) a fully connected output layer. The model takes an image as input and outputs logits for 1,000 object classes, which can be converted into predicted probabilities using the softmax activation function.

The input to ResNet50 is an input image, which will first go through a convolutional layer and a max pooling layer to extract low-level features and reduce the image size. The heart of the network consists of four residual blocks, each made up of several bottleneck sub-blocks. Each bottleneck uses three convolutional layers with a residual connection, which allows gradients and information to flow more easily through the network as it learns.

After passing through all the residual blocks, the features are aggregated by an average pooling layer, and the network ends with a fully connected output layer. Originally, this layer outputs 1,000 values (called logits), one for each possible ImageNet class (e.g.,



**Figure 13.1** The architecture of ResNet50

bee, ant, zebra). After applying the softmax activation function on these logits, we can obtain the predicted probability distribution over the 1,000 classes.

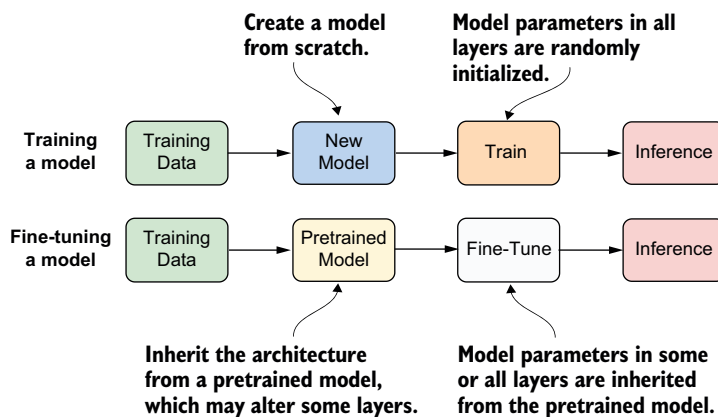
The modular and extensible design of ResNet50 makes it ideal for fine-tuning. By swapping out or modifying the output head, you can quickly adapt the model to new tasks, such as distinguishing real images from deepfakes.

### 13.3.2 A plan to fine-tune ResNet50 for classification

Fine-tuning is one of the most effective strategies in modern deep learning, especially when you want to adapt a powerful model such as ResNet50 to a new, specialized task, such as distinguishing real images from deepfakes. Instead of starting from scratch, we take advantage of the knowledge that ResNet50 has already acquired from massive datasets such as ImageNet and repurpose it for our classification problem.

To understand why this approach is so powerful, let's first contrast training from scratch with fine-tuning a pretrained model. When you train a deep neural network from scratch, you begin with randomly initialized parameters. The model must "learn everything" from scratch, which requires vast amounts of data and computational resources. This process is time-consuming and often impractical for most real-world projects.

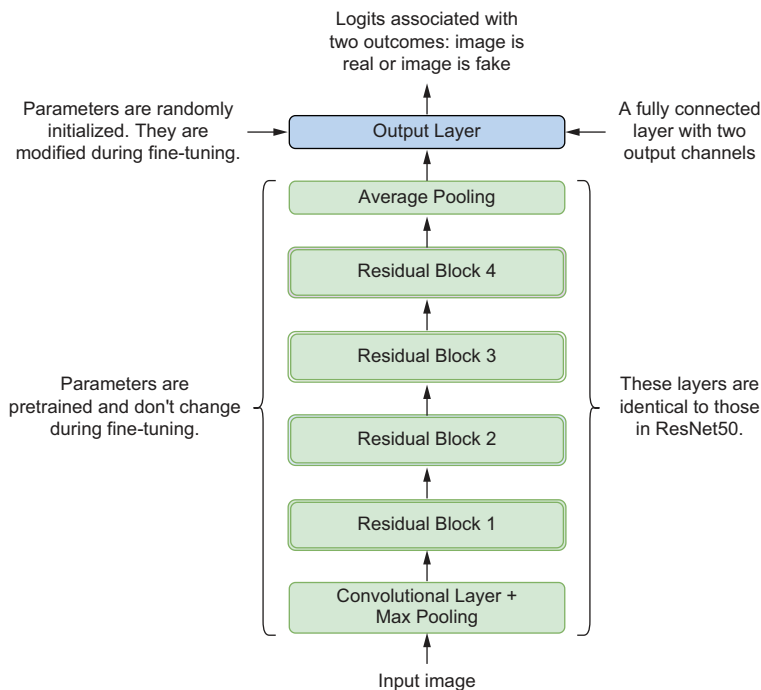
In contrast, fine-tuning starts with a model that already knows how to perform general tasks (extracting useful visual features from images in the case of ResNet50). By updating only the last few layers, typically the output head, you can quickly adapt the model to new tasks with far less data and time. Figure 13.2 illustrates the difference between training from scratch and fine-tuning.



**Figure 13.2** Training from scratch vs. fine-tuning an existing model. Training from scratch (top row in the diagram) involves building a model with randomly initialized parameters, requiring the network to learn all features from raw data. In contrast, fine-tuning uses the architecture and pretrained weights of an existing model, modifying (often only) the output layer for a new downstream task. The bulk of the network retains the learned parameters, while only the final layers are updated for the new objective.

This fine-tuning approach is the secret to achieving state-of-the-art performance with limited resources. For our task, it means we can use a ResNet50 model pretrained on ImageNet and tailor it to distinguish real from fake images simply by retraining the output head. Specifically, the original ResNet50 outputs 1,000 logits, one for each object class in ImageNet. For deepfake detection, we only need to classify images into two categories: real or fake.

Our plan involves replacing the final fully connected layer (the output head) of ResNet50 with a new one that has just two output channels. All other layers of the network are left unchanged, preserving their pretrained parameters. Only the parameters in the new output layer will be randomly initialized and updated during fine-tuning. Figure 13.3 is the architecture of the modified model we'll use in this chapter.



**Figure 13.3** Modified ResNet50 architecture for deepfake detection. All original layers of ResNet50 are retained, except for the last fully connected output layer. The original output head (with 1,000 output channels) is replaced with a new head (with 2 output channels), producing logits for the “real” and “fake” classes. Parameters in the new output layer are initialized randomly and trained on our dataset. All other parameters are inherited from the pretrained model and kept frozen during fine-tuning.

By making this simple yet powerful adjustment, we harness the deep feature extraction capabilities of ResNet50 while customizing it for our classification goal. In summary, our fine-tuning strategy consists of the following steps:

- 1 Load a ResNet50 model pretrained on ImageNet.
- 2 Replace its final output layer with a new layer for two-class output.
- 3 Freeze the weights of all other layers to retain their learned features.
- 4 Train only the new output head on our dataset of real and fake images.

This approach not only dramatically reduces training time and data requirements but also consistently delivers high performance for tasks where labeled data is limited. In the next sections, we'll implement each of these steps in code, fine-tuning ResNet50 to become a highly effective deepfake detector.

### 13.3.3 Using ResNet50 to classify images

Before we fine-tune ResNet50 for detecting deepfakes, it's valuable to see how this model works out of the box. In this subsection, we use a pretrained ResNet50 to classify some example images into one of the 1,000 categories in ImageNet. By working through hands-on examples, you'll gain insight into both the capabilities and the inner workings of ResNet50.

One of the major advantages of using the torchvision library is the ease with which you can access leading-edge, pretrained models such as ResNet50. Directly accessing the pretrained weights, learned from millions of images in the ImageNet dataset, lets us instantly benefit from powerful feature extraction without having to start from scratch.

Let's walk through the process of loading a ResNet50 model with pretrained weights, set it up for inference, and examine its architecture. These steps provide a crucial foundation for both experimentation and later customization. First, we'll import the relevant module and load the pretrained ResNet50 model:

```
import torchvision.models as models

weights = models.ResNet50_Weights.DEFAULT
model = models.resnet50(weights=weights)
model.eval()
print(model)
```

Initiates ResNet50 with pretrained weights

Sets the model to evaluation mode for inference

Prints out the model structure

We create an instance of the ResNet50 model, specifying that it should load the default pretrained weights. Setting the model to evaluation mode is essential before using it for inference. This ensures that layers such as dropout and batch normalization behave appropriately, producing deterministic and reliable results. Printing the model structure allows you to inspect all the layers and modules that make up ResNet50. The output is as follows:

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
```

```

(maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1,
dilation=1, ceil_mode=False)
(layer1): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3),
stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(64, 256, kernel_size=(1, 1),
stride=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
...
)
)
(avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
(fc): Linear(in_features=2048, out_features=1000, bias=True)
)

```

For brevity, we omitted most of the output, which is a detailed summary of the ResNet50 architecture. The output starts with the initial convolutional and pooling layers, followed by the four main residual blocks, and ending with the average pooling and fully connected output layer.

**NOTE** The Python programs for this chapter can be accessed from the Google Colab notebook (<https://mng.bz/nZld>) and the GitHub repository (<https://github.com/markhliu/txt2img>).

Go to the book's GitHub repository, and download the image `fish.png`. Place it in the `/files/` folder on your computer (or simply clone the repository to your computer). We'll use the fish image as our sample picture to show how ResNet50 works. The ResNet50 model can only accept images of size  $3 \times 224 \times 224$ , so we load the fish image first and define a transform method to convert the image to the desired shape.

#### Listing 13.1 Loading and processing the sample image

```

from PIL import Image
import torch

img = Image.open('files/fish.png').convert('RGB')  ← Reads the sample image

```

```

preprocess = models.ResNet50_Weights.DEFAULT.transforms()
input_tensor = preprocess(img)
input_batch = input_tensor.unsqueeze(0)
print(f"the shape of the input image is {input_batch.shape}")
device="cuda" if torch.cuda.is_available() else "cpu"
input_batch = input_batch.to(device)
model.to(device)

```

Transforms the image into a PyTorch tensor with the desired size

Creates a mini-batch as expected by ResNet50

Moves the input and model to GPU if available

The output from the listing 13.1 is

```
the shape of the input image is torch.Size([1, 3, 224, 224])
```

The shape of the input image is (1, 3, 224, 224), meaning there is one image in the batch, and the size of the image is  $3 \times 224 \times 224$ .

### Exercise 13.1

Go to the book's GitHub repository, and download the image `deer.png`. Place it in the `/files/` folder on your computer. Load the image, and transform it into a format that ResNet50 can accept as input. Call the input tensor `input_batch2`.

With our pretrained ResNet50 model loaded and ready, we can now see it in action by making predictions on real-world images. This section guides you through the process of classifying images using the model and interpreting the results, offering practical insights into both the mechanics and the strengths of deep convolutional networks.

Let's start with the fish image that you've preprocessed and formatted to the required size. We can now run this image through the model and observe how it produces predictions:

```

from torch.nn.functional import softmax

with torch.no_grad():
    output = model(input_batch)
    probabilities = softmax(output[0], dim=0)
    predicted_class_idx = torch.argmax(probabilities).item()
    highest_prob = torch.max(probabilities).item()
    print("the predicted class is", predicted_class_idx)
    print("the highest probability is", highest_prob)

```

Makes a prediction on the sample image

Converts logits to probabilities

Obtains the class index with the highest probability

Obtains the highest probability

The model processes the input image and produces raw outputs (logits), one for each possible class. We use the softmax function to convert these logits into a probability distribution over the 1,000 ImageNet classes. The `argmax()` function identifies the index

of the class with the highest predicted probability. We also print out the probability value for the top prediction using the `max()` function.

The output from the preceding code block is

```
the predicted class is 1
the highest probability is 0.1901196837425232
```

The predicted class index is 1, which means the model believes the image most likely belongs to class 1 (out of 1,000), with an associated probability of 19%. But what does this class index represent in human terms? To answer that, we can map class indices to actual category names:

```
labels = models.ResNet50_Weights.DEFAULT.meta['categories']
print(labels)
print("object name is:", labels[predicted_class_idx])
```

Prints out the 1,000 object names

Prints out the name of the object with index 1

This prints out all 1,000 object categories from ImageNet. Using the class index, we retrieve and print the name of the predicted class. The output is

```
['tench', 'goldfish', 'great white shark', 'tiger shark',
 'hammerhead', 'electric ray', 'stingray',
 ...,
 'gyromitra', 'stinkhorn', 'earthstar', 'hen-of-the-woods',
 'bolete', 'ear', 'toilet tissue']
object name is: goldfish
```

For brevity, we omitted most of the output. We can see that the first of the 1,000 objects is `tench`, the second object is `goldfish`, and the last object is `toilet tissue`. Because Python uses zero-based indexing, the class index 1 corresponds to the second object, `goldfish`.

### Exercise 13.2

Feed the input tensor `input_batch2` that you generated in exercise 13.1 to ResNet50, and print out the predicted class index as well as the name of the predicted object.

To better understand the model's behavior, let's try running it on a variety of images. For this, we'll use four images—a bird, a dog, a bunny, and a fish—preprocessed in the same way. We'll collect both the predicted labels and their associated probabilities for each image. Go to the book's GitHub repository, download the three images `bird.png`, `dog.png`, and `bunny.png`, and place them in the `/files/` folder on your computer



(or clone the repository to your computer). We use ResNet50 to make predictions on these images and store the predicted probabilities and the outcomes.

### Listing 13.2 Using ResNet50 to make predictions

```
probs=[]
indexes=[]
images=["bird","dog","bunny","fish"]
for img in images:
    img = Image.open(f"files/{img}.png").convert('RGB')
    input_tensor = preprocess(img)
    input_batch = input_tensor.unsqueeze(0).to('cuda')
    with torch.no_grad():
        output = model(input_batch)
        probabilities = softmax(output[0],dim=0)
        idx=torch.argmax(probabilities).item()
        indexes.append(idx)
        prob=torch.max(probabilities).item()
        probs.append(prob)
```

← Makes predictions on the four images

← Obtains the class indices

← Obtains the highest probabilities

We iterate through the four images. For each image, we use ResNet50 to make a prediction and obtain the index with the highest probability and the associated probability. We store the four probabilities in the list `probs` and the four indices in the list `indexes`. Now we can visualize the results by displaying the images alongside their predicted class names and confidence scores.

### Listing 13.3 Plotting predictions and the associated probabilities

```
from torchvision import transforms
import matplotlib.pyplot as plt

plt.figure(figsize=(8,5),dpi=100)
for i in range(4):
    plt.subplot(1,4,i+1)
    img=Image.open(f"files/{images[i]}.png")
    img=transforms.ToTensor()(img)
    plt.imshow(img.permute(1,2,0))
    pred=labels[indexes[i]]
    msg=f"{pred}\nprob: {probs[i]:.2f}"
    plt.title(msg,fontsize=15)
    plt.axis('off')
plt.tight_layout()
plt.show()
```

← Displays the four images

← Displays the predictions and the associated probabilities

Each subplot displays one of the four images. The title above each image shows the predicted class name and the model's confidence. The output is shown in figure 13.4.

This hands-on exploration demonstrates the predictive power and limitations of ResNet50 in its default setting. Understanding these behaviors is essential before we move on to fine-tuning the model for our custom classification task, detecting real images and deepfakes.



**Figure 13.4** Using ResNet50 to make predictions on images. Each image is shown with its top predicted class and the corresponding probability as determined by ResNet50. The results highlight both the strengths and quirks of large-scale classification models; for example, the model may predict “bee eater” for a bird image that looks like a kingfisher because “kingfisher” isn’t among the 1,000 ImageNet classes. Such observations show that the model’s predictions depend not only on the input image but also on the set of categories it was trained to recognize.

### 13.4 *Fine-tuning ResNet50 to detect fake images*

Having seen how ResNet50 performs on general image classification tasks, we now turn to the challenge of adapting it for deepfake detection. The first step in this process is to prepare a dataset that enables the model to learn the subtle differences between real and AI-generated (fake) images. We’ll gather a large number of images, clearly labeled as either “real” or “fake.” The more representative and diverse the dataset, the better the model will be at generalizing to new, unseen images. We’ll then transform a powerful pretrained ResNet50 into a specialized detector that can distinguish between real and AI-generated (deepfake) faces. You’ll learn how to do the following:

- Gather and organize a large number of images that are clearly labeled as either “real” or “fake.” The dataset serves as our training data.
- Freeze the weights of the early layers of ResNet50 to preserve their generic image understanding.
- Replace the output “head” of the network to support real versus fake classification.
- Train just the new output layers using your custom dataset while monitoring validation performance.
- Implement techniques such as early stopping to avoid overfitting and to ensure the model generalizes well to unseen images.

By the end of this section, you’ll have all the practical tools and knowledge required to adapt a cutting-edge image classification model to a deepfake detection task.

#### 13.4.1 *Downloading and preprocessing real and fake face images*

We’ll use the 140k Real and Fake Faces dataset from Kaggle, which provides a rich source of both genuine and AI-generated images. This collection contains a total of 140,000 images: 70,000 real faces sourced from real-world photographs and 70,000

fake faces generated by an AI model. The dataset is organized into clearly labeled folders for “real” and “fake” images and is split into training, validation, and test subsets for evaluation and testing. Let’s start by downloading the dataset and setting up the file paths.

#### Listing 13.4 Downloading the 140k Real and Fake Faces dataset

```
import kagglehub

path = kagglehub.dataset_download(
    "xhluLu/140k-real-and-fake-faces")

print("Path to dataset files:", path)

train_dir=f"{path}/real_vs_fake/real-vs-fake/train"
val_dir=f"{path}/real_vs_fake/real-vs-fake/valid"
test_dir=f"{path}/real_vs_fake/real-vs-fake/test"
```

Downloads the dataset from Kaggle

Prints out the local path to the dataset files

Defines the paths to the train, validation, and test subsets

We first use the `dataset_download()` method in the `kagglehub` library to download the dataset. We then print out the path to the dataset files. Depending on your operating system, the path may differ. I’m using Windows 11, and the output on my computer is as follows:

```
Path to dataset files: C:\Users\hlliu2\.cache\kagglehub\datasets\
xhluLu\140k-real-and-fake-faces\versions\2
```

If you’re using Google Colab, the output is

```
Path to dataset files:
/kaggle/input/140k-real-and-fake-faces
```

We then define the paths to the train, validation, and test subsets as `train_dir`, `val_dir`, and `test_dir`, respectively. We count the number of images in the three subsets in the following listing.

#### Listing 13.5 Counting and displaying real and fake face images

```
import os
import glob
import matplotlib.pyplot as plt
import PIL

def count_images(folder):
    for path, name, files in os.walk(folder):
        if len(files)>0:
            print(f"There are {len(files)} images in {path}.")
count_images(train_dir)
count_images(val_dir)
count_images(test_dir)

imgs=glob.glob(f'{train_dir}/real/*.jpg')[66:66+5]+\\
```

Counts the number of images in train, validation, and test subsets

```

glob.glob(f'{train_dir}/fake/*.jpg')[88:88+5]
plt.figure(figsize=(10,4),dpi=100)
for i in range(10):
    ax = plt.subplot(2,5, i + 1)
    img=PIL.Image.open(imgs[i])
    plt.imshow(img)
    plt.xticks([])
    plt.yticks([])
    if i<5:
        plt.title("real image")
    else:
        plt.title("fake image")
    plt.tight_layout()
plt.show()

```

← Selects five real images and five fake images

← Displays the 10 images in a 2 × 5 grid

The output from the code listing is

```

There are 50000 images in C:\Users\hlliu2\.cache\kagglehub\
datasets\hllulu\140k-real-and-fake-faces\versions\2\
real_vs_fake/real-vs-fake/train/fake.
There are 50000 images in C:\Users\hlliu2\.cache\kagglehub\
datasets\hllulu\140k-real-and-fake-faces\versions\2\
real_vs_fake/real-vs-fake/train/real.
There are 10000 images in C:\Users\hlliu2\.cache\kagglehub\
datasets\hllulu\140k-real-and-fake-faces\versions\2\
real_vs_fake/real-vs-fake/valid/fake.
There are 10000 images in C:\Users\hlliu2\.cache\kagglehub\
datasets\hllulu\140k-real-and-fake-faces\versions\2\
real_vs_fake/real-vs-fake/valid/real.
There are 10000 images in C:\Users\hlliu2\.cache\kagglehub\
datasets\hllulu\140k-real-and-fake-faces\versions\2\
real_vs_fake/real-vs-fake/test/fake.
There are 10000 images in C:\Users\hlliu2\.cache\kagglehub\
datasets\hllulu\140k-real-and-fake-faces\versions\2\
real_vs_fake/real-vs-fake/test/real.

```

Here, we select five real and five fake images from the training subset and then arrange them in a  $2 \times 5$  grid for easy comparison. After running the code cell, you'll see images similar to those shown in figure 13.5.

The fake images look very real. It's hard to tell real images and fake images apart by just looking. We'll train a deep learning model for this purpose.

### Exercise 13.3

Display the first five real images and the last five fake images in the validation subset in a  $2 \times 5$  grid.

Once the dataset of real and fake faces has been downloaded and inspected, the next step is to prepare it for training our deepfake detector. Deep learning models such as



**Figure 13.5** Sample real and fake images from the 140k Real and Fake Faces dataset. The top row displays five real face images, while the bottom row shows five face images generated by AI. At a glance, both sets look convincingly realistic. This side-by-side comparison highlights the subtlety of the deepfake detection challenge: to the naked eye, even AI-generated images can appear authentic. Such visualizations emphasize the need for advanced machine learning methods to distinguish between real and fake images.

ResNet50 expects inputs to be preprocessed in specific ways, especially regarding image size, normalization, and batching, so it's important to handle these details carefully to ensure effective training.

ResNet50, as trained on ImageNet, expects input images to be in the shape (3, 224, 224) (three color channels, height and width of 224 pixels), and normalized using the same statistics as the original dataset. We must ensure that all images in our dataset are transformed accordingly before feeding them into the model.

#### Listing 13.6 Converting images to the desired size

```
from torchvision import datasets, models
import torch

torch.manual_seed(42)
preprocess = models.ResNet50_Weights.DEFAULT.transforms()

train_data = datasets.ImageFolder(
    root=train_dir,
    transform=preprocess,
    target_transform=None)
val_data = datasets.ImageFolder(
    root=val_dir,
    transform=preprocess,
    target_transform=None)
test_data = datasets.ImageFolder(
    root=test_dir,
    transform=preprocess)
```

Processes images in the train subset

Processes images in the validation subset

Processes images in the test subset

We define preprocess as a transformation pipeline automatically provided for ResNet50, ensuring each image is resized, cropped, converted to a tensor, and normalized in a way that matches the expectations of the pretrained model.

The datasets.ImageFolder utility loads images from disk, applying the transformation pipeline to every image in the respective folder. This structure allows us to organize our dataset conveniently; each subfolder name (real, fake) is treated as a label.

After transformation, we batch the data. Training in batches is a standard practice in deep learning. It allows for efficient computation and more stable gradient updates:

```
from torch.utils.data import DataLoader
train_dataloader = DataLoader(dataset=train_data,
                             batch_size=32,
                             shuffle=True)
val_dataloader = DataLoader(dataset=val_data,
                           batch_size=32,
                           shuffle=False)
test_dataloader = DataLoader(dataset=test_data,
                            batch_size=32,
                            shuffle=True)
```

← Puts training data into batches of 32 observations

← Does the same for validation and test subsets

The result of this data pipeline is that whenever we iterate over these DataLoader objects, we receive a batch of images (as tensors, ready for input to ResNet50) along with their corresponding labels (“real” or “fake,” encoded as 0 or 1).

### 13.4.2 Fine-tuning ResNet50

The key to fine-tuning is deciding which parts of the network to retrain and which to keep fixed. In most transfer learning workflows, we “freeze” the weights of all layers except the final output head. This preserves the network’s general visual feature extraction ability, while letting the last layer learn how to map those features to our specific labels of “real” or “fake.” The following code demonstrates this process.

#### Listing 13.7 Replacing the output head in ResNet50

```
for param in model.parameters():
    param.requires_grad = False
model.fc= torch.nn.Sequential(
    torch.nn.Linear(2048,1000),
    torch.nn.ReLU(),
    torch.nn.Linear(1000,500),
    torch.nn.Dropout(),
    torch.nn.Linear(in_features=500,
                    out_features=2,
                    bias=True)).to(device)
num_trainable=sum([p.numel() for p in model.parameters()
                  if p.requires_grad])
print(f"Number of trainable parameters: {num_trainable}")
non_trainable=sum([p.numel() for p in model.parameters()])
```

← Freezes parameters in ResNet50

← Replaces the output head so that there are two output channels

```

        if not p.requires_grad]]
print(f""Number of untrainable parameters:
      {non_trainable}""")

```

← Counts the number of parameters

We first set `requires_grad=False` for every parameter, ensuring that none of the existing ResNet50 weights will be updated during training. We replace the final fully connected (fc) layer with a new “head” tailored for our task. Here, we stack several linear layers, activation functions, and dropout for added flexibility and regularization. The final output layer produces two values, logits for “real” and “fake.” The output from listing 13.7 is

```

Number of trainable parameters: 2550502
Number of untrainable parameters: 23508032

```

There are more than 23 million untrainable parameters and only about 2.5 million trainable parameters, which are the parameters in the new output layer.

To determine when to stop training, we’ll look at the performance of the model in the validation set. We’ll stop training if the model doesn’t improve for three consecutive epochs. We define a `train_epoch()` function to train the model for one epoch in the following listing.

#### Listing 13.8 Defining the `train_epoch()` function

```

from tqdm import tqdm

def train_epoch(model,dataloader,loss_fn,optimizer):
    model.train()
    train_loss = 0
    for batch, (X, y) in enumerate(tqdm(dataloader)):
        X, y = X.to(device), y.to(device)
        y_pred = model(X)
        loss = loss_fn(y_pred, y)
        train_loss += loss.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return train_loss/len(dataloader)

```

← Defines the `train_epoch()` function

← Iterates through all batches in the training dataset

← Makes predictions using the model

← Calculates the loss by comparing predictions with the ground truth

← Backpropagates to minimize the loss

The `train_epoch()` function trains the model for one epoch and calculates the training loss. We define a `val_epoch()` function to evaluate the model using the validation data:

```

def val_epoch(model,dataloader,loss_fn):
    model.eval()
    val_loss=0
    with torch.inference_mode():
        for batch, (X, y) in enumerate(dataloader):
            X, y = X.to(device), y.to(device)
            pred_logits = model(X)

```

← Defines the `val_epoch()` function

← Iterates through all batches in the validation dataset

```

        loss = loss_fn(pred_logits, y)
        val_loss += loss.item()
    return val_loss/len(dataloader)

```

← Returns the average loss in the epoch

The `val_epoch()` function evaluates the performance of the model using validation data. The function returns the average loss in each batch in the validation dataset. This helps us gauge generalization and watch for overfitting. We set up the loss function and the optimizer as follows:

```

loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(params=model.parameters(),
                               lr=0.001)

```

To prevent overfitting and wasted computation, we implement early stopping. The training loop stops if the model's performance on the validation set doesn't improve for three consecutive epochs, and we retain the best weights seen during training:

```

class EarlyStop:
    def __init__(self, patience=3):
        self.patience = patience
        self.steps = 0
        self.min_loss = float('inf')
    def stop(self, val_loss):
        if val_loss < self.min_loss:
            self.min_loss = val_loss
            self.steps = 0
        elif val_loss >= self.min_loss:
            self.steps += 1
        if self.steps >= self.patience:
            return True
        else:
            return False
stopper=EarlyStop()

```

← If the validation loss is smaller than the previously achieved minimum, keeps training

← If the loss in the validation set is larger than the minimum, adds 1 to the counter

← If the performance doesn't improve in three consecutive epochs, stops training

← Instantiates the EarlyStop() class

With this in place, we start the training process:

```

for i in range(100):
    trainloss=train_epoch(model,train_data_loader,
                           loss_fn,optimizer)
    valloss=val_epoch(model,val_data_loader,loss_fn)
    print(f'epoch {i},trainloss={trainloss},valloss={valloss}')
    if stopper.stop(valloss)==True:
        break

```

← Fine-tunes the model for a maximum of 100 epochs

← Calculates the training loss using the train\_epoch() function

← Calculates the validation loss using the val\_epoch() function

← Stops training if the performance stops improving in the validation set

The training takes about an hour. The training stops after six epochs. The best model is saved after three epochs. If you don't have GPU training, you can go to <https://mng.bz/oZBD> to download the fine-tuned model parameters.



### 13.4.3 Detecting deepfakes using the fine-tuned ResNet50

With training complete and our fine-tuned model ready, it's time for the most exciting step: seeing how well our ResNet50 distinguishes between real and AI-generated faces on never-before-seen data. This subsection demonstrates how to load the best-performing weights, generate predictions on the test set, visualize results, and evaluate the model's overall performance.

If you trained the model yourself, you can directly use your saved weights. Alternatively, you can download a set of pretrained weights, fine-tuned for this task, from the open source community. This ensures you can test the model's abilities even without a GPU.

#### Listing 13.9 Loading fine-tuned weights

```
import requests, os

url=("https://github.com/khanaabidabdal/"
     "Ai-Image-Classfier-Real-Vs-Fake-Faces/"
     "raw/main/RealityCheck.pth")

file="files/RealityCheck.pth"

if not os.path.exists(file):
    print("downloading fine-tuned weights")
    fb=requests.get(url)
    with open(file,"wb") as f:
        f.write(fb.content)
else:
    print("weights have already been downloaded")
model.load_state_dict(torch.load(file,
                                map_location=device))
```

URL to download the fine-tuned weights

Where to save the weights locally

Downloads the fine-tuned weights if not already downloaded

Loads the fine-tuned weights to the model

Now let's put our fine-tuned model to work on a batch of test images. We'll randomly sample a batch and generate predictions for each image:

```
torch.manual_seed(42)
imgs, ys = next(iter(test_data_loader))
```

The tensor `imgs` has 32 images in it. The tensor `ys` has 32 values, and each value is either 0 (the image is fake) or 1 (the image is real). Now we use the fine-tuned model to make predictions on the 32 images:

```
with torch.inference_mode():
    X, y = imgs.to(device), ys.to(device)
    logits = model(X)
    preds = logits.argmax(dim=1)
```

We print out the values of `ys`, which are ground truth, and `preds`, which are predictions, and see if each prediction is correct:

```
print(ys)
print(preds)
print(ys==preds.cpu())
```

The output is

```
tensor([0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1,
        1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0])
tensor([0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1,
        1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0],
        device='cuda:0')
tensor([True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True])
```

Results show that the model made 32 correct predictions and no wrong predictions.

### Exercise 13.4

Change the random seed to 0, and obtain a batch of 32 images from the test subset. Make predictions on them using the fine-tuned model, and see how many predictions are correct.

A powerful way to assess your model's real-world performance is to visualize a sample of predictions, including both successes and errors. Let's plot the first 10 images from the test batch, clearly labeling each one with both its ground-truth and predicted class:

```
import torchvision.transforms as T

T1=T.Compose([
    T.Normalize([0, 0, 0],
                [1/0.229, 1/0.224, 1/0.225]))

T2=T.Compose([
    T.Normalize([-0.485, -0.456, -0.406],
                [1,1,1]))

plt.figure(figsize=(10,4),dpi=100)
for i in range(10):
    ax = plt.subplot(2,5, i + 1)
    plt.imshow(T2(T1(imgs[i])).permute(1,2,0))
    plt.xticks([])
    plt.yticks([])
    if preds[i]==0 and ys[i]==0:
        plt.title("fake image\nprediction: fake",color="green")
    elif preds[i]==1 and ys[i]==0:
        plt.title("fake image\nprediction: real",color="red")
```

```

elif preds[i]==0 and ys[i]==1:
    plt.title("real image\nprediction: fake",color="red")
elif preds[i]==1 and ys[i]==1:
    plt.title("real image\nprediction: real",color="green")
plt.tight_layout()
plt.show()

```

You'll see the output in figure 13.6. Each subplot shows one test image with its true class and model prediction, letting you visually inspect where the model makes the correct prediction or not.



**Figure 13.6** Using the fine-tuned model to differentiate real images from deepfakes. Each image is labeled with both its ground-truth class (real or fake) and the model's prediction. This side-by-side visualization provides immediate feedback on the model's predictions. Notice how convincingly realistic some fake images appear, underscoring the importance of robust deep learning solutions.

For a quantitative summary, let's calculate the model's overall prediction accuracy across the entire test set, comprising 10,000 real and 10,000 fake images (because the classes are balanced, accuracy is a reliable metric):

```

model.eval()
test_acc = 0
with torch.inference_mode():
    for batch, (X, y) in enumerate(test_dataloader):
        X, y = X.to(device), y.to(device)
        test_pred_logits = model(X)
        test_pred_labels = test_pred_logits.argmax(dim=1)
        test_acc += ((test_pred_labels == y).sum(
            ).item()/len(test_pred_labels))
test_acc = test_acc / len(test_dataloader)
print(f"the prediction accuracy is {test_acc:.4f}")

```

This script computes and averages accuracy over all batches in the test set. The output is

```
the prediction accuracy is 0.9373
```

This means the fine-tuned ResNet50 correctly classified 93.73% of all test images, a testament to the effectiveness of transfer learning for deepfake detection.

## Summary

- The DALL-E series of models harnesses the combined strengths of transformer architectures and advanced image-generation techniques. Each successive version of DALL-E has pushed the boundaries of what's achievable in generating images from textual descriptions.
- Imagen combines pretrained language models with advanced cascaded diffusion architectures, generating remarkably precise, high-resolution images directly from sophisticated and nuanced textual descriptions.
- Latent diffusion models such as Stable Diffusion have emerged as a powerful paradigm in text-to-image generation by substantially reducing computational costs while maintaining visual quality and semantic fidelity. Midjourney extends the core capabilities of Stable Diffusion into user-friendly, creatively empowering applications.
- Despite remarkable progress, state-of-the-art text-to-image generation models still have many challenges and concerns. These include geometric inconsistency and the societal risks posed by misinformation, bias, and copyright concerns.
- Training from scratch involves building a model with randomly initialized parameters, requiring the network to learn all features from raw data. In contrast, fine-tuning uses the architecture and pretrained weights of an existing model, modifying (often only) the output layer for a downstream task. The bulk of the network retains the learned parameters, while only the final layers are updated for the new objective.
- Fine-tuning a neural network requires high-quality, well-labeled data that closely matches your target task. For our deepfake detection problem, this means gathering and organizing a large number of images, clearly labeled as either “real” or “fake.” The more representative and diverse the dataset, the better the model will be at generalizing to new, unseen images.
- The fine-tuned ResNet50 model performs well on unseen, out-of-sample images, achieving an accuracy of 93.73% among a test dataset.

# *appendix*

## *Installing PyTorch and enabling GPU training locally and in Colab*

---

My preferred way of installing Python and managing libraries and packages on your computer is through Anaconda, an open source Python distribution, package manager, and environment management tool. Anaconda is user-friendly and can help you effortlessly install numerous libraries and packages, which could be a pain to install otherwise. Anaconda allows you to install packages through both `conda install` and `pip install`, broadening the spectrum of available resources.

I'll also walk you through the steps in creating a dedicated Python virtual environment for all the projects in this book. This segregation ensures that the libraries and packages used in this book remain isolated from any libraries used in other, unrelated projects, thus eliminating any potential interference. However, you can choose your own way of installing Python and creating a virtual environment on your computer for projects in this book.

We'll use Jupyter Notebook as our integrated development environment (IDE). I'll guide you through the installation of Jupyter Notebook in the Python virtual environment you just created. You'll also learn to check if your computer is equipped with a Compute Unified Device Architecture (CUDA)-enabled GPU. If yes, you'll learn to install PyTorch, TorchVision, and TorchAudio on your computer.

If you don't have a CUDA-enabled GPU on your computer, you're encouraged to train models in the Google Colab for free. Because many models in this book contain hundreds of millions of parameters, running these models may become impractically slow without GPU acceleration. At the end of this appendix, I'll discuss how

to train models with a GPU in Google Colab. Don't just copy and paste the book's code snippets into a new Colab notebook and run them. Instead, use the Colab notebooks I've provided, as Google Colab has a unique filesystem that requires specific setup.

## **A.1** *Installing Python and setting up a virtual environment*

In this section, I'll guide you through the process of installing Anaconda on your computer based on your operating system. After that, you'll create a Python virtual environment for all projects in this book. Finally, you'll install Jupyter Notebook as your IDE to run Python programs in this book.

### **A.1.1** *Installing Anaconda*

To install Python through the Anaconda distribution, follow the steps outlined in this subsection. First, go to the official Anaconda website ([www.anaconda.com/download/success](http://www.anaconda.com/download/success)), and scroll to the bottom of the webpage. Locate and download the most recent Python 3 version tailored to your specific operating system (Windows, macOS, or Linux).

If you're using Windows, download the latest Python 3 graphical installer from the same link. Click on the installer, and follow the instructions to install. For macOS users, the latest Python 3 graphical installer for Mac is recommended, although a command-line installer option is also available. The installation process for Linux is more complex than for other operating systems, as there is no graphical installer. Begin by identifying the appropriate package. Click to download the latest installer bash script. The installer bash script is typically saved to your computer's Downloads folder by default. Install Anaconda by executing the bash script within a terminal.

After installation, the setup may add initialization code to your shell configuration file (e.g., `~/.bashrc`, `~/.zshrc`, or another depending on your shell). To activate the changes, reload your shell configuration. For example, for bash, use

```
source ~/.bashrc
```

or for zsh, use

```
source ~/.zshrc
```

To access Anaconda Navigator, enter the following command in a terminal:

```
anaconda-navigator
```

If you can successfully launch the Anaconda Navigator on your computer, your installation of Anaconda is complete.

#### **Exercise A.1**

Install an Anaconda on your computer based on your operating system. After that, open the Anaconda Navigator app on your computer to confirm the installation.

### A.1.2 Setting up a Python virtual environment

It's highly recommended that you create a separate virtual environment for this book. Let's name the virtual environment `txt2img`. Execute the following command in the Anaconda prompt (Windows) or a terminal (Mac and Linux):

```
conda create -n txt2img
```

After pressing the Enter key on your keyboard, follow the instructions on the screen, and press Y when you see the Y/N prompt. To activate the virtual environment, run the following command in the same Anaconda prompt (Windows) or terminal (Mac and Linux):

```
conda activate txt2img
```

The virtual environment isolates the Python packages and libraries that you use for this book from other packages and libraries that you use for other purposes. This prevents any undesired interference.

#### Exercise A.2

Create Python virtual environment `txt2img` on your computer. After that, activate the virtual environment.

### A.1.3 Installing Jupyter Notebook

To install Jupyter Notebook in the virtual environment (make sure the virtual environment is activated), run the following command, and then follow the instructions to install the app:

```
conda install notebook
```

To launch Jupyter Notebook, execute the following command in the same Anaconda prompt or terminal with the virtual environment activated, and the Jupyter Notebook will open in your default browser:

```
jupyter notebook
```

#### Exercise A.3

Install a Jupyter Notebook in Python virtual environment `txt2img`. After that, open the Jupyter Notebook app on your computer to confirm the installation.

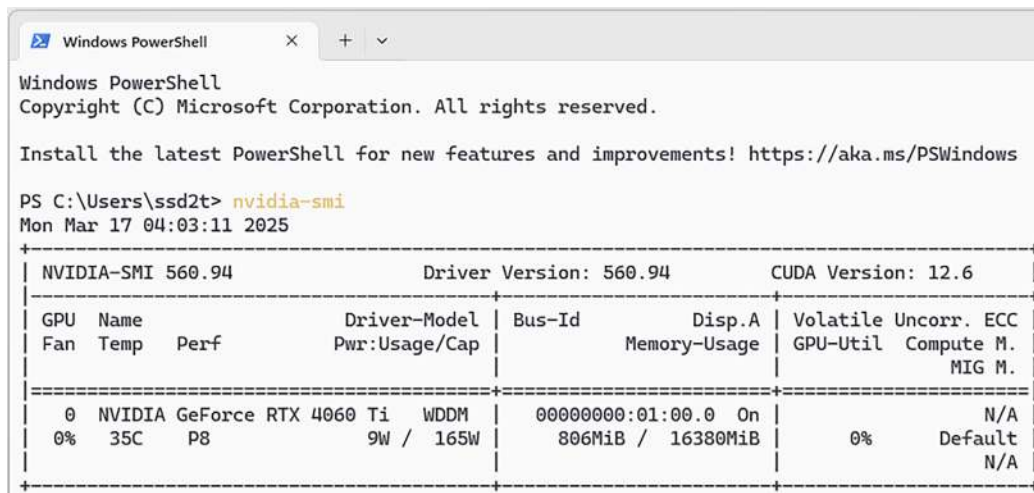
## A.2 Installing PyTorch

In this section, I'll guide you through the installation of PyTorch, assuming you have a CUDA-enabled GPU on your computer. The official PyTorch website (<https://pytorch.org/get-started/locally/>) provides updates on PyTorch installation with or without CUDA. I encourage you to check the website for any updates.

CUDA is only available on Windows or Linux, not on Mac. To find out if your computer is equipped with a CUDA-enabled GPU, open the Windows PowerShell (in Windows) or a terminal (in Linux), and issue the following command:

```
nvidia-smi
```

If your computer has a CUDA-enabled GPU, you should see output similar to figure A.1. Further, make a note of the CUDA version as shown at the top-right corner of the figure because you'll need this piece of information later when you install PyTorch. Figure A.1 shows that the CUDA version is 12.6 on my computer. The version may be different on yours.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\ssd2t> nvidia-smi
Mon Mar 17 04:03:11 2025
```

NVIDIA-SMI 560.94				Driver Version: 560.94		CUDA Version: 12.6	
GPU	Name	Perf	Driver-Model	Bus-Id	Disp.A	Volatile	Uncorr. ECC
Fan	Temp		Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M. MIG M.
0	NVIDIA GeForce RTX 4060 Ti	P8	WDDM 9W / 165W	00000000:01:00.0	On	0%	N/A
				806MiB / 16380MiB			Default N/A

**Figure A.1** Checking to see if your computer has a CUDA-enabled GPU

If you see an error message after running the command `nvidia-smi`, your computer doesn't have a CUDA-enabled GPU. In this case, I encourage you to follow the instructions in the next section to run all programs in this book in Google Colab for free. In either case, I'll provide you with the pretrained models so that you can witness text-to-image generative in action.

To install PyTorch with CUDA, first find out the CUDA version of your GPU, as shown at the top-right corner of figure A.1. I'll use my CUDA 12.6 version as an example



in the following installation. If you go to the PyTorch website (<https://pytorch.org/get-started/locally/>), you'll see an interactive interface, as shown in figure A.2.

**NOTE:** Latest PyTorch requires Python 3.9 or later.

PyTorch Build	Stable (2.6.0)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python		C++ / Java	
Compute Platform	CUDA 11.8	CUDA 12.4	CUDA 12.6	ROCm 6.2.4
Run this Command:	<pre>pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu126</pre>			

**Figure A.2** The interactive interface for installing PyTorch

Once there, choose your operating system, and select Pip as the Package, Python as the Language, and the closest Computer Platform based on what you've determined in the previous step. The command you need to run will be shown in the Run This Command panel at the bottom. For example, I'm using the Windows operating system, and I have CUDA 12.6 on my GPU. Therefore, the command for me is shown at the bottom panel of figure A.2.

Once you know what command to run to install PyTorch with CUDA, activate the virtual environment by running the following line of code in the Anaconda prompt (Windows) or a terminal (Linux):

```
conda activate txt2img
```

Then issue the line of command you've found in the preceding step. For me, the command line is

```
pip3 install torch torchvision torchaudio
--index-url https://download.pytorch.org/whl/cu126
```

Follow the onscreen instructions to finish the installation. Here, we install three libraries together: PyTorch, TorchAudio, and TorchVision. TorchAudio is a library to process audio and signals, while the TorchVision library is used to process images.

To make sure you have PyTorch correctly installed, run the following lines of code in a new cell in Jupyter Notebook:

```
import torch, torchvision, torchaudio

print(torch.__version__)
print(torchvision.__version__)
print(torchaudio.__version__)
device="cuda" if torch.cuda.is_available() else "cpu"
print(device)
```

The output is as follows on my computer:

```
2.6.0+cu126
0.21.0+cu126
2.6.0+cu126
cuda
```

The last line of the output, `cuda`, indicates that I've installed PyTorch with CUDA.

#### Exercise A.4

If you have a CUDA-enabled GPU on your computer, install PyTorch, TorchVision, and TorchAudio on your computer based on your operating system. After that, print out the versions of the three libraries you just installed.

If you have a Mac, you can potentially use the new Metal Performance Shaders backend for GPU training acceleration. More information is available at <https://developer.apple.com/metal/pytorch/> and <https://pytorch.org/get-started/locally/>. Throughout the book, you can use the following command to define the PyTorch device:

```
device="mps" if torch.backends.mps.is_available() else "cpu"
```

If the value of the PyTorch device is `mps`, it means you can use your Mac for GPU training acceleration.

### A.3 *Using Google Colab for GPU training and inference*

You can train and use PyTorch models in this book in Google Colab for free. To get started, log in to your Google account, and go to <https://colab.research.google.com/>. Start a new notebook in Google Colab, click on the Runtime menu, and select Change Runtime Type from the drop-down list, as shown in figure A.3.

After selecting Change Runtime Type from the drop-down list, a new window will pop up, as shown in figure A.4.

In the newly opened window, select Python 3 as the runtime type and T4 GPU as the hardware accelerator. The T4 GPU (Tesla T4) is a powerful and flexible GPU optimized for machine learning training and inference. If you have a Google account, you can

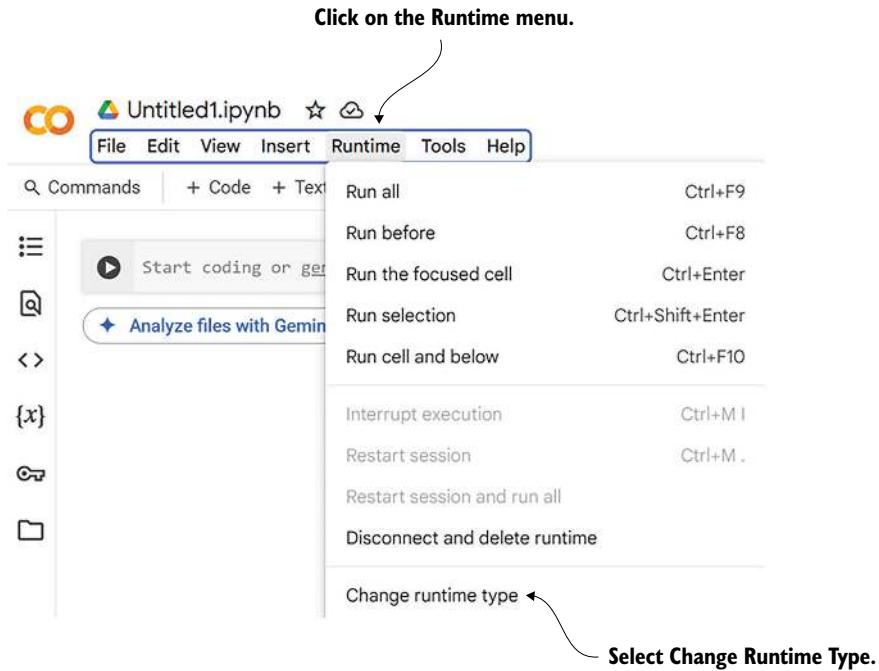


Figure A.3 Changing the runtime type in a Google Colab notebook

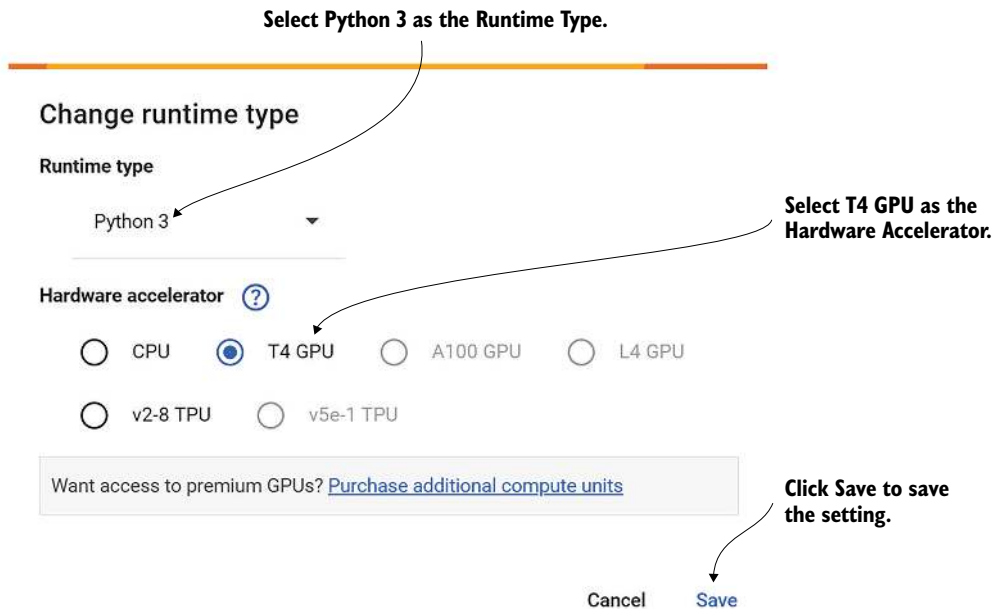


Figure A.4 Selecting the T4 GPU as the hardware accelerator in Google Colab

access a T4 GPU for free in Google Colab to train and run models in this book. Finally, click Save to apply the settings to your notebook.

In Google Colab, there's no need to set up a virtual environment because most required libraries come preinstalled. If you encounter an error indicating a missing library, simply install it using `pip install`, and you'll be ready to proceed.

Python programs from each chapter have been replicated in Google Colab for your convenience. Direct links to Colab notebooks are provided in both the book and individual Jupyter Notebooks in the book's GitHub repository (<https://github.com/markhliu/txt2img>). For instance, you can access the notebook for chapter 2 at <https://mng.bz/nZ6e>.

To get started in any Colab notebook, simply clone the book's GitHub repository using the following commands. This approach eliminates the need to manually upload files or modules. Everything is set up for you automatically:

```
!git clone https://github.com/markhliu/txt2img
import sys
sys.path.append("/content/txt2img")
```

You'll also find these setup instructions included at the beginning of each chapter's Colab notebook. This ensures you can always access the latest code and work seamlessly in the Colab environment.

### **Exercise A.5**

If you don't have a CUDA-enabled GPU on your computer, open a new notebook in Google Colab, and change the setting to use GPU training.

# references

---

## CHAPTER 1

- [1] Rombach, Robin, Blattmann, Andreas, Lorenz, Dominik, Esser, Patrick, and Ommer, Björn. (2022, April 13). High-resolution image synthesis with latent diffusion models (version 2). <https://arxiv.org/abs/2112.10752>
- [2] Chayka, Kyle. (2023, February 10). Is A.I. art stealing from artists? *The New Yorker*. [www.newyorker.com/culture/infinite-scroll/is-ai-art-stealing-from-artists](http://www.newyorker.com/culture/infinite-scroll/is-ai-art-stealing-from-artists)
- [3] Xiang, Chloe. (2023, February 1). AI spits out exact copies of training images, real people, logos, researchers find. *VICE*. [www.vice.com/en/article/ai-spits-out-exact-copies-of-training-images-real-people-logos-researchers-find/](http://www.vice.com/en/article/ai-spits-out-exact-copies-of-training-images-real-people-logos-researchers-find/)
- [4] Skelton, Sebastian Klovig. (2024, June 13). AI's environmental cost could outweigh sustainability benefits. *Computer Weekly*. [www.computerweekly.com/news/366588668/AIs-environmental-cost-could-outweigh-sustainability-benefits](http://www.computerweekly.com/news/366588668/AIs-environmental-cost-could-outweigh-sustainability-benefits)
- [5] Bond, Shannon. (2024, February 8). AI fakes raise election risks as lawmakers and tech companies scramble to catch up. *NPR*. <https://mng.bz/5vqZ>
- [6] Robertson, Adi. (2024, February 21). Google apologizes for “missing the mark” after Gemini generated racially diverse Nazis. *The Verge*. [www.theverge.com/2024/2/21/24079371/google-ai-gemini-generative-inaccurate-historical](http://www.theverge.com/2024/2/21/24079371/google-ai-gemini-generative-inaccurate-historical)

## CHAPTER 2

- [1] Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, et al. (2017, June 12). Attention is all you need. <https://arxiv.org/abs/1706.03762>

## CHAPTER 3

- [1] Dosovitskiy, Alexey, Beyer, Lucas, Kolesnikov, Alexander, et al. (2020). An image is worth  $16 \times 16$  words: Transformers for image recognition at scale. <https://arxiv.org/abs/2010.11929>

- [2] Loshchilov, Ilya, and Hutter, Frank. (2017). Decoupled weight decay regularization. <https://arxiv.org/abs/1711.05101>

## CHAPTER 5

- [1] Sohl-Dickstein, Jascha, Weiss, Eric A., Maheswaranathan, Niru, and Surya Ganguli. (2015). Deep unsupervised learning using nonequilibrium thermodynamics. <https://arxiv.org/abs/1503.03585>; Song, Yang, and Ermon, Stefano. (2019). Generative modeling by estimating gradients of the data distribution. <https://arxiv.org/abs/1907.05600>; and Ho, Jonathan, Jain, Ajay, and Abbeel, Pieter. (2020). Denoising diffusion probabilistic models <https://arxiv.org/abs/2006.11239>
- [2] Ho, Jonathan, Jain, Ajay, and Abbeel, Pieter. (2020). Denoising diffusion probabilistic models <https://arxiv.org/abs/2006.11239>
- [3] Nichol, Alex, and Dhariwal, Prafulla (2021). Improved denoising diffusion probabilistic models. <https://arxiv.org/abs/2102.09672>

## CHAPTER 6

- [1] Ho, Jonathan, and Salimans, Tim. (2022). Classifier-free diffusion guidance. <https://arxiv.org/abs/2207.12598>

## CHAPTER 7

- [1] Song, Jiaming, Meng, Chenlin, and Ermon, Stefano. (2020). Denoising Diffusion Implicit Models. <https://arxiv.org/abs/2010.02502>

## CHAPTER 8

- [1] Radford, Alec, Wook Kim, Jong, Hallacy, Chris, et al. (2021). Learning transferable visual models from natural language supervision. <https://arxiv.org/abs/2103.00020>
- [2] He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. (2016). Deep residual learning for image recognition. <https://doi.org/10.1109/CVPR.2016.90>

## CHAPTER 9

- [1] Rombach, Robin, Blattmann, Andreas, Lorenz, Dominik, Esser, Patrick, and Ommer, Björn. (2022). High-resolution image synthesis with latent diffusion models. <https://arxiv.org/abs/2112.10752>

## CHAPTER 10

- [1] Rombach, Robin, Blattmann, Andreas, Lorenz, Dominik, Esser, Patrick, and Ommer, Björn. (2022). High-resolution image synthesis with latent diffusion models (version 2). <https://arxiv.org/abs/2112.10752>

## CHAPTER 11

- [1] Esser, Patrick, Rombach, Robin, and Ommer, Bjorn. (2021). Taming transformers for high-resolution image synthesis. <https://arxiv.org/abs/2012.09841>
- [2] Kingma, Diederik P., and Welling, Max. (2013). Auto-encoding variational Bayes. <https://arxiv.org/abs/1312.6114>
- [3] van den Oord, Aaron, Vinyals, Oriol, and Kavukcuoglu, Koray. (2017). Neural discrete representation learning. <https://arxiv.org/abs/1711.00937>

- [4] Goodfellow, Ian, Pouget-Abadie, Jean, Mirza, Mehdi, Xu, Bing, Ward-Farley, David, et al. (2014). Generative adversarial networks. <https://arxiv.org/abs/1406.2661>

### CHAPTER 13

- [1] Ramesh, Aditya, Pavlov, Mikhail, Goh, Gabriel, et al. (2021). Zero-shot text-to-image generation. <https://arxiv.org/abs/2102.12092>
- [2] Ramesh, Aditya, Dhariwal, Prafulla, Nichol, Alex, Chu, Casey, and Chen, Mark. (2022). Hierarchical text-conditional image generation with CLIP latents. <https://arxiv.org/abs/2204.06125>
- [3] Betker, James, Goh, Gabriel, Jing, Li, et al. (2023). Improving image generation with better captions. <https://cdn.openai.com/papers/dall-e-3.pdf>
- [4] Saharia, Chitwan, Chan, William, Saxena, Saurabh, et al. (2022). Photorealistic text-to-image diffusion models with deep language understanding. <https://arxiv.org/abs/2205.11487>
- [5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. (2015). Deep residual learning for image recognition. <https://arxiv.org/abs/1512.03385>





## A

AdamW optimizer 57  
AI models 3  
Anaconda, installing 316  
attention mechanism  
    coding 43  
    creating transformers 28–33  
    overview of 24–28  
autoencoders, overview 249  
    VAEs, need for 251

## B

BART (Bidirectional and Auto-Regressive Transformers) 8, 269–270  
    loading pretrained decoder 279  
    predicting image tokens using decoder 280–281  
BERT (bidirectional encoder representations from transformers) 178  
BERT (Bidirectional Encoder Representations from Transformers) 270  
BOS (beginning of sentence) 32, 37, 272  
BPE (byte pair encoding) 274

## C

caption key 186  
captions

    adding to images 81–88, 91–96  
    preprocessing for training 184–187  
cascaded refinement process 292  
causal attention mask, preparing data and 81–84  
CFG (classifier-free guidance) 127, 128–131, 184, 185, 213, 222, 223, 276  
    implementing 129–131  
    overview of 128  
channel-first image format 160  
channel-last image format 160  
CIFAR-10 dataset 57–60  
    downloading and visualizing images 58  
    preparing datasets for training and testing 59  
CLIP (Contrastive Language-Image Pretraining) 23, 175, 176, 177, 202, 203, 226, 291  
    preparing training dataset 180–187  
CLIP (Contrastive Language-Image Pretraining)  
    model 7, 17, 172, 177–180, 232  
    building 190–192  
    creating 187–192  
    guidance by 213–215  
    overview of 177  
    selecting images from Flickr 8k based on text descriptions 178  
    training and using 192–199

CLS (classification) token 63, 68, 73–75, 183, 188–189  
 CNNs (convolutional neural networks) 115, 135, 178  
 codebook 253, 257  
 Colab, enabling GPU training in 315  
 computational costs 294  
 conda install 315  
 conditional denoising model 143  
 converting images to sequences 54–56  
 CUDA (Compute Unified Device Architecture) 315  
 CUDA-enabled GPU (NVIDIA GeForce RTX 20 series or better) 16

## D

DALL-E 17, 268  
   converting image tokens to high-resolution images 282–285  
   iterative prediction of image tokens 278–281  
 DALL-E 2 12  
 Dataloader package 90, 118, 308  
 datasets  
   ImageFolder utility 308  
   library 158, 159  
   package 58, 118, 119, 146  
   training 86–88  
 DDIM (Denoising Diffusion Implicit Model) 153, 161, 166, 168, 236  
 DDPMs (Denoising Diffusion Probabilistic Models) 120, 121, 144, 146, 153  
 decoder 250, 253  
   creating to generate text 94–96  
 deepfake detection, ResNet50  
   blueprint to fine-tune 295–303  
   history and architecture of 296  
   plan to fine-tune for classification 297–299  
   using to classify images 299–303  
 deepfakes, defined 289–290  
*Deep Learning with PyTorch* (Manning) 16  
 denoising  
   diffusion implicit models 156  
   diffusion probabilistic model 144–145  
   process 12  
 denoising U-Net model 131–147  
   architecture 135–137

  building 142–144, 163–165  
   denoising diffusion probabilistic model 144–145  
   down blocks and up blocks 137–142  
   time step embedding and label embedding 132–135  
   training 145–147, 165  
   using to generate images 166  
 detokenization 36, 39  
   converting image tokens to high-resolution images 282–285  
   loading pretrained VQGAN detokenizer 282  
 de\_word\_dict dictionary 38, 39, 41, 49  
 diffusers library 115–117, 120–121, 165, 227–228, 234, 236, 238, 242  
 diffusion models 105, 126  
   classifier-free guidance in 128–131  
   denoising diffusion implicit models 156  
   denoising U-Net model 142–147  
   forward diffusion process 107–115  
   generating high-resolution images with 152–153  
   generating images with trained diffusion model 147–150  
   high-resolution images 158–163  
   image interpolation in 157–158  
   incorporating attention mechanism in U-Net model 154  
   reverse diffusion process 115–117  
   text-to-image generation with 11–15  
   training and using 117–125  
   U-Net for high-resolution images 163–166  
 discriminator, defined 256, 257  
 DistilBERT model 178, 183  
 dividing images into patches 61–63

## E

encoder 250, 253  
 encoder–decoder transformer 17, 43–47  
   coding attention mechanism 43  
   creating language translator 46  
   defining Transformer() class 44–46  
   training 47  
 encoder-only transformer 66–67  
 encoding text prompts 273  
 EOS (end of sentence) 37  
 EOS token 33

**F**

fair use doctrine 294  
 Fashion MNIST (Fashion Modified National Institute of Standards and Technology) 106  
 fc (fully connected) layer 309  
 final high-resolution output 292  
 fixed positional encodings 55  
 Flickr 8k dataset 180–182  
     selecting images from based on text descriptions 178  
 Flickr 8k images, downloading and visualizing 87  
 float16 data type 278  
 float16 half-precision floating-point number 228  
 float32 data type 228  
 forward diffusion process 11, 12, 106–115  
     applying on flower images 161  
     different diffusion schedules 111–115  
     how diffusion models work 107  
     visualizing forward diffusion process 109–110

**G**

GANs (generative adversarial networks) 246, 285  
 GELU (Gaussian error linear unit) 66, 133  
 generative AI (artificial intelligence) 3  
 generator, defined 256  
 geometric inconsistency problem 293  
 GIF (graphics interchange format) 285  
 Google Colab, using for GPU training and inference 322  
 Google's Imagen 291  
 GPTs (Generative Pretrained Transformers) 270  
 GPU (graphics processing unit)  
     training locally and in Colab, enabling 315  
     using Google Colab for training and inference 322

**H**

high-resolution images 282–285  
     converting latent images to 238–242  
     generating and interpolating 166–172  
     generating with diffusion models 152–153  
     training data 158–163  
     U-Net for 163–166

    visualizing intermediate and final outputs 283–285

Hugging Face 226

**I**

IDE (integrated development environment) 315  
 image captioning 79, 96–101  
     adding captions to images 96–101  
     creating multimodal transformer 91–96  
     encoder-decoder transformer 97–99  
     training and using image-to-text transformer 96–101  
 image–caption pairs 180–182  
 image classification  
     building ViT from scratch 60–68  
     converting images to sequences 54–56  
     training and using ViT to classify images 69–77  
     training ViT 54–57  
 image embedding 55  
     prediction 270  
 image encoders, creating 189  
 image generation  
     converting image tokens to high-resolution images 282–285  
     high-resolution images 282–285  
     in latent space 236–238  
     min-DALL-E 270–273  
     with diffusion models 154, 156–158  
     with Stable Diffusion 227–230  
 image generation and interpolation 166–172  
     transitioning from one image to another 168–172  
     using trained denoising U-Net to generate images 166  
 image interpolation in diffusion models 157–158  
 image key 186  
 Imagen 12  
 image reconstruction 271  
 images  
     adding captions to 81–87, 91–96  
     converting into sequences of integers and back 9, 247–249  
     downloading 87  
     guidance parameter affecting generated images 149  
     preprocessing for training 184–187  
     visualizing generated images 148

image selection task 7

image sequences, converting images to 54–56

image tokenization 290

image tokens, iterative prediction of 278–281

- loading pretrained BART decoder 279
- predicting image tokens using BART decoder 280–281

inference, min-DALL-E, image generation at 272

input embedding 41, 56

installing

- Anaconda 316
- Jupyter Notebook 317
- Python 316
- PyTorch 315, 316
- virtual environment 316

integers, converting image into sequence of 9

intellectual property rights 294

## J

---

joint embedding space 290

JSON (JavaScript Object Notation) 87

Jupyter Notebook, installing 317

## K

---

kagglehub library 295, 305

KL divergence (Kullback-Leibler divergence) 251

## L

---

label embedding 132–135

label smoothing 47

LAION-5B dataset 226

LAION-400M dataset 226

language translators 46

latent diffusion 201, 213–219

- converting latent images to high-resolution ones 216–219, 238–242
- diffusion in latent space 215–216
- guidance by CLIP model 213–215
- text-to-image generation with 207–212, 219–223

latent space 202

- image generation in 236–238

LDMs (latent diffusion models) 13, 17, 128, 202, 226, 230, 249

- combining with VAEs 205–207
- overview of 203–207

- text-to-image generation with, modifying existing images with text prompts 219–223
- variational autoencoders (VAEs) 203–205

learned positional encodings 55

LMS (Linear Multi-Step) 236

local training, enabling GPU 315

low-resolution image generation 292

LPIPS (Learned Perceptual Image Patch Similarity) 257

## M

---

matplotlib library 58, 59, 160

Midjourney 292

min-DALL-E 17

- image generation at inference time 272
- iterative prediction of image tokens 278–281
- overview of 270–273
- training 270

misuse of output 294

modeling positions of different patches in an image 63–64

MS COCO (Microsoft Common Objects in Context) dataset 86

multi-head attention 28, 44, 64–66

multi-head self-attention mechanism 64

multimodal models 5

multimodal transformer, creating 91–96

- creating decoder to generate text 94–96
- defining ViT as image encoder 91–94

## N

---

NLP (natural language processing) 24, 57, 164, 293

noise scheduler 120, 122

## O

---

OpenAI, pretrained CLIP model 197–199

optimizer, choosing 69

## P

---

PAD token 37, 183

pandas library 180

perceptual loss 257

PIL (Python Imaging Library) 283

pipeline module 220, 221, 223

pip install 315, 322

positional encoding 26, 33–43, 55  
   input embedding from word embedding and 41  
   sequence padding function 39–41  
   word tokenization with Spacy library 34–39  
 predicting image tokens using BART decoder 280–281  
 preprocess, defined 308  
 preprocessing, training data 118–119  
 Python  
   installing 316  
   setting up virtual environment 317  
 PyTorch 16  
   installing 315, 316, 318–320  
   using Google Colab for GPU training and inference 322

## Q

quantization 259  
 quantization loss 255

## R

reconstruction loss 255  
 ReLU (rectified linear unit) 66  
 requests library 208  
 ResNet50 178  
   blueprint to fine-tune 295–303  
   detecting deepfakes using fine-tuned ResNet50 311–314  
   detecting fake images 304–314  
   downloading and preprocessing real and fake face images 305–308  
   fine-tuning 308–310  
   history and architecture of 296  
   plan to fine-tune for classification 297–299  
   using to classify images 299–303  
 ResNet (Residual Network) 296  
 reverse diffusion process 12, 10, 115–117

## S

SDPA (scaled dot-product attention) 24, 26, 156  
 SD (Stable Diffusion) 12–15, 225, 292  
   architecture of 230–234  
   converting latent images to high-resolution ones 238–242  
   generating images from text with 231  
   generating images with 227–230

  image generation in latent space 236–238  
   text embedding interpolation 232  
   text embeddings 234–235  
 self-attention, defined 26  
 SEP token 183  
 sequences, converting image into sequence of integers 9  
 Spacy library 34–39  
 src input 41  
 StableDiffusionPipeline package 15, 227–229, 234, 242  
 steps counter 71  
 stride 91, 137  
 sys library 248, 275

## T

TensorFlow 16  
 text, generating images from text with Stable Diffusion 231  
 text embeddings 234–235  
   interpolation 232  
 text encoders  
   creating 187–189  
 text encoding 270  
 text prompts, tokenizing and encoding 273–278  
 text-to-image generation 289  
   challenges and concerns 293–295  
   compressing and reconstructing images with VAEs 207–212  
   modifying existing images with text prompts 219–223  
   ResNet50 304–314  
   transformer-based 7–11  
   with diffusion models 11–15  
   with latent diffusion 201, 213–219  
   with LDMs (latent diffusion models) 203–207  
 text-to-image generation models 5–7  
   building from scratch 16–18  
   challenges for 18–19  
   practical use cases of 6  
   social, environmental, and ethical concerns 19  
   unimodal vs. multimodal models 5  
 text-to-image generators  
   generating images with Stable Diffusion 227–230  
   models 3  
   Stable Diffusion 225

- state-of-the-art 290–293
- textual embedding generation 292
- time step embedding 132–135
- timm library 189
- tokenization, defined 25
- tokenizing, text prompts 273–278
- tokens 246
  - building vocabulary of 88
- torch.amp package 69, 97, 122
- torchvision library 58, 59, 299
- training
  - denoising U-Net model 142
  - diffusion models 120–125
  - GPU, enabling locally and in Colab 315
  - U-Net for high-resolution images 163–166
  - U-Net model 117–119
  - VAEs 251
- training data, high-resolution flower images
  - as 158–163
  - applying forward diffusion on flower images 161
  - visualizing images in training dataset 159–160
- transformers 22
  - attention mechanism 24–28
  - creating 28–33
  - encoder-decoder transformers 43–47
  - overview of 24–33
  - training and using German-to-English translator 47–50
  - training and using to add captions 81–86
  - transformer-based text-to-image generation 7–11
  - word embedding and positional encoding 33–43
- transformers library 234
- transformer\_util module 40, 42
- transforms package 118, 119, 146
- trg input 41

## U

- unconditional denoising model 143
- U-Net 115
  - building and training for high-resolution images 163–166
- U-Net denoising model 121–125
- U-Net model
  - denoising 131–147
  - incorporating attention mechanism in 154

- training 117–119
- unimodal models 5
- UNK token 37

## V

- VAEs (variational autoencoders) 13, 202, 231, 249–251
  - autoencoders 249
  - combining LDMs with 205–207
  - compressing and reconstructing images with 207–212
  - downloading pretrained VAEs 207–209
  - encoding and decoding images with pretrained VAEs 209–212
  - need for 251
  - overview of 203–205
  - reconstructing images with 207–212
  - training methodology 251
- vector quantization loss 255
- virtual environment, setting up 316
- visualizing
  - forward diffusion process 109–110
  - generated images 148
  - images 87
  - intermediate and final high-resolution outputs 283–285
- ViTs (vision transformers) 4, 52, 80
  - building from scratch 60–68
  - CIFAR-10 dataset 57–60
  - defining as image encoder 91–94
  - training 54–57
  - training and using to classify images 69–77
- vocab dictionary 276
- vocabularies, building 88
- VQGAN (Vector Quantized Generative Adversarial Network) 8, 245, 256–259, 269
  - converting images into sequences of integers and back 247–249
  - detokenizer 282
  - framework 17
  - generative adversarial networks 256
  - pretrained model 259–267
  - variational autoencoders 249–251
  - VQGAN with VQ-VAE generator 257–259
  - VQ-VAEs (Vector Quantized Variational Autoencoders) 252–255
- VQ-VAE generator 203, 246, 252–255, 257, 290

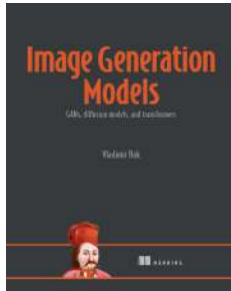
- model architecture and training process
  - 253–255
- need for 252
- VQGAN with VQ-VAE generator 257–259
- VQ-VAE (vector quantized variational autoencoders) 290

## W

---

- word2idx dictionary 88
- word embedding 33–43
  - input embedding from positional encoding and 41
  - sequence padding function 39–41
  - tokenization with Spacy library 34–39
- word tokenization 34–39

## RELATED MANNING TITLES



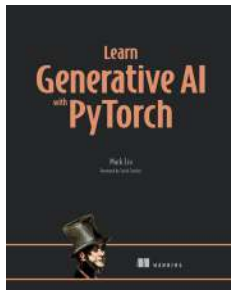
### *Image Generation Models*

By Vladimir Bok

ISBN 9781633437449

350 pages (estimated), \$59.99

January 2026 (estimated)



### *Learn Generative AI with PyTorch*

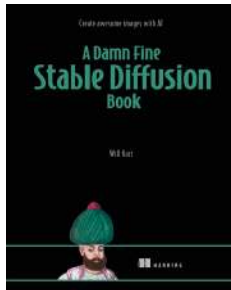
By Mark Liu

Foreword by Sarah Sanders

ISBN 9781633436466

432 pages, \$59.99

October 2024



### *A Damn Fine Stable Diffusion Book*

By Will Kurt

ISBN 9781633436800

275 pages (estimated), \$49.99

Spring 2026 (estimated)



### *Build AI into Your Web Apps*

By Theo Despoudis

ISBN 9781633436084

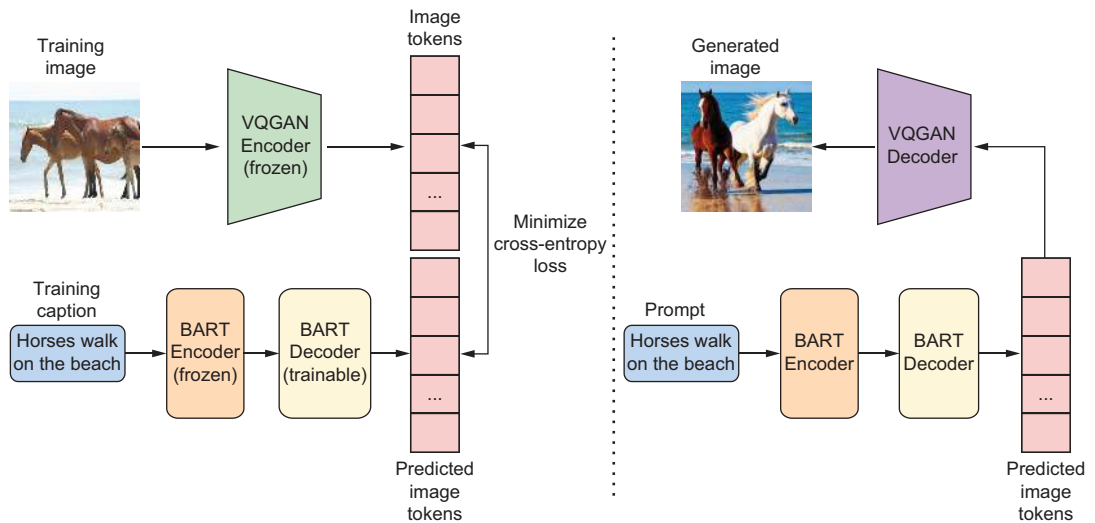
375 pages (estimated), \$59.99

Spring 2026 (estimated)

*For ordering information, go to [www.manning.com](http://www.manning.com)*



## Training and inference workflow of a transformer-based text-to-image generator



**Training pipeline of the DALL-E mini model (left). The image generation process in min-DALL-E to generate an image from a text prompt (right).**

# BUILD A **Text-to-Image Generator** (FROM SCRATCH)

Mark Liu

**A**I-generated images appear everywhere from high-end advertising to casual social media feeds. Text-to-image tools like Dall-e, Midjourney, and Flux make it easy to create AI art, but how do they work? In this book, you'll find out by building your own text-to-image generator!

**Build a Text-to-Image Generator (from Scratch)** explores both transformer-based image generation and diffusion models. You'll work hands-on to build a pair of simple generation models that can classify images, automatically add captions, reconstruct images, and enhance existing graphics. Author Mark Liu guides you every step of the way with clear explanations, informative diagrams, and eye-opening examples you can build on your own laptop.

## What's Inside

- Build a vision transformer to classify images
- Edit images using text prompts
- Fine-tune image models

Requires basic knowledge of generative AI models and intermediate Python skills.

**Mark Liu** is the founding director of the Master of Science in Finance program at the University of Kentucky. He is also the author of *Learn Generative AI with PyTorch*.

For print book owners, all digital formats are free:  
<https://www.manning.com/freebook>

“A practical and readable introduction with working code and clear explanations.”

—Andrey Lukyanenko, Meta

“Empowers you to unlock creativity at the intersection of text and imagery.”

—Bojan Tunguz, Tabul.AI

“Amazingly comprehensive, hype-free, hands-on, and code-rich guidebook.”

—Kirk Borne  
Data Leadership Group

“Successfully brings together the theoretical foundations and practical applications, from transformers to diffusion models.”

—Raymond Cheung  
Parity Technologies



**FREE  
eBook**

see first page

ISBN-13: 978-1-63343-542-1



9 781633 435421

90000