

# ChatGPT

## Principles and Architecture

Ge Cheng



机械工业出版社  
CHINA MACHINE PRESS

# ChatGPT

---

## Principles and Architecture

Ge Cheng

*Hunan National Applied Mathematics Center, Hunan, P.R.  
China*

*School of Mathematics and Computational Science,  
Xiangtan University, Hunan, P.R. China*



---

# Table of Contents

---

**Cover image**

**Title page**

**Copyright**

**Preface**

**Main Content of the Book**

**Target Audience for This Book**

**Contact the Author**

**Acknowledgments**

**Chapter 1. A new milestone in artificial intelligence—  
ChatGPT**

**Abstract**

**1.1 The development history of ChatGPT**

**1.2 The capability level of ChatGPT**

### **1.3 The technical evolution of large language models**

### **1.4 The technology stack of large language model**

### **1.5 The impact of large language models**

### **1.6 The challenges of training or deploying large models**

### **1.7 The limitations of large language models**

### **1.8 Summary**

## **Chapter 2. In-depth understanding of the transformer model**

### **Abstract**

### **2.1 Introduction to the transformer model**

### **2.2 Self-attention mechanism**

### **2.3 Multihead attention mechanism**

### **2.4 Feedforward neural network**

### **2.5 Residual connection**

### **2.6 Layer normalization**

### **2.7 Position encoding**



## **2.8 Training and optimization**

## **2.9 Summary**

# **Chapter 3. Generative pretraining**

## **Abstract**

## **3.1 Introduction to generative pretraining**

## **3.2 Generative pretraining model**

## **3.3 The generative pretraining process**

## **3.4 Supervised fine-tuning**

## **3.5 Summary**

# **Chapter 4. Unsupervised multitask and zero-shot learning**

## **Abstract**

## **4.1 Encoder and decoder**

## **4.2 GPT-2**

## **4.3 Unsupervised multitask learning**

## **4.4 The relationship between multitask and zero-shot learning**

## **4.5 The autoregressive generation process of GPT-2**

## **4.6 Summary**

# **Chapter 5. Sparse attention and content-based learning**

## **Abstract**

## **5.1 GPT-3**

## **5.2 The sparse transformer**

## **5.3 Meta-learning and in-context learning**

## **5.4 Bayesian inference of concept distributions**

## **5.5 Thought chains**

## **5.6 Summary**

# **Chapter 6. Pretraining strategies for large language models**

## **Abstract**

## **6.1 Pre-training datasets**

## **6.2 Processing of pretraining data**

## **6.3 Distributed training patterns**

## **6.4 Technical approaches to distributed training**

## **6.5 Examples of training strategies**

## **6.6 Summary**

# **Chapter 7. Proximal policy optimization**

## **Abstract**

## **7.1 Traditional policy gradient methods**

## **7.2 Actor-Critic**

## **7.3 Trust region policy optimization**

## **7.4 Principles of the proximal policy optimization algorithm**

## **7.5 Summary**

# **Chapter 8. Human feedback reinforcement learning**

## **Abstract**

## **8.1 Reinforcement learning in ChatGPT**

## **8.2 InstructGPT training dataset**

## **8.3 Training stages of human feedback reinforcement learning**

## **8.4 Reward modeling algorithms**

## **8.5 PPO in InstructGPT**

## **8.6 Multiturn dialogue capability**

## **8.7 The necessity of human feedback reinforcement learning**

## **8.8 Summary**

# **Chapter 9. Low-resource domain transfer of large language models**

## **Abstract**

## **9.1 Self-instruct**

## **9.2 Constitutional artificial intelligence**

## **9.3 Low-rank adaptation**

## **9.4 Quantization**

## **9.5 SparseGPT**

## **9.6 Case studies**

## **9.7 Summary**

# **Chapter 10. Middleware**

## **Abstract**

## **10.1 LangChain**



## **10.2 AutoGPT**

## **10.3 Competitors in middleware frameworks**

## **10.4 Summary**

# **Chapter 11. The future path of large language models**

## **Abstract**

## **11.1 The path to strong artificial intelligence**

## **11.2 Data resource depletion**

## **11.3 Limitations of autoregressive models**

## **11.4 Embodied intelligence**

## **11.5 Summary**

## **Index**

---

# Copyright

---

Elsevier

Radarweg 29, PO Box 211, 1000 AE Amsterdam,  
Netherlands

125 London Wall, London EC2Y 5AS, United Kingdom  
50 Hampshire Street, 5th Floor, Cambridge, MA 02139,  
United States

Copyright © 2025 China Machine Press Co., Ltd. Published  
by Elsevier Inc. All rights are reserved, including those for  
text and data mining, AI training, and similar technologies.

For accessibility purposes, images in electronic versions of  
this book are accompanied by alt text descriptions provided  
by Elsevier. For more information, see

<https://www.elsevier.com/about/accessibility>.

Publisher's note: Elsevier takes a neutral position with  
respect to territorial disputes or jurisdictional claims in its  
published content, including in maps and institutional  
affiliations.

No part of this publication may be reproduced or  
transmitted in any form or by any means, electronic or

mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: [www.elsevier.com/permissions](http://www.elsevier.com/permissions).

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

### **Notices**

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability

for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

ISBN: 978-0-443-27436-7

For Information on all Elsevier publications visit our website at <https://www.elsevier.com/books-and-journals>

*Publisher:* Mara Conner

*Acquisitions Editor:* Glyn Jones

*Editorial Project Manager:* Naomi Robertson

*Production Project Manager:* Selvaraj Raviraj

*Cover Designer:* Greg Harris

Typeset by MPS Limited, Chennai, India



---

# Preface

---

As a university computer science researcher and a veteran entrepreneur, I was profoundly impressed by experiencing firsthand the logical reasoning capabilities emerging from ChatGPT. Although many celebrate the efficiency enhancements in multimodal content creation brought by generative artificial intelligence (AI), the reasoning abilities displayed by ChatGPT are often underestimated. This capability enables ChatGPT to serve not only as the core of a new generation of human-computer interaction but also as an intelligent agent to build automated and semiautomated workflows. It can even merge with industrial control or robotics fields, thereby triggering profound social changes.

Many underestimate the impact of this transformation. Given the current pace of R&D and commercial application iterations, I expect that this transformation will gradually permeate all aspects of human life and production over the next 3–5 years, greatly enhancing existing productivity and thereby initiating a series of changes. If asked to pinpoint the last era called a “major technological transformation,” many would unhesitatingly refer to the dawn of the internet. This transformation will also reshape business

models related to content production, change existing work methods, and even drive changes in production methods. Of course, this still depends on whether the next generation of large language models can achieve breakthroughs in controllability of content output.

## **Main Content of the Book**

This book is designed to help readers deeply understand ChatGPT and its related technologies. It consists of 11 chapters that comprehensively explore various aspects.

Chapter 1 provides an in-depth analysis of the technological evolution of large language models, supporting technologies, and technology stacks, and discusses their significant impact on society.

Chapter 2 elaborates on the theoretical foundations and main components of the Transformer model, revealing the principles and applications behind these technologies.

Chapter 3 delves into the generative pretraining process and principles of GPT.

Chapter 4 primarily explores technologies such as layer normalization, orthogonal initialization, and reversible tokenization in GPT-2, and provides a detailed analysis of GPT-2 autoregressive generation process.

Chapter 5 introduces GPT-3 sparse attention mechanisms, metalearning, and content-based learning concepts, and discusses the application of Bayesian inference in conceptual distributions.

Chapter 6 details the pretraining datasets and data processing methods for large language models, as well as distributed training models and architectures.

Chapter 7 deeply analyzes the fundamental principles of the proximal policy optimization (PPO) algorithm.

Chapter 8 focuses on the fine-tuning datasets of reinforcement learning with human feedback (RLHF) and the application of PPO in InstructGPT, discussing the capabilities of multiturn dialog and the necessity of human feedback reinforcement learning.

Chapter 9 explores how to transfer large language models to specific domains at low resource costs.

Chapter 10 primarily introduces the middleware technologies involved in the development of large language models.

Chapter 11 predicts and prospects the future development trends of large language models.

## **Target Audience for This Book**

- **Product managers in the AI field:** For product managers looking to incorporate AI features into their products, understanding the basic principles and operational mechanisms of large language models like ChatGPT is crucial. From this book, they can learn about the design philosophies and construction methods of large language models, as well as how to integrate these models into their products. They can also better understand the



performance bottlenecks of their products, which aids in more precise product planning.

- **Researchers in AI-related fields:** For AI researchers, this book can serve as a textbook for a deep understanding of large language models. Whether it's the details of the Transformer model or tips on training and optimizing GPT models, this book provides thorough explanations. More importantly, this book explores some of the cutting-edge research areas, such as human feedback reinforcement learning and bootstrap labeling algorithms.
- **Engineers specializing in large-scale data processing and analysis:** For engineers facing challenges such as efficiently processing large-scale data or building distributed training architectures, this book offers many valuable suggestions and ideas. For example, Chapter 6 delves deeply into data processing and distributed training patterns.
- **AI enthusiasts and technologically savvy individuals in everyday life:** If you are an AI technology enthusiast or someone who uses technology to improve everyday life, this book is also suitable for you. The introduction to large language models in this book is easy to understand, providing a comprehensive overview of this powerful technology. More interestingly, this book offers

many practical usage tips and case studies that can be directly applied to your life or work.

## **Contact the Author**

Given my limited writing skills, there are inevitably some inadequacies in this book. If you have any questions or suggestions during your reading, you can contact me via email at [chenggextu@hotmail.com](mailto:chenggextu@hotmail.com). I greatly look forward to your feedback, as it will be immensely helpful for my future writing. I hope you gain profound insights and deepen your understanding of large language models and AI while reading this book.

---

# Acknowledgments

---

First, I would like to thank my family. During the writing of this book, the time spent with them was greatly reduced, but they always provided support and understanding, allowing me to fully dedicate myself to writing without any concerns.

I am grateful to the editors, Yang Fuchuan and Chen Jie. The smooth publication of this book would not have been possible without their professionalism and meticulous work attitude.

Finally, I must thank my graduate students Yin Zhibin, Luo Qifan, Yu Zhiwen, Yu Jiangnan, and Yang Jin. They created numerous illustrations for this book, and I am sincerely grateful for their contributions!

**Ge Cheng**

---

## CHAPTER 1

# A new milestone in artificial intelligence—ChatGPT

---

### Abstract

This chapter provides an in-depth exploration of the evolution, capabilities, and impact of ChatGPT, a groundbreaking artificial intelligence (AI) application developed by OpenAI. The chapter traces the development history of ChatGPT from its early predecessors like GPT-1 and GPT-2 to the more advanced GPT-3 and GPT-4 models. It highlights the technological advancements that have made ChatGPT a powerful tool capable of complex tasks such as language comprehension, code generation, and multimodal reasoning. The chapter also discusses the architecture underpinning large language models (LLMs), focusing on the transition from traditional natural language processing techniques to Transformer-based models. Additionally, it addresses the significant computational and data requirements

for training these models, alongside challenges such as model interpretability, biases, and privacy concerns. The chapter concludes by examining the broader implications of LLMs across various sectors and anticipates future trends in AI development.

## **Keywords**

ChatGPT; OpenAI; transformer model; natural language processing (NLP); artificial general intelligence (AGI); GPT series; human feedback reinforcement learning (HFRL); large language models (LLMs); computational power; model limitations

In November 2022, OpenAI introduced ChatGPT, an artificial intelligence (AI)-powered chat application with diverse functionalities. This application displays intellectual prowess that rivals and sometimes surpasses human performance across numerous professional and academic measures. Upon its launch, ChatGPT garnered widespread acclaim within the technology sector, marking a significant advancement in AI.

## **1.1 The development history of ChatGPT**

For more than 50 years, the quest to enable computers to communicate like humans has been a significant focus in technology. The general public is acquainted with chat applications, tracing back to 1966 when MIT Professor

Joseph Weizenbaum created ELIZA, the first program of its kind. Over the decades, advancements have been evident from Apple's Siri and Microsoft's Xiaoice to smart speakers and intelligent assistants used in various industries. These AI-driven products have become intricately woven into the fabric of daily life.

These products share a distinctive trait: they are easily recognized as AI, highlighting a noticeable disparity between their conversational abilities and human interaction. Yet, ChatGPT distinguishes itself in this realm. It not only conducts complex, multiturn text dialogues but also possesses the ability to generate code, marketing materials, poetry, business strategies, and film scripts. While it is not without flaws and can occasionally err, its robust capabilities position it as one of the AI tools closest to passing the Turing test.

ChatGPT was developed by OpenAI, a nonprofit AI research institute established in San Francisco in 2015. Its founders include entrepreneur Elon Musk, Sam Altman, president of the American startup incubator Y Combinator, and Peter Thiel, cofounder of the global online payment platform PayPal, among others. Based in California, USA, OpenAI has garnered substantial financial support from several prominent Silicon Valley figures, securing an initial investment of up to 1 billion dollars.

The Transformer model, first introduced by the Google Brain team in their seminal 2017 paper "Attention is All You Need," is now recognized as a pivotal advancement in

AI. While it has not entirely supplanted previous frameworks such as recurrent neural networks (RNNs) and convolutional neural networks, it has shown exceptional efficacy in diverse domains including natural language processing (NLP) and computer vision. This model has become a fundamental architecture in these fields. As depicted in **Fig. 1.1**, OpenAI has extensively pursued NLP research using the Transformer model, which ultimately culminated in the development of ChatGPT.

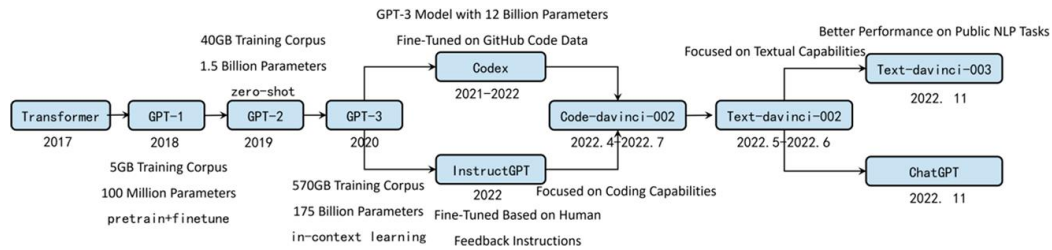


**FIGURE 1.1** GPT series evolution timeline.

The timeline starts with the Transformer model in 2017. In 2018, GPT1 was developed with a 5GB training corpus and 100 million parameters using pretrain and finetune methods. In 2019, GPT2 was introduced with a 40GB training corpus and 1.5 billion parameters, utilizing zero shot learning. GPT3 followed in 2020 with a 570GB training corpus and 175 billion parameters, employing in context learning. In 2021 2022, Codex was fine-tuned on GitHub code data from the GPT3 model with 12 billion parameters. In 2022, InstructGPT was fine-tuned based on human feedback instructions. Codedavinci002, focused on coding capabilities, was developed in April 2022. Textdavinci002, focused on textual capabilities, was developed in May 2022. Textdavinci003, aimed at better performance on public NLP tasks, was developed in November 2022. Finally, ChatGPT was introduced in November 2022.

In 2018 OpenAI introduced its first generative pretrained model, GPT-1. This model significantly advanced the language processing capabilities of AI systems by leveraging large-scale unlabeled data for pretraining. It



also showcased its ability to generalize across multiple tasks through supervised fine-tuning, marking a notable evolution in AI language models.

In March 2019, OpenAI transitioned from a nonprofit organization to a capped-profit entity, unveiling OpenAI LP and introducing Microsoft as a strategic investor. In July of that year, Microsoft made a significant investment of 1 billion dollars in OpenAI, thereby securing commercialization rights to OpenAI's technologies and integrating its developed products with Microsoft's offerings. Concurrently, OpenAI released the GPT-2 model, which utilized a larger dataset called WebText, comprising about 40 GB of text data from 8 million documents. This version increased the number of model parameters to an impressive 1.5 billion, enhancing the model's accuracy. Furthermore, GPT-2 introduced zero-shot learning capabilities, underscoring the effectiveness of unsupervised learning and the success of pretrained models in downstream NLP tasks.

By June 2020, OpenAI had released the GPT-3 language model and launched its first commercial product, the OpenAI application programming interface (API), signifying the organization's official venture into commercial operations. In September of the same year, OpenAI granted Microsoft an exclusive license to utilize its GPT-3 model, positioning Microsoft as the first company globally to adopt GPT-3 technology. Unprecedented in scale, the largest version of GPT-3 boasts 175 billion parameters—117 times

that of GPT-2. Additionally, GPT-3 introduced in-context learning, a method that uses prompts to adapt to a broader range of downstream tasks and displays various emergent abilities, such as chains of thought.

In 2021 Microsoft deepened its collaboration with OpenAI through an additional investment, propelling their partnership into a new phase. In December 2022, supported by Microsoft, OpenAI introduced ChatGPT, which is built on GPT-3.5. This iteration employs reinforcement learning with human feedback, akin to InstructGPT, to better align with human needs.

Remarkably, just 2 months after its release, ChatGPT surpassed 100 million active users globally. At the start of 2023, OpenAI launched the ChatGPT Plus subscription service alongside the GPT-4 model, which possesses multimodal data processing and reasoning capabilities, providing ChatGPT Plus users access to a version of GPT-4 enhanced for NLP tasks.

## **1.2 The capability level of ChatGPT**

In a demonstration video released on March 15, 2023, GPT-4 successfully identified an image of a balloon, as depicted in **Fig. 1.2**, and predicted the outcome of severing the attached string: “The balloon would fly away.” This demonstration underscores GPT-4 proficiency in interpreting and reasoning with multimodal data, such as images. Possessing an extensive array of parameters, GPT-4 transcends mere surface-level image interpretation to

infer deeper significances. This advancement has elicited global astonishment, positioning GPT-4 closer to achieving artificial general intelligence (AGI). Furthermore, this development prompts an ongoing debate regarding the potential of the GPT series to progress towards strong AI, possibly culminating in a capability that surpasses human intelligence.

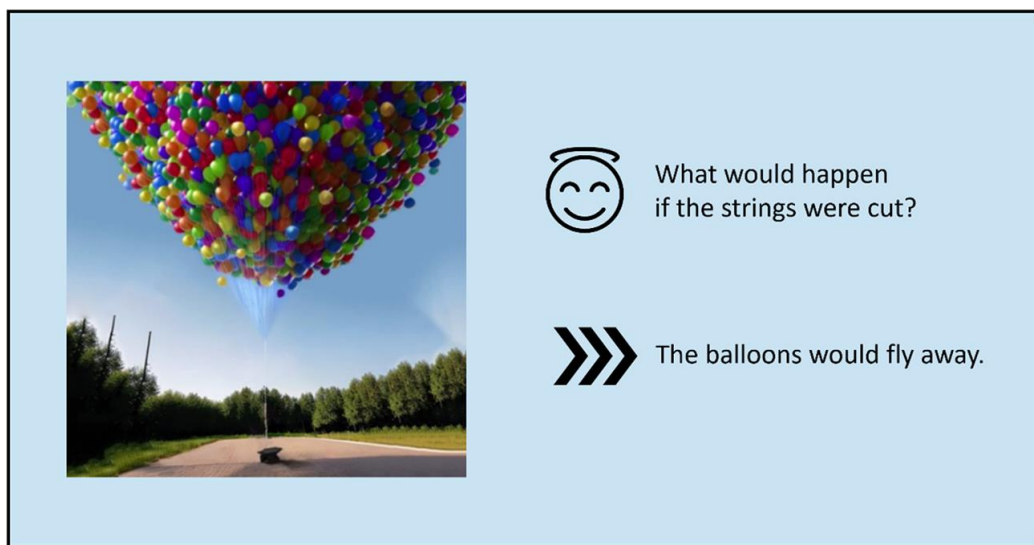


**FIGURE 1.2** GPT-4 balloon image multimodal reasoning demonstration.

The text on the right side of the image reads, What would happen if the strings were cut? followed by the balloons would fly away.

Since the 1970s, scholarly definitions of intelligence have emphasized that intelligence should not be restricted to specific tasks. Over the past few decades, the application of AI has expanded significantly across various domains,

exemplified by AlphaGo's mastery in the game of Go, AlphaFold's protein structure predictions, and the Yolo series' contributions to computer vision. These advancements, however, represent systems that are intelligent within their respective domains. This has led to the development of the concept of AGI, which is distinctly differentiated from the earlier concept of "weak AI." Despite its prominence, both the academic and industrial sectors continue to grapple with establishing a universally accepted definition of AGI.

While the primary function of ChatGPT is conversational interaction, its applications extend far beyond simple dialogue. This AI model is capable of writing and debugging computer programs, emulating the writing styles of well-known figures for the creation of commercial brochures, and generating diverse content such as music, television dramas, fairy tales, student essays, and poetry. Moreover, ChatGPT can emulate Linux operating systems, simulate chatrooms, engage in games like tic-tac-toe, and even mimic the operations of an ATM machine.

As illustrated in **Fig. 1.3**, GPT-4 performance on a range of professional and academic benchmark tests has achieved levels comparable to human capabilities. Notably, GPT-4 placed in the top 10% of candidates in a simulated bar exam and achieved a score of 330 on the graduate record examination (GRE), surpassing the majority of participants. In additional tests, GPT-4 performance exceeded that of other AI systems. These high scores not only reflect

enhanced realistic and logical reasoning skills but also signify advanced problem-solving abilities.

## GPT-4 VS HUMAN TESTS (MAY/2023)

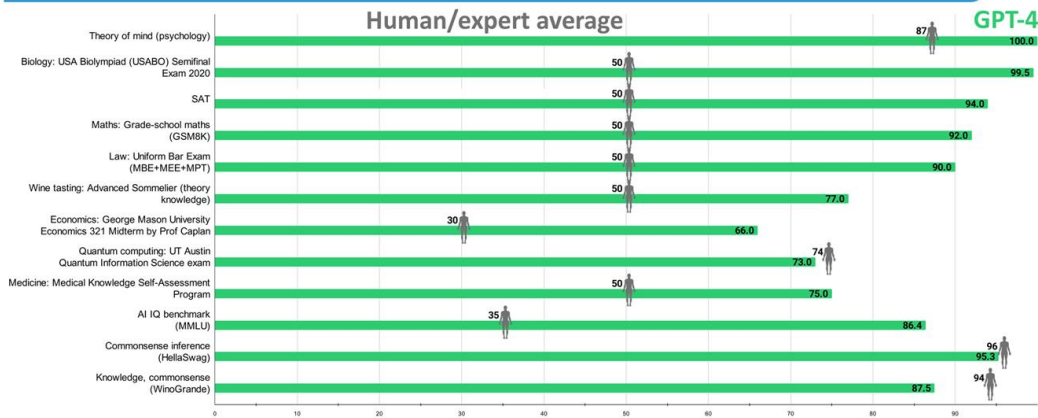


**FIGURE 1.3** Comparison of GPT-4 with human examination levels. Image source: <https://lifearchitector.ai/iq-testing-ai/>.

The horizontal bar graph is titled as GPT4 VS HUMAN TESTS MAY2023. The y axis lists various tests and exams, while the x axis represents scores from 0 to 100. For each test, there are two bars, a bar representing the Human or expert average and a green bar representing GPT4 performance. The tests and their respective scores are, 1. Theory of mind, psychology, Human 87, GPT4 100.0 2. Biology, USA Biolympiad, USABO Semifinal Exam 2020, Human 50, GPT4 99.5 3. SAT, Human 50, GPT4 94.0 4. Maths, Grade school maths, GSM8K, Human 50, GPT4 92.0 5. Law, Uniform Bar Exam, MBEplusMEEplusMPT, Human 50, GPT4 90.0 6. Wine tasting, Advanced Sommelier, theory knowledge, Human 50, GPT4 77.0 7. Economics, George Mason University Economics 321 Midterm by Prof Caplan, Human 30, GPT4 66.0 8. Quantum computing, UT Austin Quantum Information Science exam, Human 74, GPT4 73.0

9. Medicine, Medical Knowledge Self Assessment Program, Human 50, GPT4 75.0 10. AI IQ benchmark, MMLU, Human 35, GPT4 86.4 11. Commonsense inference, HellaSwag, Human 96, GPT4 95.3 12. Knowledge, commonsense, WinoGrande Human 94, GPT4 87.5 In all but two tests, Quantum computing and Commonsense inference, GPT4 outperforms the human or expert average, often by a significant margin. The detailed data points are as follows. For the Theory of mind psychology test, the human expert average was 87, while GPT4 scored 100.0. For the Biology, USA Biolympiad USABO Semifinal Exam 2020 test, the human expert average was 50, while GPT4 scored 99.5. For the SAT test, the human expert average was 50, while GPT4 scored 94.0. For the Maths, Grade school maths GSM8K test, the human expert average was 50, while GPT4 scored 92.0. For the Law, Uniform Bar Exam MBE plus MEE plus MPT test, the human expert average was 50, while GPT4 scored 90.0. For the Wine tasting, Advanced Sommelier theory knowledge test, the human expert average was 50, while GPT4 scored 77.0. For the Economics, George Mason University Economics 321 Midterm by Prof Caplan test, the human expert average was 30, while GPT4 scored 66.0. For the Quantum computing, UT Austin Quantum Information Science exam test, the human expert average was 74, while GPT4 scored 73.0. For the Medicine, Medical Knowledge Self-Assessment Program test, the human/expert average was 50, while GPT4 scored 75.0. For the AI IQ benchmark

MMLU test, the human expert average was 35, while GPT4 scored 86.4. For the Commonsense inference HellaSwag test, the human expert average was 96, while GPT4 scored 95.3. For the Knowledge, commonsense WinoGrande test, the human expert average was 94, while GPT4 scored 87.5.

Although GPT-4 has demonstrated proficiency in a range of exams, assisted in translation and programming tasks, and generated innovative ideas, the methodology for evaluating its intelligence level continues to be a subject of intense debate. Given that GPT-4 has been trained on vast datasets encompassing nearly all information available on the internet, it poses a significant challenge to discern whether it has simply memorized content related to specific tasks or has achieved a true understanding of them. This distinction is crucial for accurately assessing its level of intelligence.

From the perspective of evaluating human intelligence, a key attribute is the understanding of multidomain tasks. In this context, the assessment of GPT-4 intelligence can be examined through several critical aspects:

- **Task performance:** This metric is crucial for evaluating GPT-4 capabilities. Notably, GPT-4 has excelled in various tasks such as exams, translation, programming, and generating innovative ideas, often surpassing the average human performance in many professional and academic tests.



- **Knowledge understanding and application:** It is essential to determine whether GPT-4 genuinely comprehends the knowledge it has acquired and how it applies this knowledge across different contexts. A truly intelligent system should demonstrate the ability to understand and flexibly utilize knowledge in diverse scenarios. GPT-4 has shown remarkable proficiency in applying knowledge to a wide range of activities, including writing and debugging programs, emulating writing styles for commercial brochures, composing music, and scripting television shows.
- **Generalization ability:** An intelligent entity's performance on new tasks and with unseen data—its generalization ability—is a vital measure of its intelligence. Despite potential timeliness issues with its training data, GPT-4 continues to offer innovative solutions to novel problems, showing it does not solely rely on preexisting knowledge and patterns.
- **Autonomous learning and reasoning:** GPT-4 has proven its ability in autonomous learning and reasoning by incorporating new knowledge through prompts and examples. It can independently discover knowledge and formulate reasoning across various tasks and contexts.

Through a comprehensive evaluation of GPT-4 capabilities, encompassing task performance, understanding and application of knowledge,

generalization ability, and autonomous learning and reasoning, it is clear that GPT-4 epitomizes the zenith of current AI research. It stands as the most advanced approximation to AGI thus far. The implications of such advancements are profound, promising to significantly alter human lifestyles in the near future. While the extent of these changes may be underestimated by many, the reality is that the future envisioned by these technological advancements is already unfolding.

## **1.3 The technical evolution of large language models**

### **1.3.1 Symbolism versus connectionism**

One of the fundamental capabilities of ChatGPT is NLP. Initially, NLP utilized symbolic representations to interpret the real world, relying on predefined rules for knowledge-based reasoning. This method primarily involved querying a knowledge base using established rules and executing logical operations for reasoning. However, symbolic approaches often struggle to effectively manage the diversity and ambiguity inherent in language. Furthermore, they exhibit limited robustness and generalization abilities. As linguistic phenomena grow in complexity, the quantity of manually crafted rules increases exponentially, complicating the management of language's diversity and statistical properties.

The advent of machine learning and deep learning heralded a paradigm shift toward a data-driven approach in

NLP technologies. During the machine learning phase, it was feasible to identify optimal mappings within a predefined function space based on training data for specific NLP subtasks. However, this approach did not entirely eliminate the need for rule design. Feature engineering emerged as a crucial determinant of outcomes in machine learning applications, yet the predefined function spaces often exhibited limited robustness and generalization capabilities for specific subtasks.

The introduction of deep learning models catalyzed a transformative development in NLP. Initially, NLP text generation predominantly utilized traditional RNNs, long short-term memory networks (LSTMs), or gated recurrent units (GRUs). These models, based on recurrent architectures, are adept at managing serialized NLP tasks due to their sequential processing capabilities.

RNNs incorporate a memory mechanism to address dependencies within sequence data (whether sequential or temporal), integrating the concept of memory into neural networks and learning contextually relevant patterns through training. Nonetheless, RNNs are challenged by their short memory spans. To overcome this limitation, LSTMs and GRUs were developed. Their internal gating mechanisms manage the information flow, preserving essential data in the sequence while eliminating nonessential elements. This allows significant information to persist in long-sequence processing tasks.

Nevertheless, RNN architectures, including LSTM and GRU, encounter several challenges that compromise their suitability as foundational structures for large language models (LLMs):

1. **Insufficient parallel computing capability:** The inherent need for RNNs to process each time step sequentially in sequence data impedes their ability to fully leverage the parallel computing capabilities of modern GPUs during training. This limitation significantly affects the efficiency of model training.
2. **Long-range dependency issues:** Despite the improvements made by LSTM and GRU over basic RNNs in managing long-range dependencies, they still face difficulties with very long sequences.
3. **Model capacity limitations:** LSTMs and GRUs possess relatively limited model capacities, which restrict the scalability of models in training LLMs. This constraint hinders their ability to capture more complex semantic information and construct intricate representations.

### 1.3.2 Transformer

The GPT series are based the Transformer model, a highly prevalent deep learning framework in NLP tasks, which was introduced by Vaswani et al. in 2017. At the heart of the Transformer model lies the multihead attention mechanism—an enhancement of the self-attention mechanism. This mechanism computes query, key, and

value vectors for each input element within its architecture. It determines the relationships (probabilities) between elements via matrix operations and generates the most probable sequence, thereby effectively capturing long-range dependencies within the input sequence.

In contrast to RNNs, the Transformer model's self-attention mechanism is capable of processing the entire sequence concurrently (refer to **Fig. 1.4**), maximizing the use of parallel computing capabilities to boost training efficiency. This architecture enables direct capture of dependencies between any positions in the sequence, thus effectively addressing long-range dependency challenges. With its excellent scalability, the Transformer model facilitates extensive model expansion during large-scale training, enhancing its capability to assimilate more information and develop more intricate representations. Consequently, it has become the de facto foundational architecture for applications in NLP, visual processing, and multimodal processing.

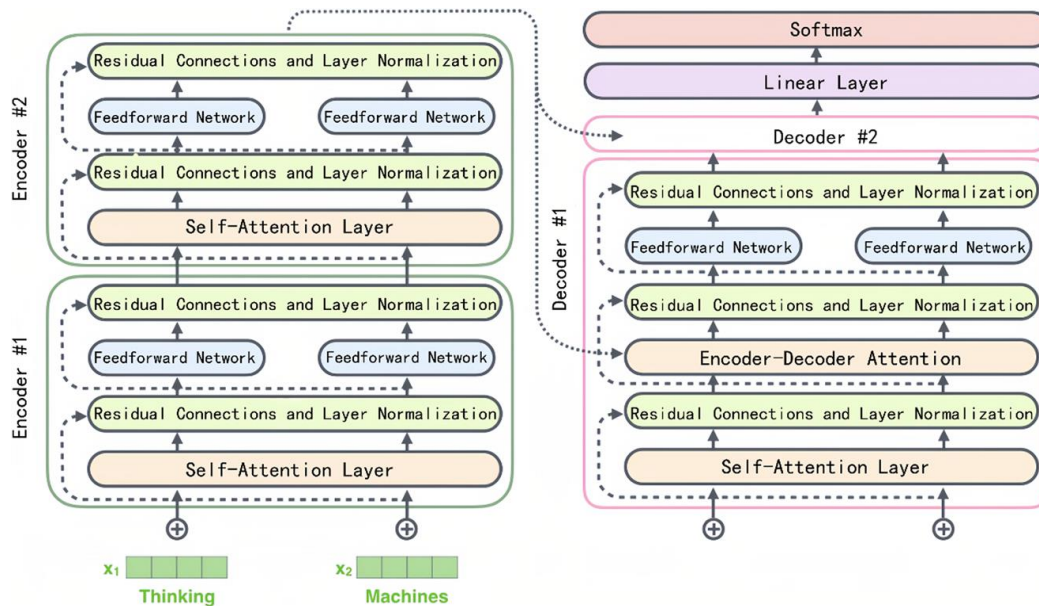


**FIGURE 1.4** Transformer model. Image source: <http://jalammar.github.io/illustrated-transformer/>.

On the left, there are two encoders labeled Encoder number 1 and Encoder number 2. Each encoder consists of a Self-Attention Layer followed by Residual Connections and Layer Normalization, then a Feedforward Network and again Residual Connections and Layer Normalization. The input to Encoder number 1 is labeled as  $x_1$  end sub Thinking and the input to Encoder number 2 is labeled as  $x_2$  end sub Machines. On the right, there are two decoders labeled Decoder number 1 and Decoder number 2. Each decoder consists of a Self-Attention Layer, followed by Residual Connections and Layer Normalization, then an Encoder-Decoder Attention layer, followed by Residual Connections and Layer Normalization, then a Feedforward

Network and again Residual Connections and Layer Normalization. The output of Decoder number 2 goes through a Linear Layer and then a Softmax layer.

It is essential to highlight that while the original Transformer model incorporates an encoder-decoder architecture, GPT exclusively employs the decoder component (refer to **Fig. 1.5**). Both the encoder and decoder consist of  $N$  identical layers, each stacked sequentially. The source and target sequences, which may be images or text, are initially processed through an embedding layer to standardize the data to the same dimensionality. Each encoder layer features a multihead attention module (with  $Q$ ,  $K$ ,  $V$  inputs) and a feedforward network module. The decoder layer commences with a multihead attention module, succeeded by another multihead attention module that interacts with the outputs from the encoder stack (i.e., the queries  $Q$  originate from the decoder, while the values  $V$  and keys  $K$  are derived from the encoder), and concludes with a feedforward network module. The output stage is processed through a softmax classifier, which determines the category or vocabulary with the highest probability.

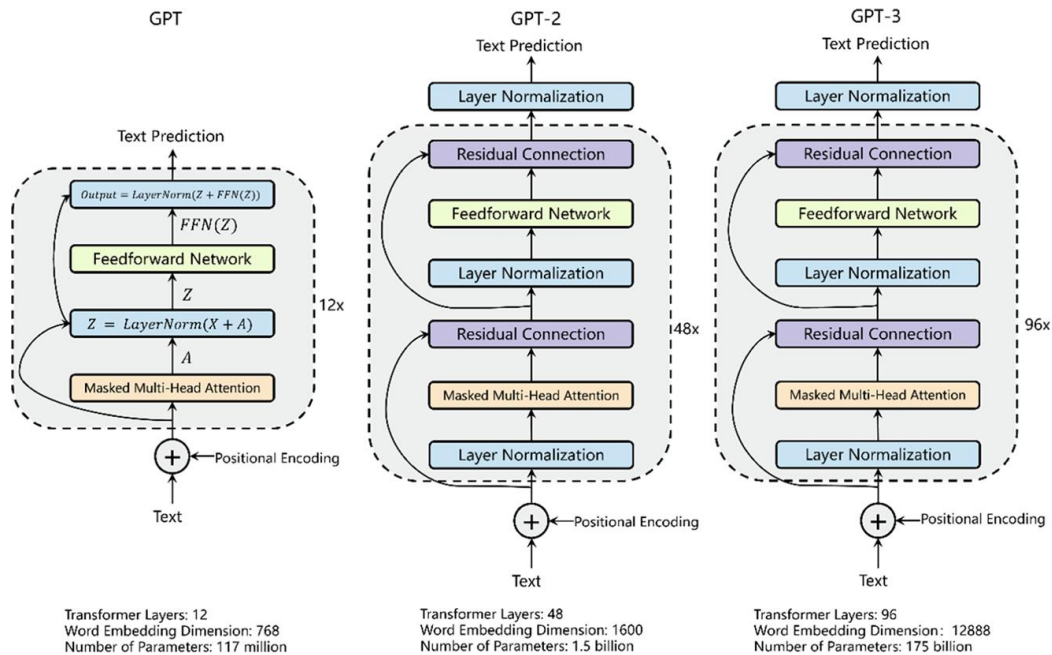


**FIGURE 1.5** Diagram of the GPT series model architecture. Image source: <https://api.stock.us/api/v1/report-file/wz3g1myv?download>.

The GPT model includes 12 transformer layers, a word embedding dimension of 768 and 117 million parameters. It features masked multi-head attention, layer normalization and a feedforward network. The GPT-2 model has 48 transformer layers, a word embedding dimension of 1600 and 1.5 billion parameters. It includes masked multi-head attention, layer normalization, residual connections and a feedforward network. The GPT-3 model consists of 96 transformer layers, a word embedding dimension of 12888 and 175 billion parameters. It also features masked multi-head attention, layer normalization, residual connections



and a feedforward network. Each model uses positional encoding for text input and outputs text predictions.

### **1.3.3 Unsupervised pretraining**

All models in the GPT series undergo an unsupervised pretraining phase; however, they differ markedly in model size and the volume of textual data utilized. This phase leverages extensive corpus statistics to capture either unidirectional or bidirectional contextual information, which is sensitive to the relative relationships of data before and after in the sequence. As a generative model, GPT aims to predict the next word in a given context, consistently striving to generate text that is most relevant and coherent with the context, thereby maximizing the conditional probability  $P(\text{target}|\text{context})$ . As illustrated in **Fig. 1.6**, during the pretraining phase, GPT employs a unidirectional (masked self-attention) language model, predicting solely based on the left-side context without utilizing right-side context information.

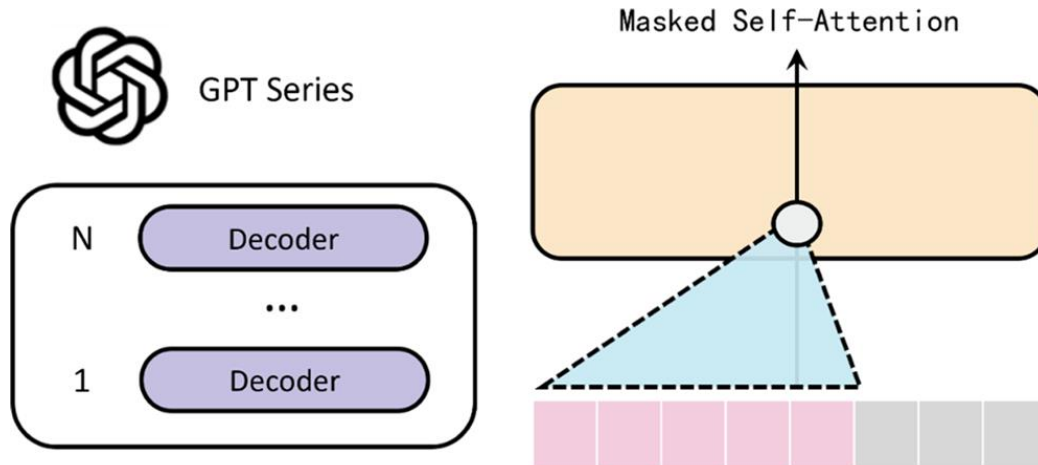


**FIGURE 1.6** GPT architecture’s masked self-attention. Image source: <http://jalammar.github.io/illustrated-gpt2/>.

On the left side, there is a box labeled GPT Series containing multiple decoders stacked from 1 to N. On the right side, there is a representation of masked self-attention. The masked self-attention mechanism is depicted with an arrow pointing upwards from a circle, indicating the focus on the left-side context without utilizing right-side context information. Below the masked self-attention, there are several blocks arranged horizontally.

The pretraining mechanism of the GPT model acquires linguistic and world knowledge from vast text corpora. Linguistic knowledge includes lexical, morphological, syntactic, and semantic aspects, which facilitate the understanding of natural language by both humans and machines. World knowledge encompasses factual

knowledge (events occurring in the real world) and common-sense knowledge. Transformer-based language models are capable of learning basic linguistic knowledge from corpora ranging from tens of millions to billions of examples. However, acquiring factual knowledge requires significantly larger-scale training data. Unlike the relatively stable and finite linguistic knowledge, factual knowledge is not only abundant but also continuously evolving. Current research indicates that as training data volumes increase, the performance of pretrained models on various downstream tasks consistently improves, suggesting that these models primarily acquire world knowledge from the incremental training data. This expansion in training data enables models to more effectively address challenges in NLP tasks, thereby enhancing their performance and accuracy across different application scenarios.

#### **1.3.4 Supervised fine-tuning**

Although the pretraining phase endows GPT models with a broad spectrum of linguistic and world knowledge, achieving exemplary performance on specific tasks often necessitates supervised fine-tuning. In this phase, the model utilizes a labeled dataset to refine its parameters via supervised learning, thereby better aligning with specific task requirements. Typically, for the GPT series, this process is applied to NLP tasks such as question answering, dialogue generation, and sentiment analysis. However, not all GPT models are subjected to supervised

fine-tuning due to the complexities and high costs associated with constructing training datasets for downstream tasks. Moreover, as the number of model parameters increases, the cost of training escalates exponentially. In the GPT series, while models like GPT-1 and GPT-3.5 and above have undergone supervised fine-tuning with a limited number of samples, their core competencies primarily derive from the extensive pretraining phase. These models are further adapted to more downstream tasks through contextual learning.

### **1.3.5 Human feedback reinforcement learning**

Supervised fine-tuning can improve model performance on specific tasks, yet challenges may persist. To further refine the model, versions GPT-3.5 and above have incorporated human feedback reinforcement learning (HFRL). As depicted in **Fig. 1.7**, this process involves evaluating the responses generated by the model using human reviewers, whose assessments serve as reinforcement learning signals to enhance model performance. Consequently, ChatGPT can progressively improve the quality of its responses through interactions with humans.

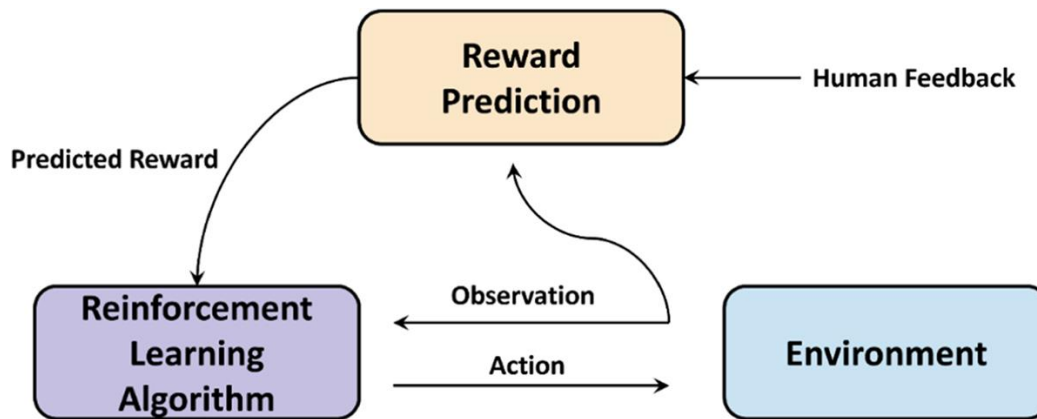


**FIGURE 1.7** The fundamental principles of reinforcement learning with human feedback.

The components are: Reinforcement Learning Algorithm, Reward Prediction and Environment. The Reinforcement Learning Algorithm generates a Predicted Reward which is sent to Reward Prediction. Human Feedback is provided to Reward Prediction. The Reward Prediction sends Observation to the Reinforcement Learning Algorithm. The Reinforcement Learning Algorithm sends Action to the Environment. The Environment sends Observation back to the Reinforcement Learning Algorithm.

Specifically, questions are randomly selected from a dataset, and human annotators provide high-quality answers, which are then utilized to fine-tune the GPT model. Through this supervised learning phase, the model is trained to produce answers that more accurately meet human expectations. HFRL comprises three steps: initial supervised fine-tuning; subsequent training of a reward

model (RM), where human annotators rank the outputs, and these rankings are employed to train the RM; and finally, using the trained RM to refine the strategy through reinforcement learning, updating the parameters of the pretrained model based on feedback from the RM. By iteratively repeating the last two steps, a higher-quality model is developed, better aligning the generated content with human requirements.

## 1.4 The technology stack of large language model

LLMs such as ChatGPT are rapidly evolving into essential infrastructural elements. They encompass a varied technological stack that incorporates features like containerization, performance monitoring, business intelligence, event processing, cloud services, microservices, and analytical tools. This technological stack can be systematically categorized into five distinct layers, ascending from the foundational level as depicted in **Fig. 1.8**.

1. **Infrastructure layer:** This layer is fundamental in providing the necessary infrastructure for the execution of both training and inference tasks, which includes the use of public clouds, private clouds, and specialized computational hardware. At this level, the efficiency of AI-specific hardware, such as GPUs and tensor processing units, often becomes a critical

bottleneck. The computational efforts for most extensive language models predominantly occur on these devices within cloud environments, serving both model providers and research laboratories for training purposes, as well as businesses engaged in model deployment or fine-tuning. Since the transition from CPUs to GPUs as the primary computing units in AI, the field has confronted renewed challenges associated with the immense demands for computational power, marking a significant challenge over the past decade.

2. **Framework layer:** This layer includes advanced deep learning and distributed training frameworks that are essential for the training and deployment of models. Prominent frameworks such as PyTorch and TensorFlow, along with distributed training systems like Pathway, Megatron-LM, and DeepSpeed Zero, fall within this category.
3. **Model layer:** This layer comprises a variety of open-source and proprietary models, which typically support higher layers, such as the application layer or middleware, through APIs.
4. **Middleware layer:** This layer involves the encapsulation of interfaces for LLMs, optimization prompts, integration of external knowledge, and interfaces for additional third-party tools. It provides critical middleware services equipped with long-term memory and proxy capabilities, which meet diverse

workflow requirements from the user layer. This layer simplifies workflow decomposition, tool invocation, and the connection between the application and model layers, thereby easing the interaction between developers and models and reducing barriers to the practical utilization of LLMs.

5. **Application layer:** This final layer integrates AI models into end-user applications. Many businesses incorporate LLMs into their client-facing applications or search engines, either via the middleware layer or through direct model invocation. This integration empowers products with the advanced collaborative and generative capabilities of LLMs, exemplified by Microsoft's New Bing.



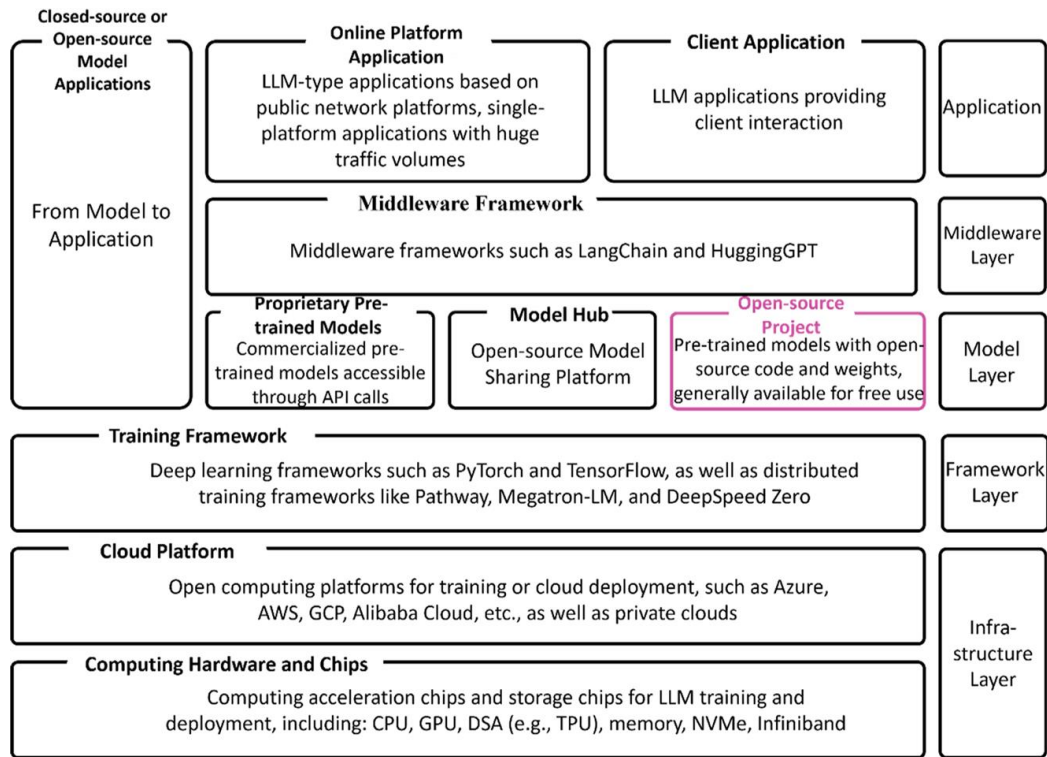


**FIGURE 1.8** Technology stack of large language models.

The layers are as follows: Application Layer, Middleware Layer, Model Layer, Framework Layer and Infrastructure Layer. The Application Layer includes Online Platform Application and Client Application. Online Platform Application refers to LLM-type applications based on public network platforms, single-platform applications with huge traffic volumes. Client Application refers to LLM applications providing client interaction. The Middleware Layer includes Middleware Framework, which refers to middleware frameworks such as LangChain and HuggingGPT. The Model Layer includes Proprietary Pre-trained Models, Model Hub and Open-source Project. Proprietary Pre-trained Models refer to commercialized

pre-trained models accessible through API calls. Model Hub refers to open-source model sharing platform. Open-source Project refers to pre-trained models with open-source code and weights, generally available for free use. The Framework Layer includes Training Framework, which refers to deep learning frameworks such as PyTorch and TensorFlow, as well as distributed training frameworks like Pathway, Megatron-LM and DeepSpeed Zero. The Infrastructure Layer includes Cloud Platform and Computing Hardware and Chips. Cloud Platform refers to open computing platforms for training or cloud deployment, such as Azure, AWS, GCP, Alibaba Cloud, etc., as well as private clouds. Computing Hardware and Chips refer to computing acceleration chips and storage chips for LLM training and deployment, including CPU, GPU, DSA (e.g., TPU), memory, NVMe, Infiniband.

Moreover, some proprietary models, such as OpenAI's ChatGPT, are directly embedded into upper-layer applications. Other organizations, motivated by data privacy or specific business needs, may adapt open-source large models to develop bespoke solutions tailored to unique industry requirements, thereby offering customized services to end users.

## **1.5 The impact of large language models**

The success of ChatGPT has had profound implications for NLP and related fields. Looking forward, technological development trends in this area may exhibit the following characteristics:

1. The independent research significance of NLP subfields may gradually diminish. As the scale of LLMs expands, the performance of various subtasks within the NLP domain is expected to improve significantly. Traditional fine-tuning methods based on small models may gradually be replaced by approaches leveraging large models and content learning techniques, such as prompting. Many issues previously considered “domain-specific” may primarily stem from a lack of domain knowledge. With sufficient domain knowledge, these problems can potentially be resolved. The path to achieving AGI might be more straightforward than anticipated: it may simply require providing LLMs with more domain-specific data to enable self-learning.
2. LLMs are anticipated to become multimodal and serve as general human-computer interfaces. As illustrated in **Fig. 1.9**, the application domains of LLMs are expected to extend beyond NLP to include image processing, video, and speech tasks. The demonstration during the release of GPT-4 has already showcased the application scenarios of multimodal LLMs. These models have the potential to evolve into general human-computer interfaces

supporting multimodal input and output. This trend may start with operating systems and commonly used software tools, gradually permeating software across various industries, ultimately achieving widespread application.

3. The training and deployment of large models with low computational power will become a focal point of research and application. Given the substantial financial investment required to develop LLMs, only a few major enterprises are likely to continue releasing LLMs that can compete with ChatGPT. Many companies operating in specific vertical domains may attempt to customize open-source versions by leveraging domain-specific data to establish their own competitive “moats.” However, the capabilities of LLMs generally become evident only after reaching a certain scale threshold. Thus, achieving ideal results through small-scale models with domain data fine-tuning may not be effective. Consequently, low computational power training and deployment of large models will inevitably become a prominent area of research and application.
4. Middleware based on LLMs will integrate enterprise workflows. Many small and medium-sized enterprises may utilize middleware based on LLMs (see **Fig. 1.10**) to integrate their workflows, thereby enhancing their competitiveness, despite potential compromises to the security of domain data. Large

models like ChatGPT not only demonstrate efficiency in processing and generating multimodal data but also exhibit complex reasoning capabilities and robustness to out-of-distribution events. This enables LLMs to function as intelligent agents that integrate existing enterprise workflows, thereby automating processes and improving efficiency. This could lead to a concerning spiral effect: the more domain data is monopolized by LLMs, the faster their capabilities will iterate. Consequently, other small and medium-sized enterprises, to avoid falling behind in the competition, will increasingly rely on the efficiency improvements brought about by integrating their workflows with middleware based on LLMs.

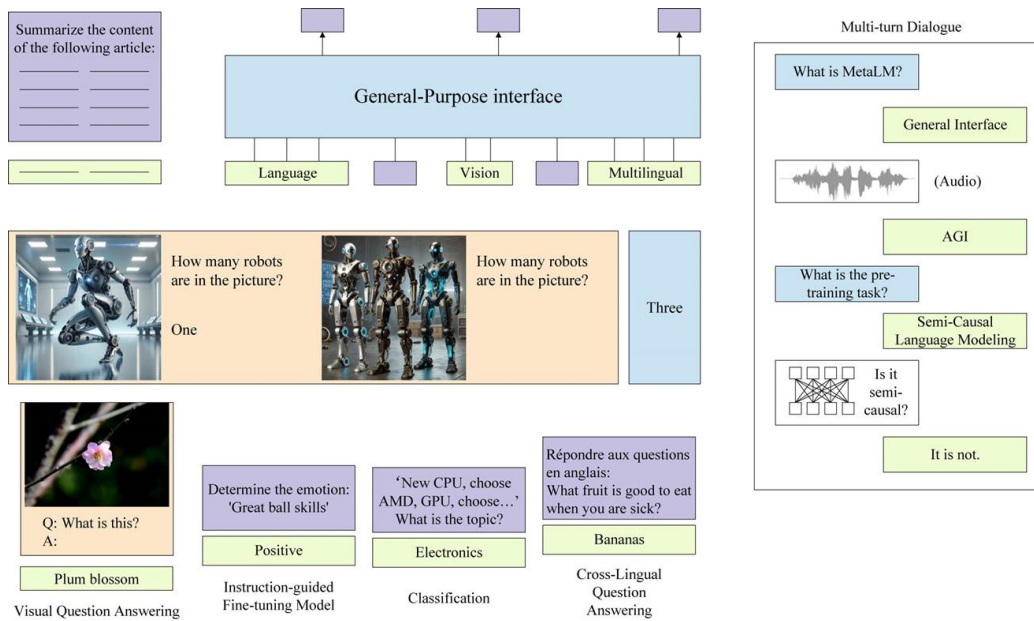


**FIGURE 1.9** Large language models as a universal interface.

At the top, there is a box labeled Summarize the content of the following article connected to a General-Purpose interface box. Below this, there are three boxes labeled Language, Vision and Multilingual. Below these, there are various examples, a picture of a robot with the question How many robots are in the picture? and the answer One, another picture of multiple robots with the same question and the answer Three. There is also a picture of a flower with the question Q: What is this? and the answer Plum blossom, labeled as Visual Question Answering. Other examples include Determine the emotion: Great ball skills with the answer Positive labeled as Instruction-guided Fine-tuning Model, New CPU, choose AMD, GPU, choose... What is the topic? with the answer Electronics labeled as Classification and Répondre aux questions en anglais: What

fruit is good to eat when you are sick? with the answer Bananas labeled as Cross Lingual Question Answering. On the right side, there is a section labeled Multi turn Dialogue with questions and answers about MetaLM, general interface, audio, AGI, pre-training task and semi-causal language modeling.

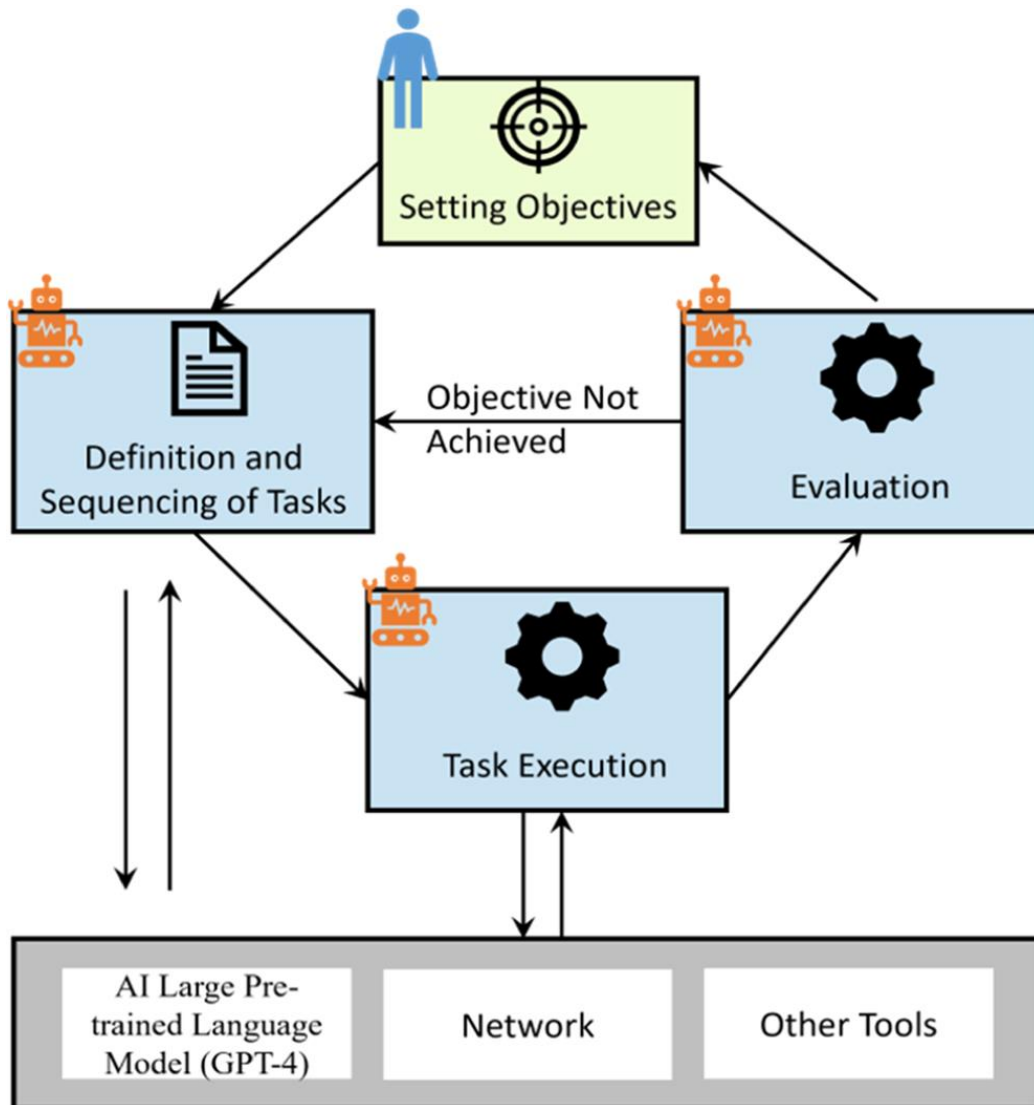


**FIGURE 1.10** Diagram for integrating workflows based on the middleware of large language models (AutoGPT).

The process starts with Setting Objectives which leads to Definition and Sequencing of Tasks. If the Objective Not Achieved, it loops back to Definition and Sequencing of Tasks. Otherwise, it proceeds to Task Execution. Task Execution is followed by Evaluation. If the evaluation is successful, the process ends. If not, it loops back to



Definition and Sequencing of Tasks. The Task Execution is supported by AI Large Pre-trained Language Model (GPT-4), Network, and Other Tools.

## **1.6 The challenges of training or deploying large models**

### **1.6.1 Computational power**

Before the emergence of LLMs like GPT-4, there were few applications that required such powerful GPU computing capabilities. The computational cost of these LLMs mainly involves two aspects: the initial training of the model and subsequent operational maintenance. For companies planning to train or deploy LLMs like GPT-4, their demand for computing power is mainly reflected in the following two stages:

1. Pretraining and fine-tuning stage: In this stage, the model undergoes pretraining from scratch using vast amounts of general data, simultaneously validating the effectiveness of the pretrained model to establish its foundational knowledge. Subsequently, the model is fine-tuned with specific data tailored to particular commercial applications or other scenarios to enhance response accuracy. While specific data for GPT-4 at this stage has not been disclosed, estimates can be made based on other LLMs. For instance, the Meta LLaMA model with 65 billion parameters was

trained on 1.4 trillion tokens over approximately 21 days using 2048 Nvidia A100 GPUs, incurring a training cost of about 1 million GPU hours.

For a model like GPT-3, with 175 billion parameters, the training requires  $3.14 \times 10^{23}$  FLOPS. Assuming the use of NVIDIA 80GB A100 GPUs (theoretical performance of 312 TFLOPS) and employing tensor parallelism and pipeline parallelism technologies to achieve over 50% utilization, the estimated requirement is 758 A100 GPUs for a training duration of 1 month. Although specific parameters for ChatGPT 3.5 and 4.0 have not been disclosed, they are presumed to be on a similar scale to GPT-3, with 175 billion parameters. Based on available data from the ChatGPT website, the memory usage is estimated to be between 350 and 500 GB. However, training might require the computational power of over 1000 A100 GPUs to achieve results within an acceptable timeframe. It is also reported that Microsoft and OpenAI constructed a supercomputer with over 10,000 GPUs for training GPT-3, with a single training session costing around \$5 million.

2. Inference and deployment stage: Once fine-tuned, the LLM can be deployed in real-world production environments. Compared with the training stage, the computational demand during the deployment stage is relatively lower, but the scale is significantly larger. For extensive online interactions, the

deployment stage incurs substantial server and chip costs, potentially surpassing those of the training stage. This is because numerous servers are needed in a cluster to provide parallel network services, primarily involving extensive matrix computations and storage management. In specific scenarios, such as edge applications, there may be unique requirements for hardware cost-effectiveness and response latency, where traditional GPUs might not suffice.

According to a recent interview with OpenAI CEO Sam Altman, we can make the following assumptions: If OpenAI adopts the dense model architecture of GPT-3, with 175 billion parameters, a hidden layer dimension of 16k, a sequence length of 4k, an average token count of 2k per response, and 15 responses per user for 13 million daily active users, considering 50% hardware utilization due to idle times and a cost of \$1 per GPU per hour (with performance twice that of FasterTransformer), the daily computational hardware cost for ChatGPT is approximately \$690,000. OpenAI would require about 3617 HGX A100 servers (totaling 28,936 GPUs) to provide ChatGPT services, with each query costing around 0.36 cents.

### **1.6.2 Data**

The training of LLMs relies on vast amounts of data, which enable them to learn and master the various knowledge encapsulated in natural language. At the same time,

considering the quality of LLMs, these data must be effectively cleaned, annotated, organized, and stored. Therefore, the cost of the infrastructure, tools, and data engineers required for these tasks, as well as data management and processing, can rapidly increase.

For example, the LLaMA model utilized a training dataset totaling 4.6TB, comprising 1.4 trillion tokens. As illustrated in **Fig. 1.11**, the pretraining text data for GPT-3 amounted to a staggering 45TB. These massive data volumes highlight the resources needed for model training and underscore the importance of data quality. It has been reported that the world's largest Chinese corpus, WuDaoCorpora, stores 3TB of Chinese text data, with 200GB made publicly available.

Text Training Dataset

Pre-trained Dataset (Basic Training)

| Dataset                | Quantity (Tokens) | Weights in Mixed Training | Epochs Required for Training 300B Tokens |
|------------------------|-------------------|---------------------------|------------------------------------------|
| Common Crawl(filtered) | 410 billion       | 60%                       | 0.44                                     |
| WebText2               | 19 billion        | 22%                       | 2.9                                      |
| Books1                 | 12 billion        | 8%                        | 1.9                                      |
| Books2                 | 55 billion        | 8%                        | 0.43                                     |
| Wikipedia              | 3 billion         | 3%                        | 3.4                                      |

Training Dataset for Reward Model (Establishing Behavior Norms)

Used for training the Reward Model (RM), all annotations and scores are manually labeled, providing corresponding rankings between K=4-9.

| 6B Reward Model Dataset (33K Labeled Instances) |               |       |
|-------------------------------------------------|---------------|-------|
| Category                                        | Corpus Source | Count |
| Training Dataset                                | Annotator     | 6623  |
| Training Dataset                                | Client        | 26584 |

Training Dataset for Supervised Fine-Tuning (Cognitive/Expressive Training)

Three Types of Prompts: Plain, Few-Shot, User-based

| SFT Dataset (13K Labeled Instances) |               |       |
|-------------------------------------|---------------|-------|
| Category                            | Corpus Source | Count |
| Training Dataset                    | Annotator     | 11295 |
| Training Dataset                    | Client        | 1430  |

Training Dataset for PPO (Domain Generalization)

Unlabeled, Used as Input for RLHF Fine-Tuning

| SFT Dataset (13K Labeled Instances) |               |       |
|-------------------------------------|---------------|-------|
| Category                            | Corpus Source | Count |
| Training Dataset                    | Annotator     | 11295 |

**FIGURE 1.11** Composition of the GPT-3 training dataset. Data source: paper “Training language models to follow instructions with human feedback”.

The table consists of three main sections: Pre-trained Dataset (Basic Training), Training Dataset for Supervised Fine-Tuning (Cognitive/Expressive Training) and Training Dataset for PPO (Domain Generalization). The Pre-trained Dataset section has four columns: Dataset, Quantity (Tokens), Weights in Mixed Training and Epochs Required for Training 300B Tokens. It reads as follows: Row 1. Common Crawl (filtered): 410 billion, 60 percent, 0.44. Row 2. WebText2: 19 billion, 22 percent, 2.9. Row 3. Books1: 12 billion, 8 percent, 1.9. Row 4. Books2: 55 billion, 8 percent, 0.43. Row 5. Wikipedia: 3 billion, 3 percent, 3.4. The Training Dataset for Supervised Fine-

Tuning section has three columns: Category, Corpus Source and Count. It reads as follows: Row 1. Training Dataset: Annotator, 11295. Row 2. Training Dataset: Client, 1430. The Training Dataset for PPO section has three columns: Category, Corpus Source and Count. It reads as follows: Row 1. Training Dataset: Annotator, 11295.

For training a Chinese version of GPT-3.5, only a few vertical industry companies, apart from giants in the search and social sectors, have access to sufficient data. However, a significant legal and ethical issue arises regarding whether the text data acquired during business operations can be used for model training. Even Codex, which was trained on open-source data from GitHub, faced considerable criticism. Notably, the multimodal training dataset for GPT-4 comprises both images and text, potentially including various types such as chart reasoning, physics exams, image understanding, paper summarization, and comic text.

### **1.6.3 Engineering**

The development of LLMs relies on specialized researchers and engineers who are responsible for creating the architecture and correctly configuring the training process. As illustrated in **Fig. 1.12**, the architecture of the model determines how it learns and generates text. Designing, implementing, and managing these architectures require expertise in various fields of computer science. The salaries

for engineers and researchers responsible for publishing and delivering cutting-edge research results often reach hundreds of thousands of dollars. It is important to note that the skill set required for developing LLMs may differ significantly from the traditional skill set of machine learning engineers, as depicted in **Fig. 1.2**.

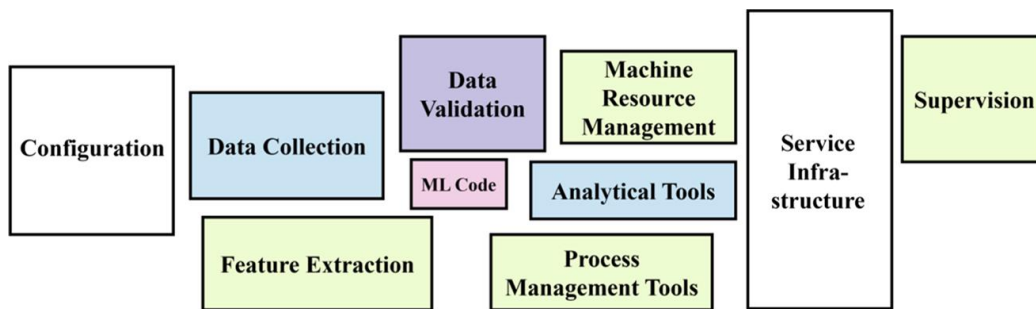


**FIGURE 1.12** Infrastructure of machine learning systems.

The diagram illustrates various components of a model architecture. The components are Configuration, Data Collection, Data Validation, Machine Resource Management, Service Infrastructure, Supervision, Feature Extraction, Process Management Tools, Analytical Tools and ML Code.

Moreover, training LLMs imposes high standards on engineering practices. This includes data cleaning, the engineering implementation of large-scale distributed training, and techniques for maintaining training stability with a large number of parameters and data volumes.

## 1.7 The limitations of large language models

While LLMs like GPT-4 have demonstrated significant capabilities in context-based dialogue, programming, and even in understanding images and analyzing data graphs, they are still far from achieving AGI. These models face several critical limitations:

**Authenticity and accuracy:** LLMs often lack robust guarantees of authenticity and accuracy. For example, in areas not extensively covered during training, GPT-4 may not exhibit human-like common sense or robust logical reasoning, sometimes generating incorrect or misleading answers. Despite improvements over previous models, GPT-4 can still produce inappropriate suggestions or inaccurate data, making it unsuitable for use in high-risk sectors like legal and medical fields.

**Interpretability:** The internal workings of LLMs like GPT-4 are complex and not fully transparent, making it challenging to guarantee the safety of their outputs. While there have been advancements in reducing error rates, the potential for harmful outputs poses a risk, potentially leading to legal issues for developers and users.

**Social and ethical implications:** These models can inadvertently generate biased, false, or offensive content due to their “black box” nature. Despite efforts to mitigate risks, there remains a possibility for misuse in creating misleading information or harmful content, such as fake news or propaganda.



**Privacy concerns:** LLMs are trained on vast amounts of data, some of which may include sensitive personal information. This raises concerns about privacy breaches, as the model could inadvertently learn and reproduce private data.

## 1.8 Summary

This chapter conducts a thorough analysis and discussion on the evolutionary trajectory of ChatGPT AI, highlighting its significant influence. Tracing the development from the initial GPT-1 and GPT-2 models to the more sophisticated GPT-3 and GPT-4, it emphasizes the pivotal role each version has played in propelling the field of AI forward.

The chapter explores the functional breadth of ChatGPT, detailing its proficiency in language comprehension and generation, the accuracy of its sentiment analysis, and its adeptness at answering questions and crafting poetry. It also offers a detailed examination of the shift from early NLP technologies to contemporary Transformer models, with a focus on the architectural design of large-scale models.

Further, the text delves into the extensive impact of LLMs across various sectors including commercial, academic, and policy realms, while also addressing significant concerns such as the immense data and computational demands required for these models. Critical challenges such as model hallucinations, biases,

interpretability issues, and privacy concerns are also discussed.

In summary, this chapter provides a holistic understanding of the operational mechanisms, potential, and challenges associated with ChatGPT and similar LLMs. Future chapters will further dissect the principles underlying ChatGPT to better comprehend its influence and prospective advancements.

---

## CHAPTER 2

# In-depth understanding of the transformer model

---

### Abstract

This chapter offers a comprehensive exploration of the Transformer model, a pivotal innovation in deep learning, particularly in natural language processing. The chapter begins by outlining the model's core components, including the self-attention mechanism, multihead attention, feedforward neural networks (FNNs), residual connections, layer normalization, and positional encoding. It delves into the self-attention mechanism's ability to capture long-range dependencies by computing attention weights among input elements. Multihead attention is discussed as an enhancement that enables simultaneous processing of multiple aspects of input sequences. The chapter also covers the role of FNNs in capturing nonlinear features, the importance of residual connections for gradient flow, and layer normalization for training stability. Positional encoding is highlighted as crucial for incorporating sequence order information. Finally,

training and optimization techniques such as loss functions, optimizers, and learning rate strategies are discussed to enhance model performance.

## **Keywords**

Transformer model; self-attention mechanism; multihead attention; feedforward neural networks; residual connections; layer normalization; positional encoding; natural language processing (NLP); training optimization

The Transformer model represents a significant shift in deep learning paradigms, primarily through its utilization of the self-attention mechanism. This innovation has fundamentally altered the conventional approaches typically associated with recurrent neural networks (RNNs) and convolutional neural networks, thereby reshaping the landscape of natural language processing (NLP). Currently, the Transformer stands as a cornerstone in NLP research and development, underpinning a variety of prominent models including bidirectional encoder representations from transformers (BERT) and generative pre-trained transformer (GPT). This chapter will provide a comprehensive examination of the Transformer model's essential components.

## **2.1 Introduction to the transformer model**

As depicted in **Fig. 2.1**, the fundamental components of the Transformer model encompass the multihead attention mechanism, feedforward neural networks (FNNs), residual connections coupled with layer normalization, and positional encoding.

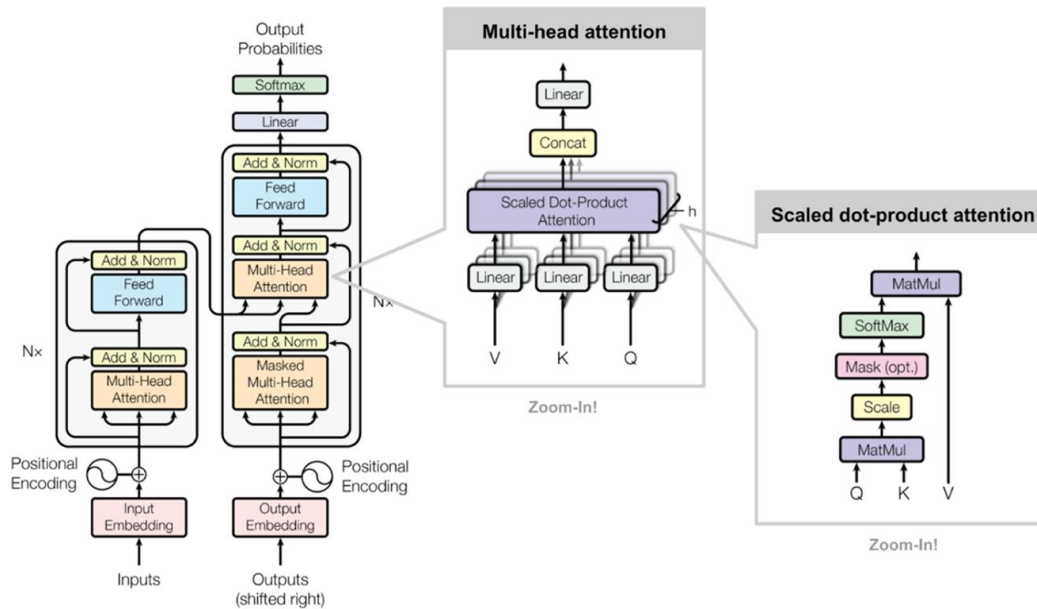


**FIGURE 2.1** Transformer model.

On the left, there is a sequence of operations starting with Input Embedding followed by Positional Encoding and then multiple layers of Add and Norm with Multi-Head Attention and Feed Forward blocks, repeated  $N$  times. On the right, there is a similar structure starting with Output Embedding followed by Positional Encoding and multiple layers of Add and Norm with Masked Multi-Head Attention, Multi-Head Attention, and Feed Forward blocks, also repeated  $N$  times. The output probabilities are obtained through a Linear layer followed by Softmax. The diagram also zooms into the Multi-Head Attention mechanism, showing Scaled Dot-Product Attention with inputs  $V$ ,  $K$ , and  $Q$  passing through Linear layers, concatenated and then processed through another Linear layer. Another zoom-in shows the Scaled Dot-Product Attention process, where  $Q$  and  $K$  are

multiplied (MatMul), scaled, optionally masked, passed through SoftMax and then multiplied with V (MatMul).

Central to the Transformer architecture is the self-attention mechanism, which assesses relationships between each element of the input sequence and every other element. Specifically, it generates “key,” “query,” and “value” representations for each input element, employing these to execute a weighted summation that produces the output sequence. An elaboration of self-attention, the multihead attention mechanism, allows the model to simultaneously address various contexts within the input sequence. It partitions the input into multiple subspaces, each processed through independent self-attention calculations. The concatenated results of these subspaces formulate the composite output sequence.

Additionally, the FNN is a pivotal element within the Transformer. Each FNN within the Transformer is composed of two linear layers with an intervening activation function. Outputs from self-attention undergo processing through the first linear layer, proceed through the activation function, and continue through the second linear layer, a sequence repeated across every Transformer layer.

To accelerate training and improve model efficacy, the Transformer architecture incorporates residual connections and layer normalization. Residual connections facilitate the integration of the input sequence by adding it element wise

to the outputs of the self-attention and FNNs, thus forming a modified sequence. Subsequent layer normalization of this sequence across its dimensions aids in hastening the training process.

The inherent design of the self-attention mechanism does not capture the sequential order of elements, necessitating the introduction of positional encoding within the Transformer model. This encoding utilizes sine and cosine functions to create unique positional encodings for each element, which are subsequently added to the elements of the input sequence, thereby embedding information about the sequence order.

For training, the Transformer predominantly employs the maximum likelihood estimation method, aiming to optimize model parameters by minimizing the cross-entropy loss. To further enhance the training speed, the Transformer commonly uses the Adam optimizer, alongside techniques such as learning rate warmup and gradient clipping.

## **2.2 Self-attention mechanism**

The conceptual foundation of the attention mechanism draws inspiration from human visual and cognitive processes. When assimilating visual or linguistic data, individuals tend to concentrate on specific segments of the input, disregarding extraneous details. This selective focus mitigates computational and cognitive loads, thereby bolstering the efficiency and accuracy of information processing.



### 2.2.1 The calculation process of self-attention

Self-attention, a specialized variant of the attention mechanism, facilitates interactions among all elements within an input sequence to ascertain their respective attention weights. This capability allows the self-attention mechanism to incorporate all elements when processing sequential data. The principal rationale of self-attention is to allocate a weight to each sequence element, dictating the model's focus intensity on other elements during the processing of a specific one. A significant advantage of self-attention is its proficiency in capturing long-range dependencies within the sequence, thus augmenting the model's representational capacity. In the Transformer, the self-attention computation involves several steps:

Initially, the input sequence—comprising elements like words, characters, or other types—is converted into vector representations via an embedding layer. Subsequently, for each element, vectors known as queries, keys, and values are derived typically through multiplication with distinct weight matrices.

As illustrated in **Fig. 2.2**, an input sequence denoted as  $x_{i=1,2,3}$  is transformed through an embedding layer to yield token embeddings  $e_{i=1,2,3}$ . Three transformation matrices,  $W^Q$ ,  $W^K$ , and  $W^V$ , are employed to convert these embeddings into vectors  $q_{i=1,2,3}$ ,  $k_{i=1,2,3}$ , and  $v_{i=1,2,3}$ , respectively, representing the tokens' semantic features in a latent space.

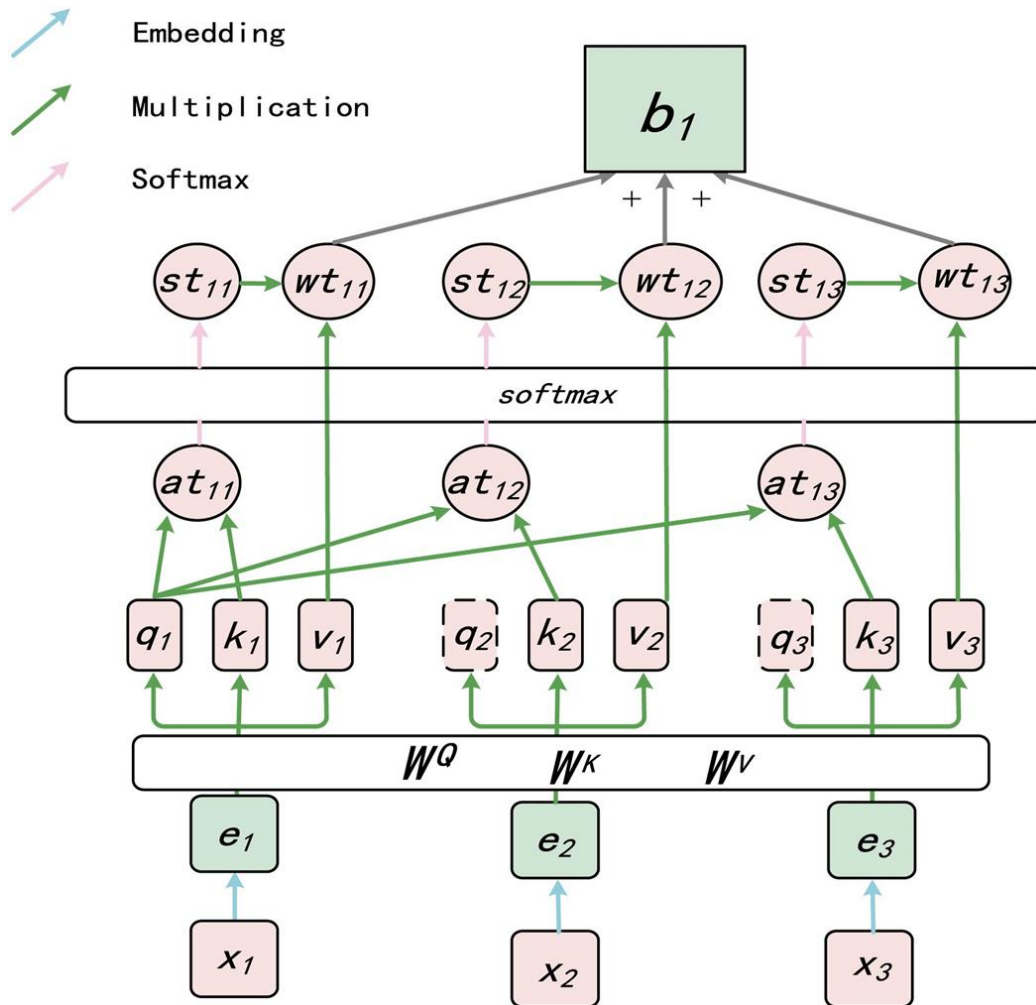


**FIGURE 2.2** Self-attention mechanism.

It includes embeddings, multiplications and softmax operations. At the bottom, there are three input vectors labeled  $x_1$ ,  $x_2$  and  $x_3$ , which are transformed into embeddings  $e_1$ ,  $e_2$  and  $e_3$ , respectively. These embeddings are then processed through weight matrices  $W^Q$ ,  $W^K$  and  $W^V$  to produce  $q_1, k_1, v_1$ ,

$q_2$ ,  $k_2$ ,  $v_2$ ,  $q_3$ ,  $k_3$  and  $v_3$ . The attention scores at  $i=1$ , at  $i=2$  and at  $i=3$  are computed using these vectors. These scores are then passed through a softmax layer to produce  $st_1$ ,  $st_2$  and  $st_3$ , which are multiplied by  $w_1$ ,  $w_2$  and  $w_3$ , respectively. The results are summed and added to  $b_1$ . The diagram includes arrows indicating the flow of operations: embedding, multiplication and softmax.

Subsequently, vector  $q_i$  is dot-multiplied with each  $k_{j=1,2,3}$  to compute initial attention scores,  $at_{ij}$ . Notably,  $at_{ij}$  is scalar. These scores are normalized using a softmax function to derive  $st_{ij}$ . Each  $st_{ij}$  then multiplies the corresponding value vectors  $v_{j=1,2,3}$  to generate vectors  $w_{ij}$ . Softmax normalization of the dot products, adjusted by the square root of the vector dimension, ensures that the sum of attention weights is unity, with individual weights ranging from 0 to 1. The summation of all  $w_{ij}$  vectors yields the output vector  $b_i$ , embodying the self-attention mechanism's essence: each input token's output vector is computed similarly. This procedure is replicated across the input sequence to generate all output vectors  $b_2, b_3$ . Unlike RNNs, the computation of self-attention does not rely on prior hidden states but includes positional information in the embeddings, allowing for the model to recognize the

relative positions of tokens and facilitating the parallelization of computations.

As depicted in **Fig. 2.3**, by concatenating the vectors  $e_{1,2,3}$  into matrix “ $E$ ,” matrix-based self-attention operations are enabled. Multiplying “ $E$ ” by the parameter matrices  $w^Q$ ,  $w^K$ , and  $w^V$  results in matrices “ $Q$ ,” “ $K$ ,” and “ $V$ .”

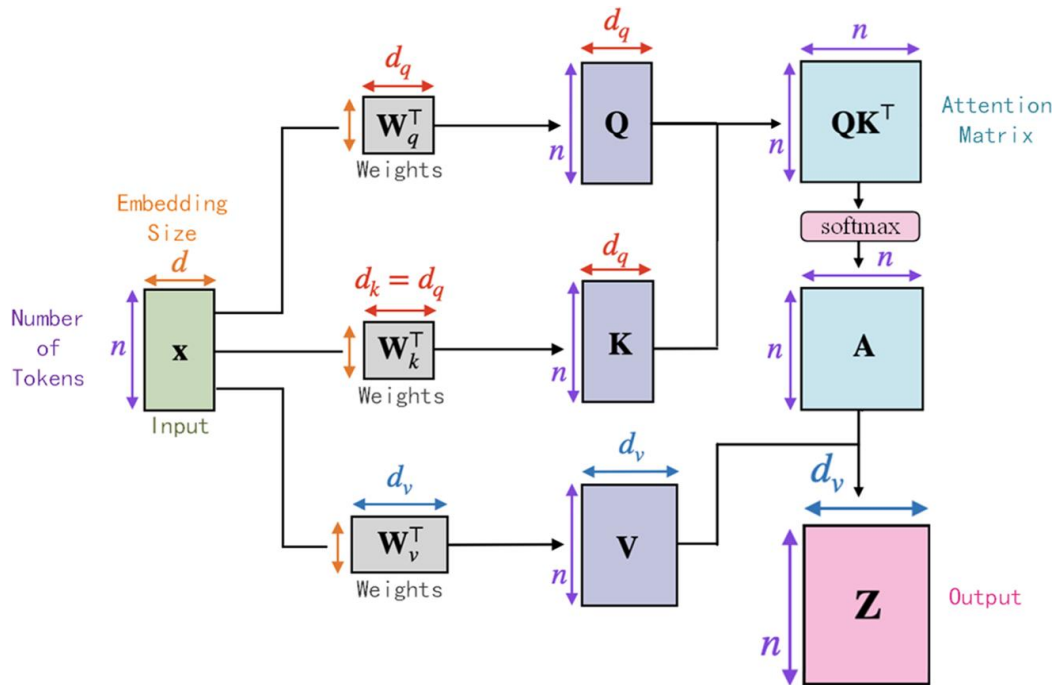


**FIGURE 2.3** Matrix computation process of the self-attention mechanism.

<https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html>.

The process starts with an input matrix  $\mathbf{x}$  with dimensions  $n$  by  $d$ , where  $n$  is the number of tokens and  $d$  is the embedding size. The input  $\mathbf{x}$  is multiplied by three weight matrices  $\mathbf{W}_q^T$ ,  $\mathbf{W}_k^T$  and  $\mathbf{W}_v^T$  to produce matrices  $\mathbf{Q}$ ,  $\mathbf{K}$  and  $\mathbf{V}$ , respectively. The dimensions of  $\mathbf{W}_q^T$  and  $\mathbf{W}_k^T$  are  $d$  by  $d_q$  and the dimension of  $\mathbf{W}_v^T$  is  $d$  by  $d_v$ . The resulting matrices  $\mathbf{Q}$  and  $\mathbf{K}$  have dimensions  $n$  by  $d_q$  and  $\mathbf{V}$  has dimensions  $n$  by  $d_v$ . The matrix  $\mathbf{Q}$  is then multiplied by the transpose of matrix  $\mathbf{K}$  ( $\mathbf{K}^T$ )

superscript T end superscript) to produce an attention matrix with dimensions  $n$  by  $n$ . This attention matrix is passed through a softmax function to produce matrix  $A$  with dimensions  $n$  by  $n$ . Finally, matrix  $A$  is multiplied by matrix  $V$  to produce the output matrix  $Z$  with dimensions  $n$  by  $d_v$ .

The attention score matrix is calculated by multiplying “ $Q$ ” with the transpose of “ $K$ .” This matrix is then row-normalized using softmax to produce a normalized attention score matrix “ $A$ ,” where each row sums to one. This matrix “ $A$ ” of size  $n \times n$  is multiplied with “ $V$ ” to produce the final matrix “ $Z$ ,” which dimensions  $n \times d_v$  indicate “ $n$ ” tokens, each with a value vector of length  $d_v$ . The self-attention formula is expressed as

$$AT(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \quad (2.1)$$

Here, “ $Q$ ,” “ $K$ ,” “ $V$ ” are hidden state matrices derived by multiplying the input matrix “ $X$ ” with three parameter matrices, and  $d_k$  represents the number of columns in matrix  $W_k$ .

The division by  $\sqrt{d_k}$  addresses potential vanishing gradient issues. When  $d_k$  is large, the variance in the attention scores can be significant, leading to diminished gradients as extreme values are squashed by the softmax function. Normalizing by  $\frac{1}{\sqrt{d_k}}$  reduces this variance from  $d_k$  to 1, thereby stabilizing the gradients.

### 2.2.2 The essence of the self-attention mechanism

The matrices “Q,” “K,” and “V” represent linear transformations of the input matrix “X” and can be conceptualized as the semantic representations of “X” in a latent space. For simplification, these matrices may collectively be referred to as “X.” Omitting the normalization factor  $\frac{1}{\sqrt{d_k}}$  leads to a modified version of [Eq. \(2.1\)](#), expressed as

$$AT(Q, K, V) = \text{softmax}(XX^T)X. \quad (2.2)$$

In this formulation,  $XX^T$  symbolizes the product of matrix “X” and its transpose, an operation in vector space representing the inner products between vectors. This reflects the similarity of the input vectors. The geometric interpretation of an inner product between two vectors is indicative of the angle between them and the projection of one vector onto another. Consequently, the higher the similarity between two vectors, the greater the projection value; conversely, when two vectors are perpendicular, indicating linear independence, their inner product equates to zero.

As demonstrated in [Fig. 2.4](#), consider an input sequence  $x_1$  represented by elements  $(x_1, x_2, x_3)$ . Should the embedding layer utilize one-hot encoding, the input matrix “X” assumes a specific configuration. The product of this matrix with its transpose effectively serves as an attention weight

matrix. This configuration facilitates the straightforward computation of attention weights, deriving from the similarities and dissimilarities among various input elements, thereby simplifying and enhancing the efficiency of the computational process within the attention mechanism.

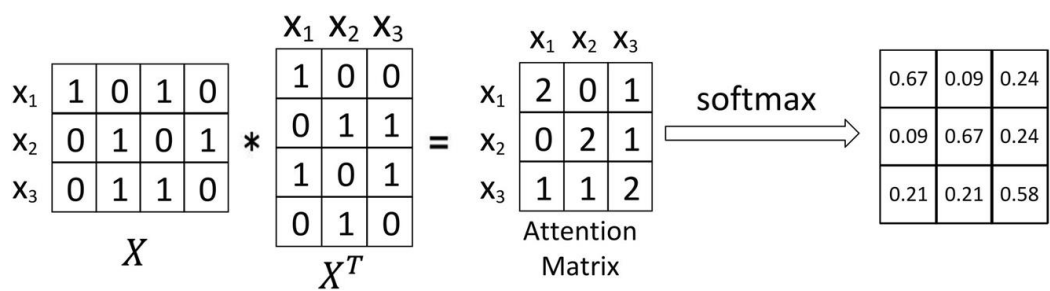


**FIGURE 2.4** Attention weight matrix.

The first matrix X has elements: row x1 (1, 0, 1, 0), row x2 (0, 1, 0, 1) and row x3 (0, 1, 1, 0). The second matrix X superscript T end superscript has elements: row x1 (1, 0, 0, 0), row x2 (0, 1, 1, 1) and row x3 (1, 0, 1, 0). The result of the multiplication is the Attention Matrix with elements: row x1 (2, 0, 1), row x2 (0, 2, 1) and row x3 (1, 1, 2). The softmax operation is then applied to the Attention Matrix, resulting in a matrix with elements: row x1 (0.67, 0.09, 0.24), row x2 (0.09, 0.67, 0.24) and row x3 (0.21, 0.21, 0.58).

The attention weight matrix exemplifies the essence of inner product calculations, which is to gauge similarity. For instance, consider the first row, which calculates the



similarity of the token  $x_1$  with all other tokens in the sequence. If tokens A and B cooccur frequently, their similarity is generally high. In this example, the similarity of  $x_1$  with itself and with the token  $x_3$  is notably high, suggesting a closer relationship between  $x_1$  and these two tokens compared to  $x_2$ .

Subsequently, these results are normalized using the softmax function to produce a normalized attention weight matrix. As depicted in **Fig. 2.5**, this matrix, once normalized, functions as a coefficient matrix. This matrix is then right-multiplied by the input matrix  $x$ , resulting in  $\hat{x}$ . This operation involves multiplying each row of the coefficient matrix by each column of  $x$ , effectively computing the values for each row of the output matrix  $\hat{x}$ . Each row in  $\hat{x}$  represents the weighted word embedding of the corresponding token, influenced by the attention weights.

|       |          |       |       |   |       |     |   |   |   |   |           |      |      |      |      |
|-------|----------|-------|-------|---|-------|-----|---|---|---|---|-----------|------|------|------|------|
|       | $x_1$    | $x_2$ | $x_3$ |   |       |     |   |   |   |   |           |      |      |      |      |
| $x_1$ | 0.67     | 0.09  | 0.24  | * | $x_1$ | 1   | 0 | 1 | 0 | = | $x_1$     | 0.67 | 0.33 | 0.91 | 0.09 |
| $x_2$ | 0.09     | 0.67  | 0.24  |   | $x_2$ | 0   | 1 | 0 | 1 |   | $x_2$     | 0.09 | 0.91 | 0.33 | 0.67 |
| $x_3$ | 0.21     | 0.21  | 0.58  |   | $x_3$ | 0   | 1 | 1 | 0 |   | $x_3$     | 0.21 | 0.79 | 0.79 | 0.21 |
|       | $\alpha$ |       |       |   |       | $X$ |   |   |   |   | $\hat{X}$ |      |      |      |      |

**FIGURE 2.5** Word embeddings weighted by attention scores.

The table consists of three matrices: alpha, X and X hat. It reads as follows: Matrix alpha. Row 1.  $x_1$ : 0.67, 0.09, 0.24. Row 2.  $x_2$ : 0.09, 0.67, 0.24. Row 3.  $x_3$ : 0.21, 0.21, 0.58. Matrix X. Row 1.  $x_1$ : 1, 0, 1, 0. Row 2.  $x_2$ : 0, 1, 0, 1. Row 3.  $x_3$ : 0, 1, 1, 0. Matrix X hat. Row 1.  $x_1$ : 0.67, 0.33, 0.91, 0.09. Row 2.  $x_2$ : 0.09, 0.91, 0.33, 0.67. Row 3.  $x_3$ : 0.21, 0.79, 0.79, 0.21.

In summary, the self-attention mechanism, given an input matrix  $x$ , yields an output matrix  $\hat{x}$  by computing attention weights among the input elements. This process involves calculating dot products between the query vectors and the key vectors to assess the similarity between two elements. These attention weights are subsequently utilized to execute a weighted summation of the value vectors, generating an output vector for each input element. These output vectors may either serve as inputs to subsequent layers or, in combination with other layers, contribute to the model's overall output.

As illustrated in **Fig. 2.6**, the matrices “ $Q$ ,” “ $K$ ,” and “ $V$ ” are all derived from linear transformations of the input matrix  $X$ . In theory, these transformations could be accomplished with a single matrix to achieve attention weighting. Nonetheless, utilizing three distinct matrices to generate query, key, and value vectors is instrumental in the attention mechanism. The query vector acts as the “question” posed by the current element during attention weight computation, evaluating its focus on other elements by comparing with their key vectors. The key vector serves as the “answer” for each element in the input sequence, with attention weights calculated based on the similarity between query and key vectors. The value vector carries the actual information of each element in the sequence. These attention weights are then used to aggregate the value vectors, producing the final output vector.

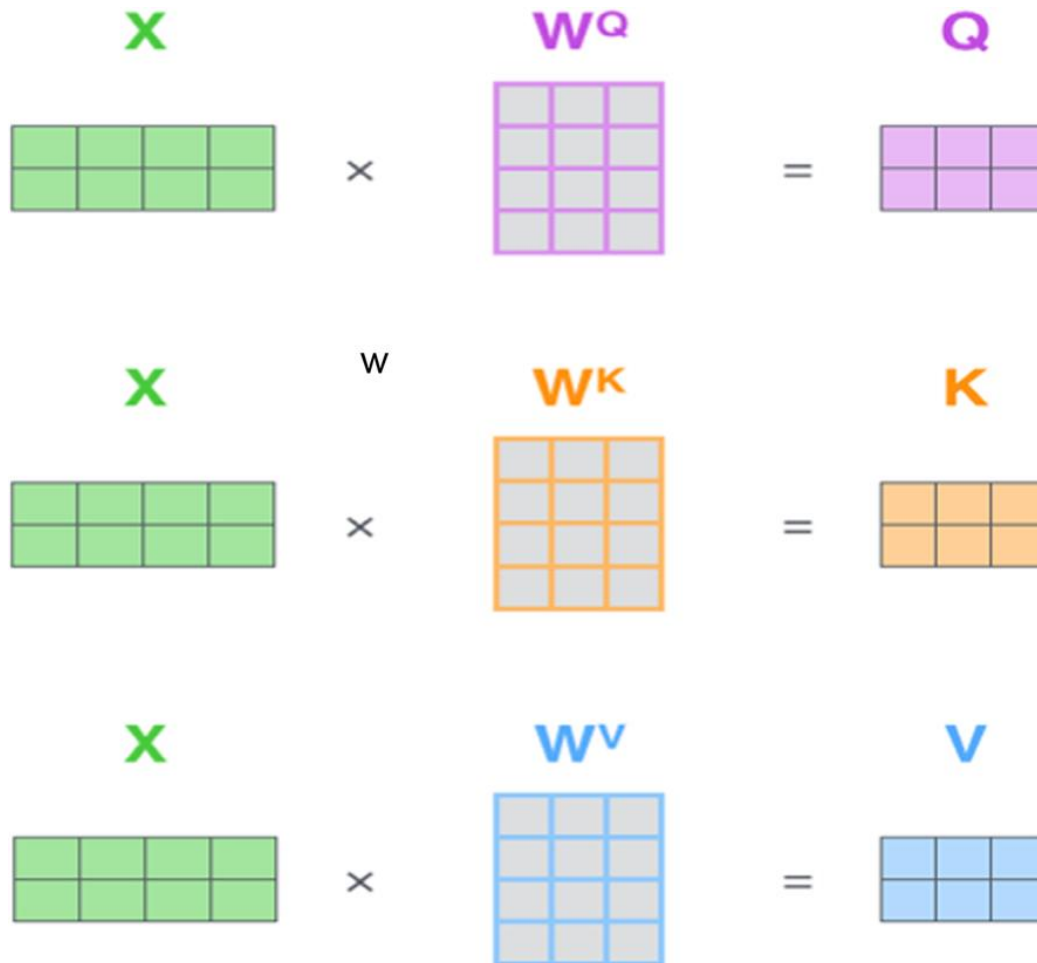


**FIGURE 2.6** Calculation methods for the  $Q$ ,  $K$ , and  $V$  matrices.

In the first example, matrix  $X$  is multiplied by matrix  $W^Q$  to result in matrix  $Q$ . In the second example, matrix  $X$  is multiplied by matrix  $W^K$  to result in matrix  $K$ . In the third example, matrix  $X$  is multiplied by matrix  $W^V$  to result in matrix  $V$ .

### 2.2.3 The advantages and limitations of the self-attention mechanism

Self-attention is a key component of the Transformer model, focusing on calculating attention weights among elements within the input sequence to highlight essential information. Unlike traditional sequence models like RNNs and long short-term memory networks (LSTMs), which process sequences element by element, the attention mechanism can directly evaluate the interactions between any pair of elements, enhancing its ability to handle long-range dependencies. The visibility of attention weights also adds a layer of interpretability: by examining these weights, one can discern the focal points of the model when dealing with specific elements, aiding in model analysis, debugging, and optimization. Furthermore, self-attention can be computed in parallel, making it highly effective on parallel processing units like GPUs. This contrasts sharply with RNNs and LSTMs, whose sequential operations do not fully leverage parallel hardware capabilities.

Self-attention also encounters certain limitations, particularly with processing long sequences. While it excels at capturing long-distance dependencies, the computational and memory demands can become overwhelming with very long sequences. To mitigate this, researchers have developed various optimization techniques such as hierarchical, sparse, and low-rank attention, which help reduce both computational load and memory usage. Additionally, self-attention models typically require substantial amounts of training data to perform optimally,

which can restrict their effectiveness in scenarios with limited data or in specialized domains.

## **2.3 Multihead attention mechanism**

The multihead attention mechanism is a sophisticated expansion of the self-attention system, designed to let a model simultaneously process and understand multiple aspects of the information presented in an input sequence. This capability allows for the creation of more nuanced and varied interpretations, significantly boosting the model's effectiveness.

With its ability to grasp a range of information, the multihead attention mechanism uses distinct “attention heads” to focus on different facets of the input. For instance, in translating languages, one head might analyze grammar, while another delves into the meanings behind words. This approach not only yields translations that are both accurate and fluid but also enriches the model's understanding.

Furthermore, by adding more attention heads, the mechanism does not just enhance the model's capacity for dealing with complex tasks—it also increases its overall expressiveness without needing more layers. In addition, the structure of multihead attention is inherently suited for parallel processing, making it exceptionally efficient on systems with advanced hardware like multiple CPUs or GPUs.

### 2.3.1 Implementation of multihead attention

**Fig. 2.7** illustrates the inner workings of the multihead attention mechanism. To set this system up, the query  $w^q$ , key  $w^k$ , and value  $w^v$  matrices from the traditional self-attention system are divided, with each section aligning with an individual attention head. Each head then processes self-attention independently using its designated subset of weights, ensuring that each head's queries, keys, and values are distinctly produced.

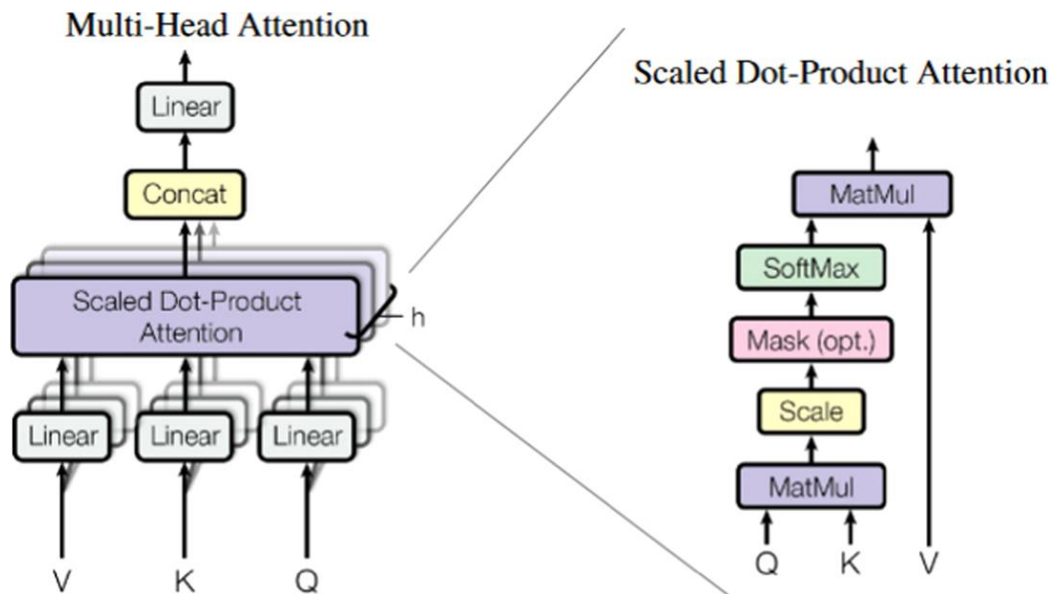


**FIGURE 2.7** Multihead attention.

It is divided into two main parts: Multi-Head Attention and Scaled Dot-Product Attention. On the left, under Multi-Head Attention, there are multiple Linear blocks feeding into a Scaled Dot-Product Attention block, which then connects to a Concat block and another Linear block. The inputs are labeled as  $V$ ,  $K$  and  $Q$ . On the right, under Scaled Dot-Product Attention, the process starts with MatMul, followed by Scale, Mask (opt.), SoftMax and another MatMul block. The inputs are labeled as  $Q$ ,  $K$  and  $V$ .

The process involves calculating the dot products of the query and key matrices, scaling these products, applying a softmax function for normalization, and then multiplying the result with the value matrix to produce the output for each head. This methodology enables each head to latch onto various slices of information from the input sequence.



However, more heads do not always mean better results. The foundational Transformer study suggests that eight heads are optimal. Increasing to 16 or 32 heads doesn't necessarily improve performance and may, in fact, lead to diminishing returns, especially when compared to having too few heads, like 1 or 4, which can degrade the model's effectiveness.

Once all heads have processed their inputs, the resulting vectors are merged into a larger vector, as depicted in **Fig. 2.8**. For instance, with eight heads, each produces an output vector, and these are then joined together. This combined output then passes through a linear layer (marked as " $W$ " in **Fig. 2.8**), which readjusts the dimensions back to match the original input while blending the information from all heads into a cohesive whole. This shared linear layer plays a crucial role in harmonizing the diverse insights gathered by the individual attention heads.

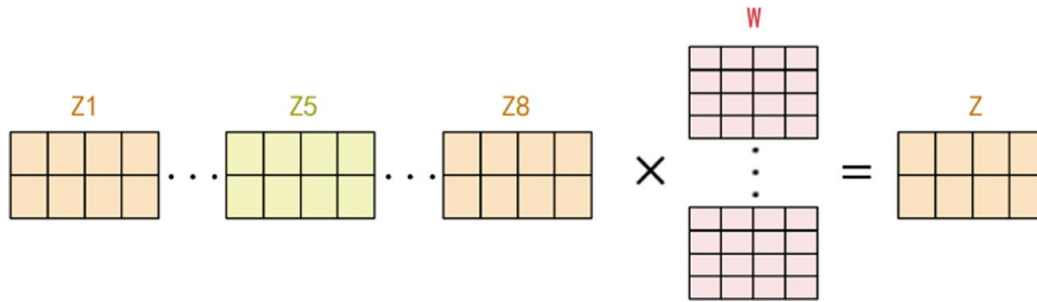


**FIGURE 2.8** Weighted fusion of multihead attention.

The image shows an illustration of merging vectors Z1, Z5, Z8 into a larger vector Z. The vectors Z1, Z5 and Z8 are depicted as matrices. These matrices are multiplied by matrix W, resulting in the larger vector Z.

### 2.3.2 The role of multihead attention

In the Transformer architecture, the multihead attention layer's output isn't just used for subsequent layers—it helps build the complex structure of the model itself. Each head within this layer zeroes in on a unique aspect of the input sequence, enhancing the model's depth of analysis. Studies on Sparse Transformers have shed light on how each head plays a specialized role, focusing on aspects such as syntax, contextual relationships, and the emphasis on rare words. For instance, the syntax-focused head blocks grammatically incorrect words from influencing the outcome, while the context-focused head parses the sentence, paying close attention to nearby words. Meanwhile, the head that hones in on rare words picks out key elements of sentences.

**Fig. 2.9** offers a visual demonstration of how different heads within an encoder influence the attention matrix, with this particular study focusing on an encoder configuration composed of four layers, each labeled from 0 to 3, and each containing six heads, labeled from 0 to 5. In this visualization, each row correlates with one of the encoder layers, and each column corresponds to one of the heads within these layers.

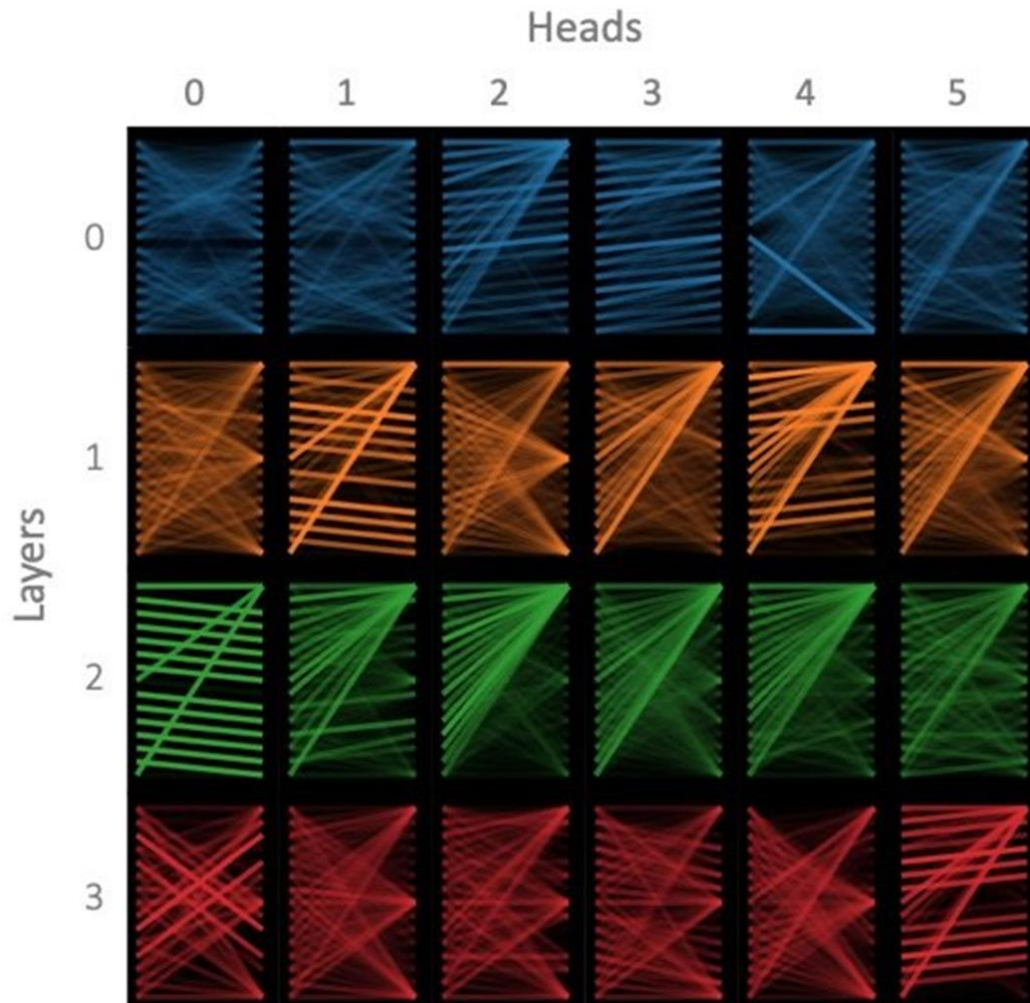


**FIGURE 2.9** Visualization of multihead attention. Image source: Paper “A multiscale visualization of attention in the transformer model”.

The visual demonstration of how different heads within an encoder influence the attention matrix, with this particular study focusing on an encoder configuration composed of four layers, each labeled from 0 to 3 and each containing six heads, labeled from 0 to 5. The image shows a grid with layers on the y-axis and heads on the x-axis. Each cell in the

grid contains a pattern representing the attention matrix influenced by the corresponding head and layer.

The diagram illustrates the dynamics within the layers: while some heads tend to align and focus on similar features—like heads 1 through 4 in both the second and third layers—others exhibit distinct and independent behaviors. For example, in the second layer, head 1, and in the third layer, heads 0 and 5, stand out for their unique choices, indicating a divergence in how they process and prioritize information. This diversity among heads highlights the nuanced capabilities of multihead attention in capturing a broad spectrum of features in the input sequence.

The multihead attention mechanism illustrates that, despite sharing the same method for residual gradient backpropagation, different attention heads manage to develop unique ways of selecting features. This diversity in feature selection could stem from variations in initial configurations and the use of Dropout strategies. Studies have specifically examined how different initialization strategies can decrease variability between layers, leading to more effective training results. Moreover, it has been suggested that employing distinct Dropout strategies for each head can allow them to capture and learn varied features, enhancing the overall functionality of the model.

### **2.3.3 Optimization of multihead attention**

Given the substantial number of parameters in multihead attention, each with its distinct “Q,” “K,” and “V” matrices, the model faces considerable computational and memory challenges. Researchers have developed several strategies to mitigate these issues:

1. **Local multihead attention:** This strategy limits each attention head to focus only on a specific portion of the input sequence rather than the entire sequence. This approach significantly reduces computational demands and memory usage while retaining the key benefits of multihead attention.
2. **Sparse multihead attention:** This technique reduces both computational complexity and memory usage by having each attention head focus only on select elements of the input sequence. Employing sparse connections can dramatically decrease the resources needed.
3. **Layered multihead attention:** In this structure, attention heads are organized into multiple layers, with each layer focusing on different sections and types of information from the input. This arrangement allows the model to grasp more complex contextual details and enhances performance, especially in processing lengthy sequences.

## 2.4 Feedforward neural network

The FNN, a pivotal component of the Transformer model, may constitute only a fraction of the model's structure, yet it is essential for extracting nonlinear features and capturing local details from input sequences. As illustrated in **Fig. 2.10**, the FNN fundamentally operates as a multilayer perceptron, composed of several layers, each populated with numerous neurons. In this setup, the input layer takes in the raw data, the hidden layers process this data to extract features, and the output layer is tasked with producing the final predictions. The connections among these layers are directional and form a loop-free path, ensuring that information flows through the network in one direction only.

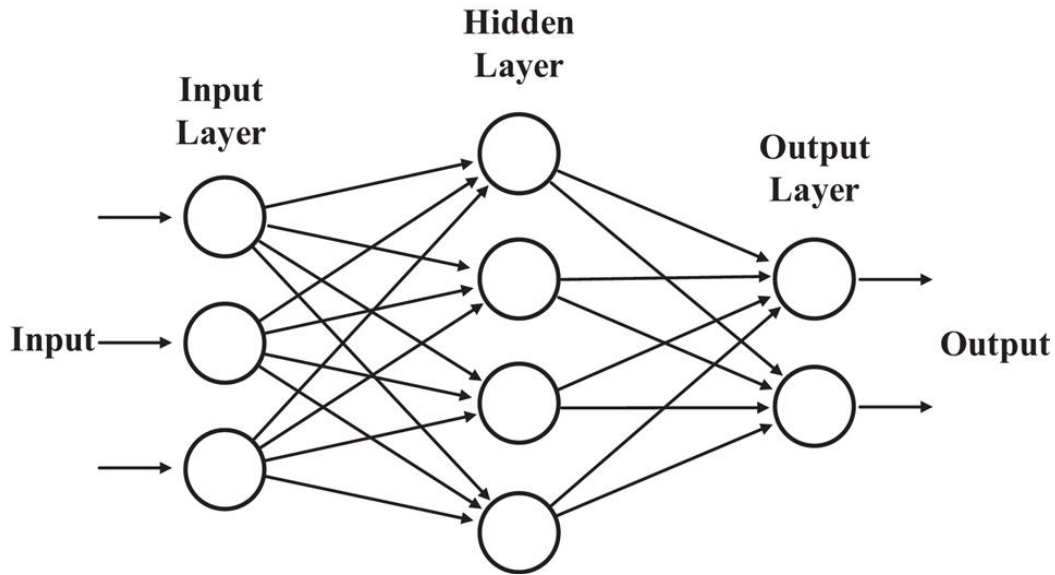


**FIGURE 2.10** Feedforward neural network.

It consists of three layers: an input layer, a hidden layer and an output layer. The input layer has three neurons labeled as Input, each receiving an input signal. The hidden layer contains three neurons, each connected to all neurons in the input layer. The output layer has three neurons, each connected to all neurons in the hidden layer. Arrows indicate the direction of signal flow from the input layer to the hidden layer and from the hidden layer to the output layer. The output layer neurons are labeled as Output.

In the Transformer model, the FNN is fundamentally composed of two linear layers and a nonlinear activation function. This design allows the network to capture nonlinear characteristics within the input sequence, significantly boosting the model's ability to express complex ideas. The first linear layer, often referred to as



the fully connected layer, projects the input data into a higher dimensional space. The subsequent layer, the output layer, then brings this high-dimensional representation back down to the original dimension. Positioned between these layers, the activation function injects nonlinearity into the process, enabling the network to discern intricate patterns in the data.

Activation functions are vital in FNNs as they convert the outputs of linear transformations into nonlinear outputs, adding necessary complexity to the model. The effectiveness of the network often hinges on the specific activation function used. Popular choices include Sigmoid, Tanh, and ReLU (Rectified Linear Unit), with the Transformer model frequently employing the GELU (Gaussian Error Linear Unit) function. GELU is mathematically expressed as

$$\text{GELU}(x) = 0.5x \left( 1 + \tanh \left( \sqrt{\frac{2}{\pi}} (x + 0.044715x^3) \right) \right). \quad (2.3)$$

As depicted in **Fig. 2.11**, GELU behaves almost linearly when inputs are around zero but tapers off towards zero with larger or smaller inputs. This property makes GELU particularly effective in deep neural network training, helping to prevent gradients from vanishing.

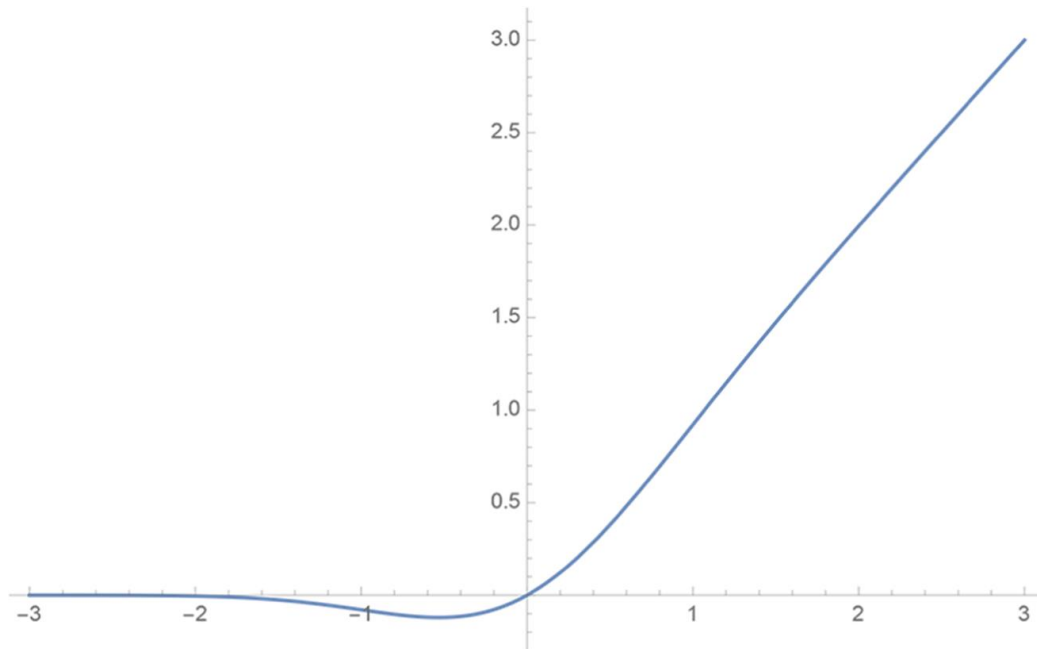


**FIGURE 2.11** GELU activation function.

The x-axis ranges from negative 3 to 3 in increments of 1. The y-axis ranges from negative 0.5 to 3.0 in increments of 0.5. The curve starts in the negative x quadrant slightly below the x-axis. It descends gradually, reaching its lowest point around x equals negative 0.5, where it's approximately -0.1 on the y-axis. From there, the curve begins to rise. As the curve enters the positive x quadrant, it increases more steeply. Around x equals 0.5, the curve crosses the x-axis and continues to rise. The slope becomes increasingly steep as x increases. By x equals 1, the y-value is approximately 0.5. At x equals 2, y is about 2.0. The curve continues its upward trajectory, reaching about y equals 3.0 at x equals 3. The shape of the curve resembles that of an activation function used in machine learning,

specifically the Gaussian Error Linear Unit (GELU) function, as mentioned in the provided context information.

In a FNN, each connection is assigned a weight that determines how strongly an input signal is passed to the next layer. These weights are fine-tuned through gradient descent during training to minimize prediction errors. Additionally, each neuron has a bias value that adjusts its activation threshold, which is similarly tuned during training to optimize network performance.

Despite its relatively small footprint within the Transformer model, the FNN is pivotal in capturing local information and unlocking nonlinear features. Its contributions to the Transformer model are notably significant in several areas:

1. **Local information extraction:** Positioned within each Transformer layer, the FNN processes outputs from the self-attention mechanism, adeptly extracting local features. This functionality enables the model to grasp short-range dependencies within the input sequence, substantially boosting its expressive power.
2. **Nonlinearity introduction:** Equipped with nonlinear activation functions, the FNN adeptly identifies nonlinear features within the input data. This capability is crucial for handling complex natural language sequences, which often contain inherently nonlinear patterns and structures.

3. **Generalization enhancement:** By harnessing both local and nonlinear features, the FNN significantly enhances the model's ability to generalize effectively when encountering new and previously unseen data.

## 2.5 Residual connection

Residual connections, a type of shortcut linkage in neural networks, allow for the direct transmission of layer inputs to later stages. This architectural feature is crucial for alleviating the vanishing gradient issue, thereby boosting both the training process and the overall model efficacy. These connections were pioneered by Kaiming He and colleagues in 2015 within the framework of deep residual networks to tackle the prevalent problems of vanishing and exploding gradients during deep neural network training. Within the Transformer architecture, residual connections are extensively employed, enhancing the multihead attention mechanisms, position-wise FNNs, and the various sublayers of the decoder.

**Fig. 2.11** depicts the core concept of residual connections, which involve introducing a shortcut in a neural network layer that effectively skips over certain layers. This design allows information from a preceding layer to bypass others and proceed directly to further layers. In mathematical terms, if  $x$  is the input to a layer, and it undergoes a nonlinear transformation denoted by  $F(x)$ , the output becomes  $F(x) + x$ . Here,  $x$  is the input signal, and  $F(x)$  is the transformation applied by the layer.

Adding  $x$  to  $F(x)$  results in combining the original input with the residual changes learned by the layer. This configuration is particularly beneficial during the backpropagation process as it facilitates smoother gradient flow, effectively addressing the issue of vanishing gradients and thus improving the training process (**Fig. 2.12**).

## Residual Learning

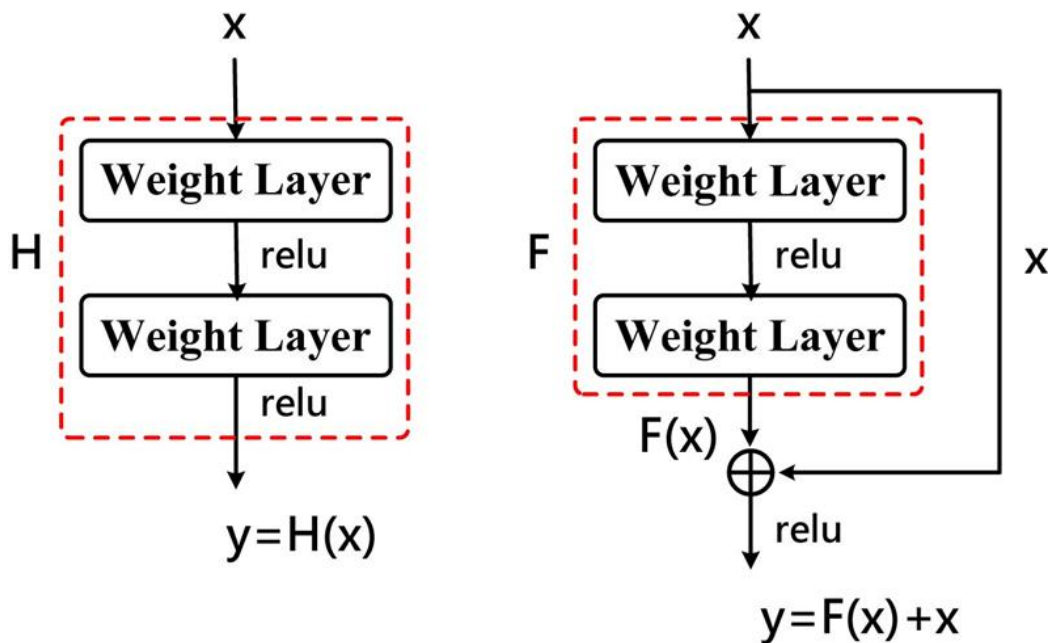


**FIGURE 2.12** Residual connection.

The left diagram shows a standard learning block with an input  $x$ , two weight layers each followed by a ReLU activation and an output  $y$  equals  $H(x)$ . The right diagram shows a residual learning block with an input  $x$ , two weight layers each followed by a ReLU activation, an intermediate output  $F(x)$  and a final output  $y$  equals  $F(x)$  plus  $x$ . The residual learning block includes a shortcut connection that adds the input  $x$  to the intermediate output  $F(x)$  before the final ReLU activation.

Residual connections are a key feature in the Transformer model, widely applied across its components to speed up training and boost performance. In the multihead attention sublayer, once the input undergoes

self-attention processing, the resulting output is combined back with the original input through a residual connection. This setup ensures that the model maintains the initial information while seamlessly integrating it with the insights derived from the self-attention.

Similarly, in the FNN sublayer, the input goes through two stages of linear transformations followed by an activation function. The output from this sequence is then added back to the original input, establishing another residual connection. This arrangement preserves the original input data and melds it with the new information learned through the FNN, fostering a richer understanding and better retention of input features.

In both the encoder and decoder components of the Transformer, each layer includes one or more sublayers equipped with residual connections. These connections are crucial for allowing information to flow effectively through the depth of the network, ultimately contributing to improved performance of the model.

In summary, the Transformer model incorporates residual connections across its multihead attention mechanisms, position-wise FNNs, and both encoder and decoder layers. These connections effectively address the challenges of gradient vanishing and exploding by offering a shortcut that allows for direct gradient flow to earlier layers. This feature is crucial for preventing common issues associated with training deep models, thereby boosting overall model performance. Residual connections also

expedite the training process by eliminating unnecessary calculations, streamlining the training duration. Moreover, by enabling the learning of more complex representations, residual connections enhance the model's ability to generalize, improving its performance on test datasets.

## 2.6 Layer normalization

Layer Normalization is a proven effective strategy for stabilizing neural network training and enhancing model performance. Introduced by Jimmy Ba et al. in 2016, it is extensively utilized in Transformer models. The core process involves normalizing the outputs of each layer's neurons to have consistent means and variances.

This normalization works by first calculating the mean and standard deviation across all outputs of a layer's neurons. Each neuron's output is then adjusted by subtracting the mean and dividing by the standard deviation. This results in normalized outputs, described by the equation:

$$y_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}. \quad (2.4)$$

Here,  $x_i$  represents the output from the  $i$ -th neuron,  $\mu$  and  $\sigma^2$  are the computed mean and variance, and  $\epsilon$  is a small positive number added for numerical stability. With zero mean and unit variance, these normalized outputs significantly expedite the training process and enhance the model's efficiency.



In the Transformer model, layer normalization plays a pivotal role in optimizing the training process. By normalizing neuron outputs, it accelerates the convergence of gradient descent, enhances training efficiency, reduces covariance shifts between layers, and prevents gradient explosions or disappearances. This normalization significantly reduces the model's sensitivity to variations in input data, improving its performance on new datasets.

Layer normalization is strategically deployed across the Transformer architecture:

1. Multihead attention mechanism: It is used to normalize attention scores, improving the efficacy of the attention process.
2. Position-based feedforward networks: Here, it normalizes the outputs of neurons, facilitating faster training and boosting overall model performance.
3. Encoder-decoder architecture: Layer normalization is applied both to each sublayer's output and to the final outputs of the encoder and decoder, stabilizing training and enhancing generalization capabilities.
4. Residual connections: It works alongside addition operations to ensure outputs maintain consistent means and variances postaddition, further enhancing performance and training speed.

Collectively, residual connections and layer normalization are critical to the Transformer's architecture. Residual connections introduce skip links that alleviate gradient

vanishing, boost training efficiency, and strengthen generalization. Layer normalization, by standardizing outputs within layers, speeds up training and stabilizes the model. These technologies are integral to the multihead attention, position-based feedforward networks, and the sublayers of the encoder and decoder, underpinning the Transformer's robust performance.

## **2.7 Position encoding**

Position encoding is a fundamental component of the Transformer model (as illustrated in **Fig. 2.13**), equipping the model with the capability to manage the positional information of words in sequences. Given that the self-attention mechanism lacks the ability to detect the positions of words, position encoding is crucial for understanding the order within sequences.

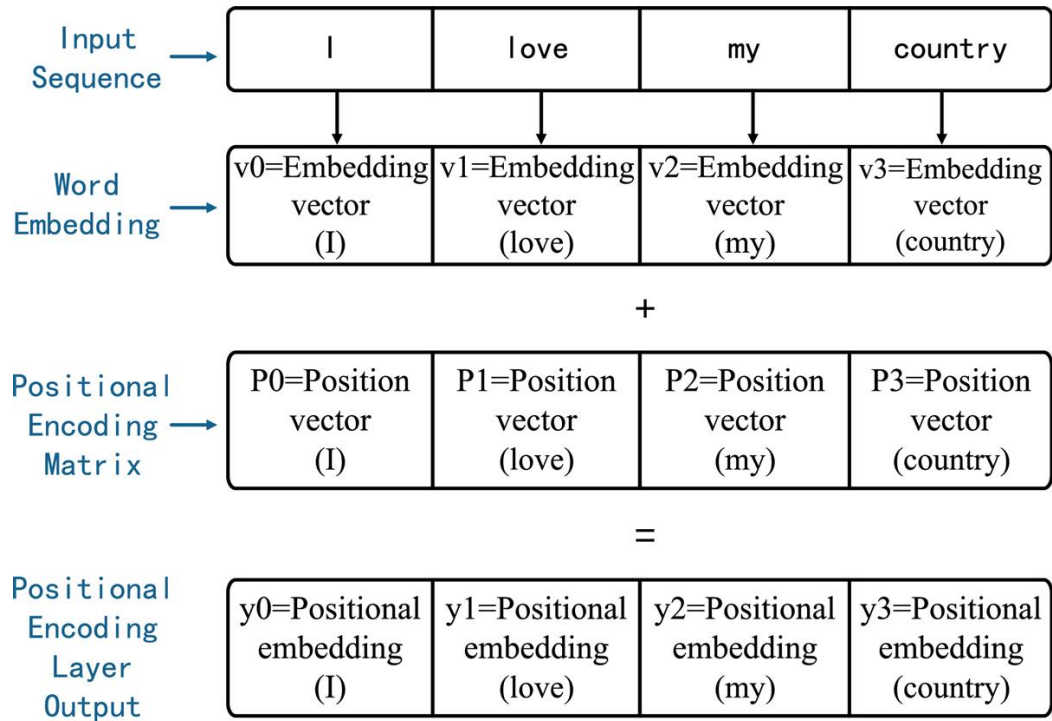


**FIGURE 2.13** Workflow of positional encoding.

It starts with the input sequence I love my country. Each word in the sequence is converted into an embedding vector: v0 equals embedding vector (I), v1 equals embedding vector (love), v2 equals embedding vector (my) and v3 equals embedding vector (country). These embedding vectors are then added to their corresponding position vectors from the positional encoding matrix: P0 equals position vector (I), P1 equals position vector (love), P2 equals position vector (my) and P3 equals position vector (country). The result is the positional encoding layer output: y0 equals positional embedding (I), y1 equals positional embedding (love), y2 equals positional

embedding (my) and y3 equals positional embedding (country).

### 2.7.1 Design and implementation of positional encoding

The self-attention mechanism leverages matrix multiplication to assess relationships between words, ignoring their sequential positions—an essential factor in processing ordered data such as text. To address this, position encoding represents each word's position in the input sequence and integrates this data with the original word representations. Position encoding in Transformers is produced using predefined sine and cosine functions. The encoding for a given position  $p$  and dimension  $i$  is calculated as follows:

$$\text{PE}_{(p,2i)} = \sin\left(\frac{p}{10,000^{2i/d_{\text{model}}}}\right) \quad (2.5)$$

$$\text{PE}_{(p,2i+1)} = \cos\left(\frac{p}{10,000^{2i/d_{\text{model}}}}\right). \quad (2.6)$$

This setup, where  $d_{\text{model}}$  denotes the dimension of word embeddings, offers several benefits:

- The encoding varies in frequency across different positions and dimensions, aiding the model in discerning complex positional relationships.

- The cyclical nature of the sine and cosine functions allows the model to process longer sequences beyond the training data.
- Position encoding is adaptable, permitting the model to tailor the significance of positional information based on specific task needs.

To implement position encoding, a vector of dimension  $d_{\text{model}}$  is generated for each position using the equations above. These vectors are then added to the word embeddings of the input sequence, creating enriched word representations with embedded positional information. These enriched representations are then utilized in subsequent layers, including multihead attention and feedforward networks, enhancing the model's ability to interpret and process sequences.

Consider a sentence of sequence length 4, "I love my country," with each word depicted as a two-dimensional vector. This requires generating a  $4 \times 2$  matrix for position encoding. In Transformers, this encoding is derived from sine and cosine functions, creating distinct positional signals. In these formulas,  $i$  denotes the position within the sequence (starting at 0),  $j$  is the dimension index, and  $d_{\text{model}}$  is the dimensionality of the input vectors. For this example,  $d_{\text{model}}$  is 2. Using **Eqs. (2.5, 2.6)**, the position encoding matrix for this example, with dimensions  $4 \times 2$ , can be calculated as follows:

$$PE(0, 0) = \sin\left(\frac{0}{10,000 \wedge \left(\frac{0}{2}\right)}\right) = \sin(0) = 0$$

$$PE(0, 1) = \cos\left(\frac{0}{10,000 \wedge \left(\frac{0}{2}\right)}\right) = \cos(0) = 1$$

$$PE(1, 0) = \sin\left(\frac{1}{10,000 \wedge \left(\frac{0}{2}\right)}\right) = \sin(1) \approx 0.84$$

$$PE(1, 1) = \cos\left(\frac{1}{10,000 \wedge \left(\frac{0}{2}\right)}\right) = \cos(1) \approx 0.54$$

$$PE(2, 0) = \sin\left(\frac{2}{10,000 \wedge \left(\frac{0}{2}\right)}\right) = \sin(2) \approx 0.91$$

$$PE(2, 1) = \cos\left(\frac{2}{10,000 \wedge \left(\frac{0}{2}\right)}\right) = \cos(2) \approx -0.42$$

$$PE(3, 0) = \sin\left(\frac{3}{10,000 \wedge \left(\frac{0}{2}\right)}\right) = \sin(3) \approx 0.14$$

$$PE(3, 1) = \cos(3/10,000 \wedge (0/2)) = \cos(3) \approx -0.99.$$

These calculated values are compiled into the position encoding matrix as follows:

$$\begin{bmatrix} 0 & 1 \\ 0.84 & 0.54 \\ 0.91 & -0.42 \\ 0.14 & -0.99 \end{bmatrix}$$

Subsequently, this position encoding matrix can be added to the original word vectors. This addition enables the model to integrate positional information into its processing, as depicted in **Fig. 2.13**. This figure illustrates the process of calculating the position encoding matrix and its application to sequence data. It's important to note that this example is a simplified representation. Typically, in real-world applications, the dimensionality of  $d_{\text{model}}$  is considerably larger, often set to 512.

### 2.7.2 Variants of positional encoding

While the Transformer's original architecture relies on fixed sine and cosine functions for generating position encodings, researchers have developed alternative methods for conveying positional information:

1. **Learned position encoding:** This approach involves initializing position encodings with random values that are subsequently optimized through gradient descent during training. This method enables the model to dynamically refine its positional representations to meet the specific demands of the task.
2. **Relative position encoding:** Focusing on the relative distances between words within a sequence rather than their absolute positions, relative position

encoding aids the model in learning more universal positional patterns. This facilitates better adaptation to sequences of varying lengths and can be implemented by altering the self-attention mechanism to account for the relative distances between queries and keys.

3. **Combined position encoding:** This method merges various positional representation techniques, including fixed sine/cosine functions, learned position encodings, and relative position encodings. By integrating these diverse sources of positional information, the model can draw on a broader range of insights to improve its predictive capabilities.

### 2.7.3 The advantages and limitations of positional encoding

Position encoding significantly boosts the ability of neural networks to process sequence data by providing essential positional information in specific ways. This capability allows models to recognize and leverage the relationships between words, enhancing the handling of sequential data such as natural language. Thanks to the periodic properties of sine and cosine functions, position encoding efficiently manages long sequences and facilitates generalization beyond the lengths encountered during training.

Additionally, models can improve their adaptability by learning to adjust the importance of positional information based on the requirements of specific tasks.



However, it is important to note that for tasks not reliant on sequence order, introducing position encoding could add unnecessary computational complexity. Fixed sine and cosine position encodings might also fail to detect some complex positional patterns within sequences, particularly regarding local structures. When sequences exceed the maximum length processed during training, the model might struggle with effective generalization using position encoding. In these instances, relative position encoding or other expandable methods of position encoding might offer a more effective solution.

## **2.8 Training and optimization**

Training and optimizing a Transformer model involves selecting appropriate methods, such as loss functions and optimizers. This section explores training techniques and strategies for optimizing Transformers.

### **2.8.1 Loss function**

At the heart of training is defining a loss function that quantifies the discrepancy between the model's predictions and the true labels. The cross-entropy loss function is often used in Transformers. It computes a loss by comparing the probability distributions from the model's output against the actual labels. The training objective is to minimize this loss.

Cross-entropy measures differences between two probability distributions and is crucial in classification

tasks. In Transformers, the output layer converts scores into probabilities using the softmax function. Cross-entropy then assesses the discrepancies between these probabilities and the true distributions. Its formula is

$$H(p, q) = - \sum_{i=1}^n p(x_i) \log(q(x_i)) \quad (2.7)$$

where  $p$  is the actual distribution and  $q$  is the predicted one. Cross-entropy's main benefit is its direct measurement of the prediction accuracy against true labels, although it may not be as effective for less probable categories.

Alternative loss functions can be adopted based on the task's needs. For instance, regression tasks might use mean squared error or mean absolute error, while generative tasks could use negative log-likelihood or Kullback-Leibler (KL) divergence.

## 2.8.2 Optimizer

Minimizing the loss function requires selecting an appropriate optimizer to update the model parameters. In Transformers, Adam is a preferred choice. Adam, or Adaptive Moment Estimation, is an optimizer that automatically adjusts the learning rate based on both the first and second moment estimates of gradients. This feature allows Transformers to be optimized with varying learning rates at different training phases.

Adam combines the advantages of fast convergence and high computational efficiency, and is generally less

sensitive to the choice of hyperparameters. However, it can sometimes compromise generalization performance.

Other available optimizers include Stochastic Gradient Descent (SGD), AdaGrad, and RMSprop. SGD is straightforward but might be slow to converge in certain scenarios. AdaGrad and RMSprop, which adaptively adjust the learning rate based on the first and second moments, respectively, can lead to premature learning rate decay in prolonged training sessions. Each optimizer comes with its own set of pros and cons, necessitating a choice that best fits the specific requirements of the task and dataset.

### 2.8.3 Learning rate adjustment strategy

In the original Transformer paper, the authors introduced a novel approach to learning rate scheduling known as “Noam scheduling.” This method initially increases the learning rate linearly, followed by a gradual decrease, enhancing early training convergence and later preventing overfitting. The formula for this strategy is

$$\text{lr} = d_{\text{model}}^{-0.5} \cdot \min(\text{step\_num}^{-0.5}, \text{step\_num} \cdot \text{warmup\_steps}^{-1.5}) \quad (2.8)$$

where  $d_{\text{model}}$  is the model’s dimension,  $\text{step\_num}$  is the current step in training, and  $\text{warmup\_steps}$  is the number of initial steps before regular training pacing begins. This approach balances the model’s convergence speed and stability effectively.

Learning rate strategies are critical as they influence how quickly and effectively a model learns and generalizes. The

Noam schedule, specifically, promotes a faster initial convergence and helps in maintaining model stability, although it may require fine-tuning of warmup steps and other hyperparameters.

Other strategies like exponential decay, cosine annealing, and piecewise constant learning rates also exist.

Exponential decay reduces the learning rate steadily over time, while cosine annealing adjusts the learning rate cyclically, reflecting the cosine function's periodic nature. The piecewise constant strategy divides the training into phases, each with a consistent learning rate. Choosing the right strategy depends on the specific requirements of the task and the data characteristics.

### **2.8.4 Regularization**

To combat overfitting, a range of regularization techniques are utilized during the training process. In Transformers, weight decay (also known as L2 regularization), dropout, and layer normalization are prevalent strategies. Weight decay boosts generalization by incorporating the L2 norm of parameters into the loss function, which helps control the scale of the parameters. Nonetheless, this approach might sometimes overly penalize certain parameters.

As depicted in **Fig. 2.14**, dropout is a technique used during training to randomly deactivate a subset of neurons, effectively reducing the model's tendency to overfit to particular training examples and thus improving its ability

to generalize. However, one downside of dropout is that it can slow down the model's convergence rate.

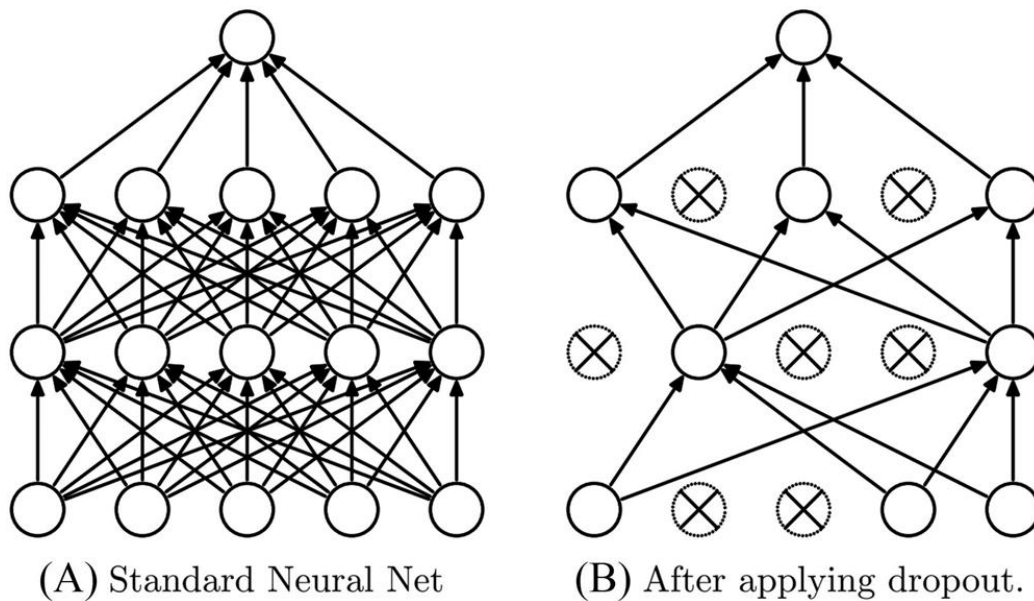


**FIGURE 2.14** Dropout regularization.

Diagram (A) is labeled Standard Neural Net and shows a fully connected neural network with three layers of neurons. Each neuron in one layer is connected to every neuron in the adjacent layers with arrows indicating the direction of connections. Diagram (B) is labeled After applying dropout and shows the same neural network with some neurons deactivated, indicated by circles with an X inside them. The connections to and from these deactivated neurons are also removed.

Layer normalization standardizes the activations within each layer of a neural network, speeding up training and

enhancing generalization. However, it can sometimes adversely affect specific tasks.

Moreover, there are additional regularization methods like L1 regularization, data augmentation, and early stopping. L1 regularization incorporates the L1 norm of parameters into the loss function, creating a sparse matrix of parameters, though this might sometimes lead to excessive penalties on certain parameters. Data augmentation improves generalization by variably transforming training data to enlarge the dataset, but this can extend the training duration. Early stopping, which halts training when no significant improvements are seen on the validation set, helps prevent overfitting but may result in an underfit model.

### **2.8.5 Other training and optimization techniques**

Training and optimizing Transformers involve various additional techniques such as learning rate warmup, gradient clipping, weight initialization, dropout, label smoothing, gradient accumulation, and mixed precision training, which will be elaborated upon next.

The learning rate warmup strategy starts with a low learning rate at the onset of training and gradually increases to a preset maximum. This approach is designed to prevent the instability that can occur from large updates to parameters early in training. Typically, Transformers employ a linear warmup strategy where the learning rate

progressively increases from zero to the established maximum before reducing according to a specific scheme like inverse square root decay, as illustrated in **Fig. 2.15**.

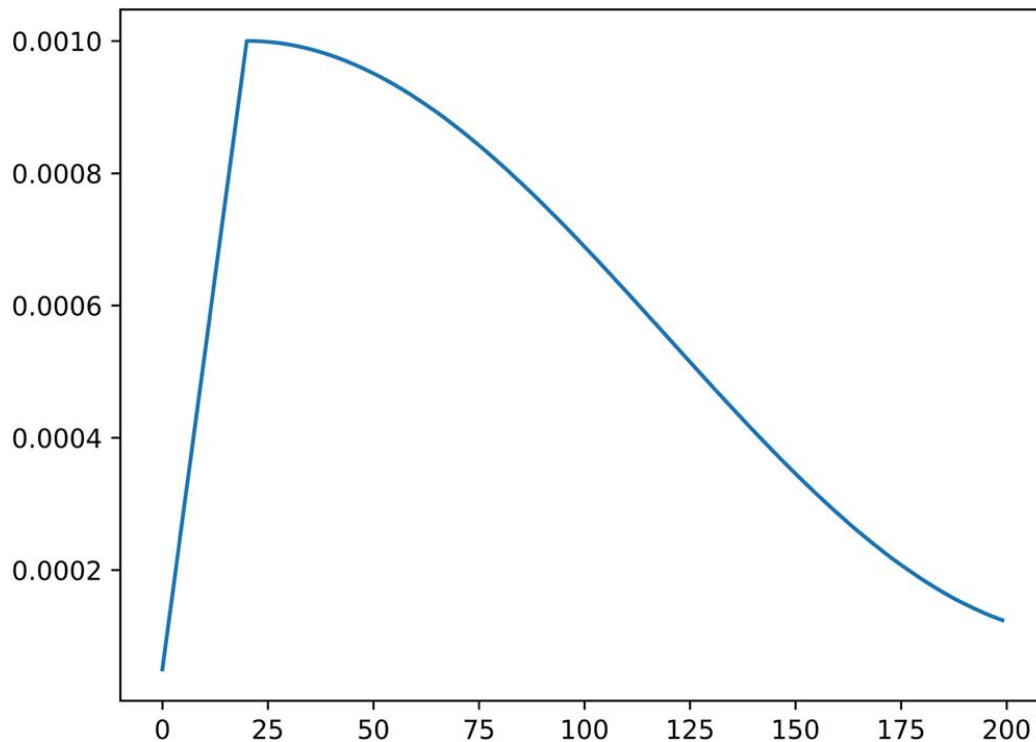


**FIGURE 2.15** Learning rate warmup strategy.

The x-axis represents time, ranging from 0 to 200 in increments of 25. The y-axis represents the learning rate, ranging from 0 to 0.0010 in increments of 0.0002. The line starts near zero and rapidly increases to a peak of about 0.0010 at around time 25. This represents the warmup phase where the learning rate increases from zero to a maximum value. After reaching the peak, the line gradually decreases in a curved manner, illustrating the decay phase. The decay follows what appears to be an inverse square root pattern, as mentioned in the context information. By the end of the graph at time 200, the learning rate has decreased to approximately 0.0001, which is about 10 percent of its peak value. This graph visually represents the



typical learning rate strategy employed in Transformer models, with an initial linear warmup followed by a decay phase, as described in the context information provided.

Gradient clipping is a technique employed to counteract the issue of gradient explosion. In neural network training, if gradients become too large, they can destabilize the model by causing disproportionately large updates to weights. By establishing a maximum limit for gradient values, these excessive updates can be curbed. Should gradients exceed this limit, they are scaled back to match the established threshold.

Weight initialization plays a critical role in both the speed of training and the ultimate performance of the model. Transformer models typically use random initialization techniques with either uniform or normal distributions, such as Glorot (also known as Xavier) or He initialization.

Dropout is another potent regularization strategy used to combat overfitting. It works by randomly disabling a subset of neurons during training, which is particularly common in the Transformer's multihead attention and feedforward layers.

Label smoothing is a technique to combat overfitting by blending true labels with a controlled amount of noise, thereby boosting the model's generalization abilities.

Gradient accumulation collects gradients from several batches before updating the model's parameters. This method effectively mimics training with larger batch sizes,

enhancing both the stability and the speed of convergence of the model, while not requiring additional memory resources.

Mixed precision training combines single precision (float32) and half precision (float16) computations to reduce memory demands and computational load, thus speeding up the training process. As demonstrated in the comparative experiment shown in **Fig. 2.16**, with identical hyperparameters but different data types (float32 (FT32) or mixed precision), the training loss curves (left image) are nearly the same, yet the training speed of the mixed precision model is noticeably quicker (right image).

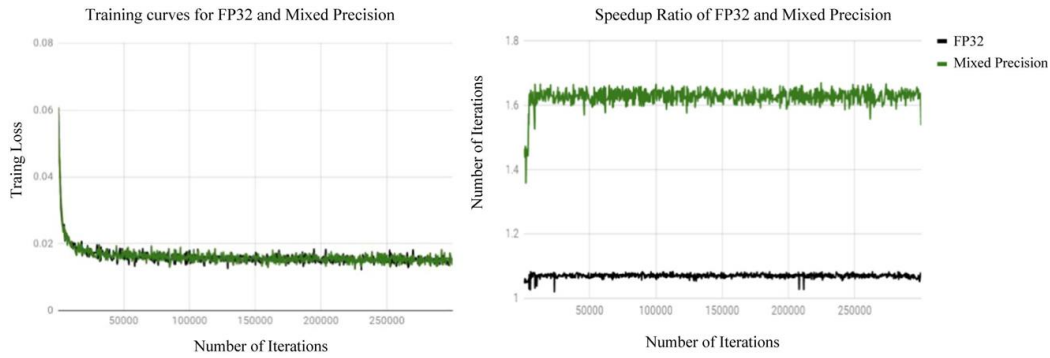


**FIGURE 2.16** Comparison between mixed precision acceleration ratio and training loss.

The left graph is titled Training curves for FP32 and Mixed Precision. The x-axis represents the Number of Iterations ranging from 0 to 250000 in increments of 50000. The y-axis shows Training Loss with values from 0 to 0.08 in increments of 0.02. Two lines are plotted: one for FP32 and one for Mixed Precision. Both lines start high and quickly drop, then stabilize at a low value, with the Mixed Precision line slightly lower than the FP32 line. The right graph is titled Speedup Ratio of FP32 and Mixed Precision. It shares the same x-axis as the left graph, representing Number of Iterations from 0 to 250000. The y-axis shows Number of Iterations (which appears to be mislabeled and likely represents the speedup ratio) ranging from 0 to 1.8 in increments of 0.2. Two lines are plotted, labeled as FP32 and Mixed Precision in the legend. The Mixed Precision line remains consistently higher, around 1.6, while the FP32 line stays close to 0.2 throughout the iterations. Both graphs demonstrate a comparison between FP32 and

Mixed Precision methods across training iterations, with Mixed Precision showing lower training loss and higher speedup ratio.

These techniques are crucial in the training and optimization of Transformer models. Tailoring these methods to the specific needs of the task and dataset can significantly improve the training results and model performance. Generally, training and optimizing Transformer models involve selecting the right loss functions, optimizers, learning rate adjustments, and regularization techniques. For each task and dataset, selecting the appropriate settings and fine-tuning hyperparameters are essential to maximize model performance. Each method has its pros and cons in real-world applications, necessitating careful evaluation and choice.

## **2.9 Summary**

This chapter thoroughly explains the fundamental principles and applications of the Transformer model and its essential components. It starts with a succinct overview of the Transformer model, then dives deep into each critical component. The self-attention mechanism is covered extensively, detailing its principles, benefits, and limitations. The chapter also delves into multihead attention, discussing its rationale, detailed implementation, practical uses, and various enhancements and adaptations.

Future chapters will further explore the progression of large language models, examining enhancements to the Transformer model across different GPT versions and discussing its uses in areas like NLP, generative models, and reinforcement learning.

---

## CHAPTER 3

# Generative pretraining

---

### Abstract

This chapter delves into Generative Pretraining (GPT), a foundational technique in natural language processing (NLP) that underlies the GPT series of models. GPT pretrains on extensive unlabeled text data to learn a generative probability distribution, enabling it to predict subsequent words in a sequence based on prior context. The chapter explains the architecture of GPT, which is built on a Transformer decoder with masked self-attention mechanisms, allowing the model to focus solely on previous words during generation. The process of pretraining is followed by supervised fine-tuning, where the model is adapted for specific NLP tasks using labeled data. This dual-phase approach significantly enhances the model's performance across various applications, making GPT a powerful tool in language understanding and generation.

### Keywords

Generative pretraining; natural language processing; transformer decoder; masked self-attention; supervised fine-tuning; language model; unsupervised learning; text generation

Generative Pretraining (GPT) is a crucial technology in natural language processing (NLP) that also underpins the GPT series of models. The core concept revolves around pretraining on extensive text data to grasp the generative probability distribution of language models. These pretrained models are then utilized across various downstream tasks. This chapter offers an in-depth exploration of the principles and techniques behind GPT, equipping readers with a thorough understanding of the GPT model's technical nuances.

## **3.1 Introduction to generative pretraining**

Pretraining techniques have demonstrated remarkable effectiveness across many tasks in computer vision and natural language processing (NLP). These methods leverage extensive amounts of unlabeled data to uncover the hidden structures of the underlying data.

Generative Pretraining (GPT) aims to develop a language model focused on learning a generative probability distribution. This means it predicts the likelihood of the next word based on a given context sequence, such as words, phrases, or sentences. GPT uses a neural network, specifically a Transformer model equipped with self-

attention mechanisms, to model these probabilities. This allows it to capture the long-distance relationships in text sequences. As depicted in **Fig. 3.1**, at the heart of GPT is a multilayer Transformer encoder, each layer composed of a masked multihead attention sublayer and a feedforward neural network sublayer. Masked self-attention, a special type of self-attention, captures dependencies across different positions in the input sequence and ensures that the model during pretraining only considers information from previous words (this is done by introducing a masking matrix during self-attention calculations, which excludes subsequent words in the sequence).





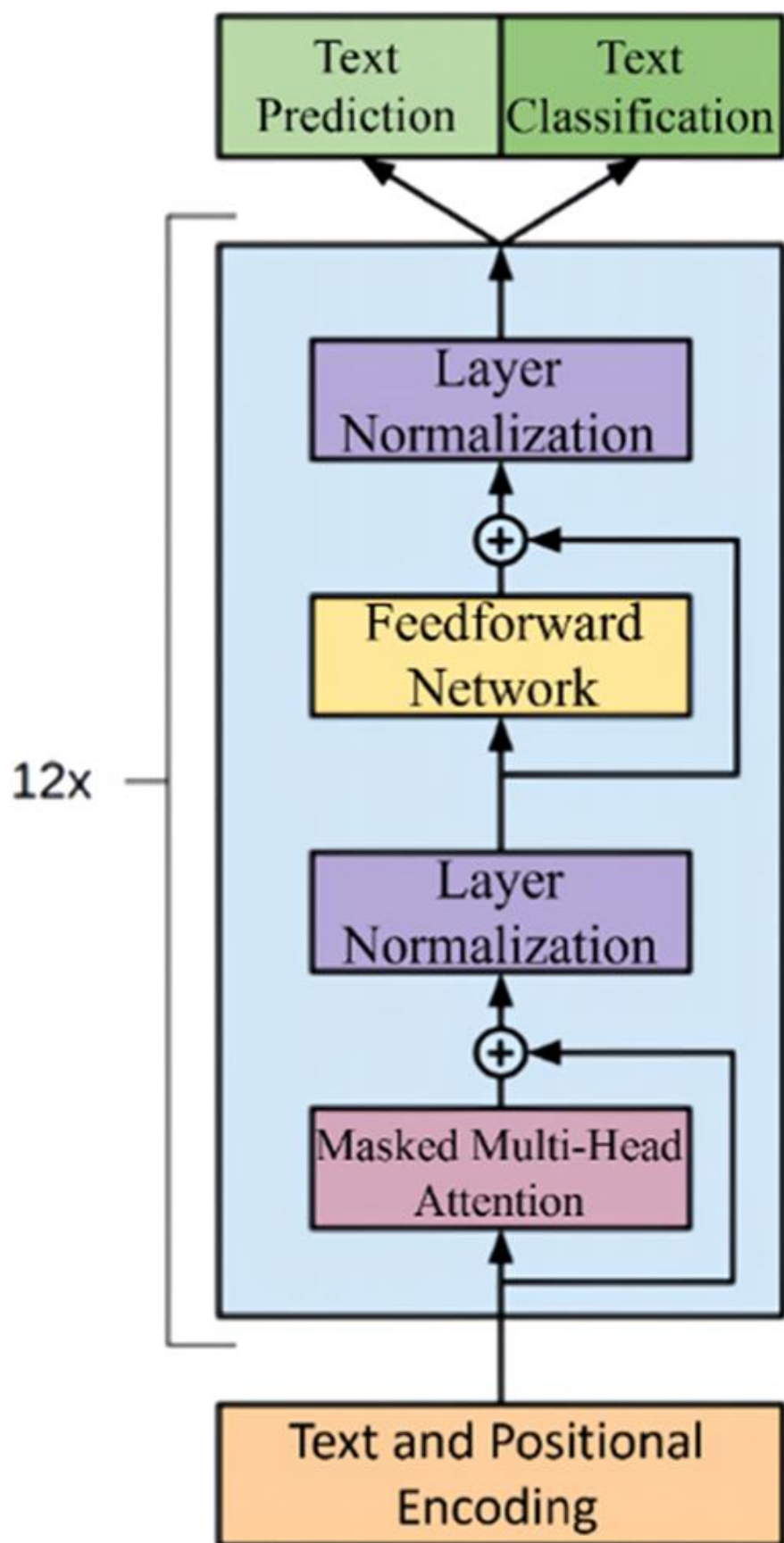


**FIGURE 3.1** Transformer decoder in GPT-1. *GPT*, Generative pretraining. Image source: Paper “Improving language understanding by generative pretraining”.

At the heart of GPT is a multilayer Transformer encoder, each layer composed of a masked multihead attention sublayer and a feedforward neural network sublayer. The diagram shows the flow starting from Text and Positional Encoding at the bottom, followed by Masked Multi-Head Attention, Layer Normalization, Feedforward Network and another Layer Normalization. This sequence is repeated 12 times. The output is directed towards Text Prediction and Text Classification.

In the training phase, GPT seeks to maximize the data likelihood by minimizing the cross-entropy loss, enhancing the chances of accurately predicting the next word. Optimization of this loss improves the precision of the generative probability distribution. Popular optimization techniques include Stochastic Gradient Descent (SGD) and variants like Adam and Adagrad. The model’s performance is heavily influenced by how the learning rate is adjusted and regularization techniques employed. Typically, strategies like warming up the learning rate and gradually reducing it help stabilize the training process.

## 3.2 Generative pretraining model

The foundation of the GPT model is the decoder architecture of the Transformer, which is renowned for its self-attention mechanism that effectively captures long-distance dependencies and boosts parallel computing power. As depicted in **Fig. 3.2**, the essence of this model is a series of Transformer decoder layers, each comprising a self-attention sublayer and a feedforward neural network sublayer. In this setup, the self-attention sublayer comes first, focusing on identifying dependencies across various positions in the input sequence, while the subsequent feedforward sublayer handles nonlinear transformations at each position.

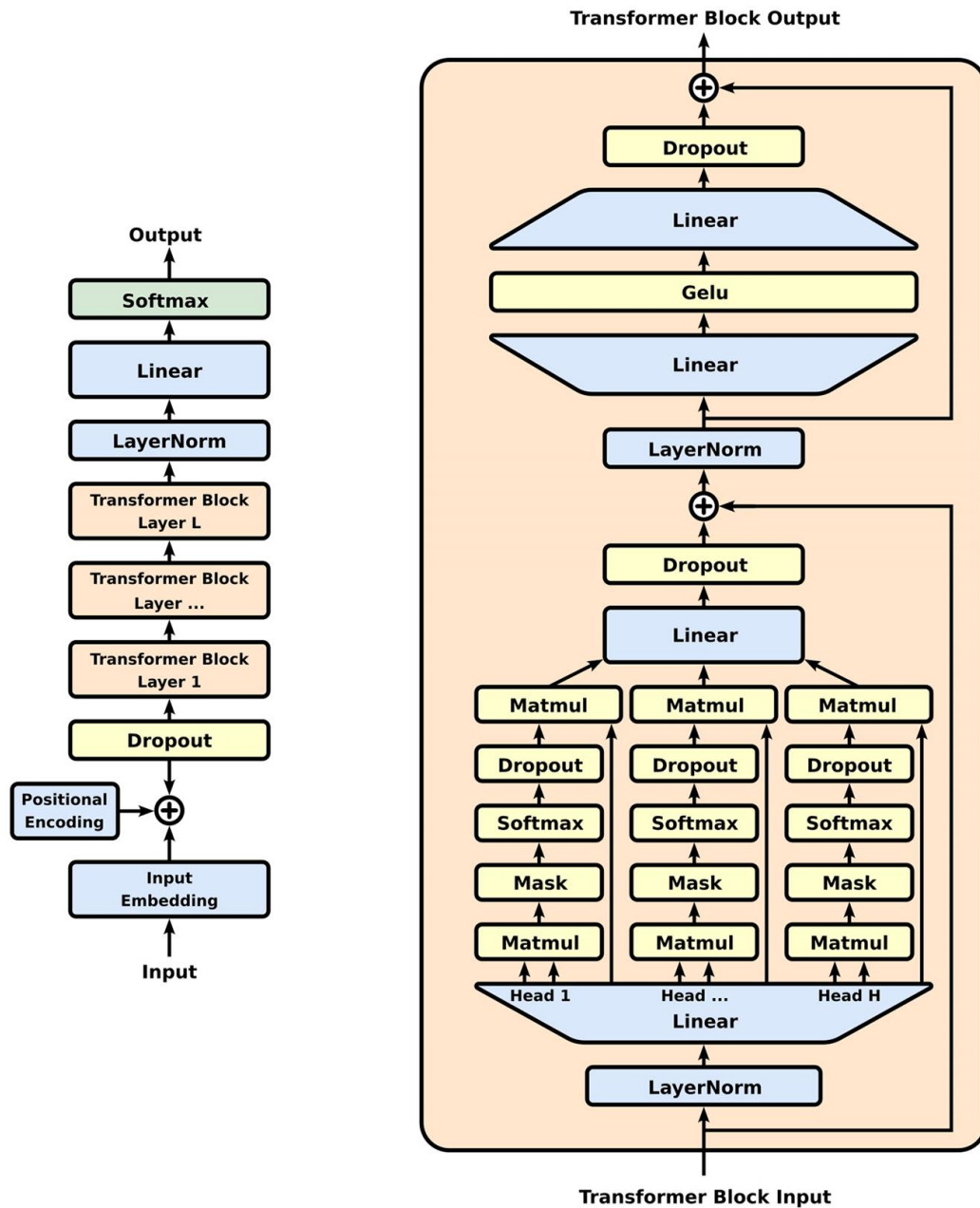


**FIGURE 3.2** Transformer model in GPT-1. Image source: <https://www.wikiwand.com/zh-hans/OpenAI>.

The diagram illustrates a series of Transformer decoder layers, each comprising a self-attention sublayer and a

feedforward neural network sublayer. The process starts with the input, which goes through Input Embedding and Positional Encoding, followed by Dropout. This is then fed into Transformer Block Layer 1, followed by subsequent Transformer Block Layers, up to Layer L. Each Transformer Block consists of a LayerNorm, followed by a self-attention mechanism with multiple heads (Head 1 to Head H), each involving Matmul, Mask, Softmax, Dropout and another Matmul. The outputs of these heads are combined through a Linear layer. This is followed by another LayerNorm, a Linear layer, Gelu activation, another Linear layer, Dropout and a residual connection. The final output of the Transformer Block is passed through another LayerNorm, a Linear layer and Softmax to produce the final output.

To ensure that the model generates text based solely on the context it has seen and not on any future information, GPT uses a technique called masked self-attention. In practice, this means that during the self-attention scoring process, the model ignores any words that appear after the current position in the sequence. This is accomplished by integrating a mask matrix into the self-attention calculation, which guarantees that only preceding information is taken into account.

The core principle of masked self-attention is that the model's focus is restricted to the current and preceding positions when calculating self-attention weights. For instance, in an input sequence  $x=(x_1, x_2, \dots, x_n)$ , the self-attention

weight at position  $i$  considers only the elements from position 1 to  $i$ . This approach ensures that the text generated is informed only by the context already observed and not by any subsequent information. To achieve this, the attention matrix used in self-attention calculations is modified by masking, where weights for future positions are set to negative infinity. When processed through a softmax function, these weights effectively become zero, thereby enforcing causal masking.

The masked self-attention mechanism is crucial in GPT tasks. As illustrated in **Fig. 3.3**, during the pretraining phase, the model is tasked with learning a broad representation of language. Masked self-attention ensures that the model concentrates solely on the current position and the preceding context when generating text, aiding in adherence to the natural patterns of language generation. During the fine-tuning phase, this mechanism continues to be essential, helping the model produce outputs that align with the previously observed context.

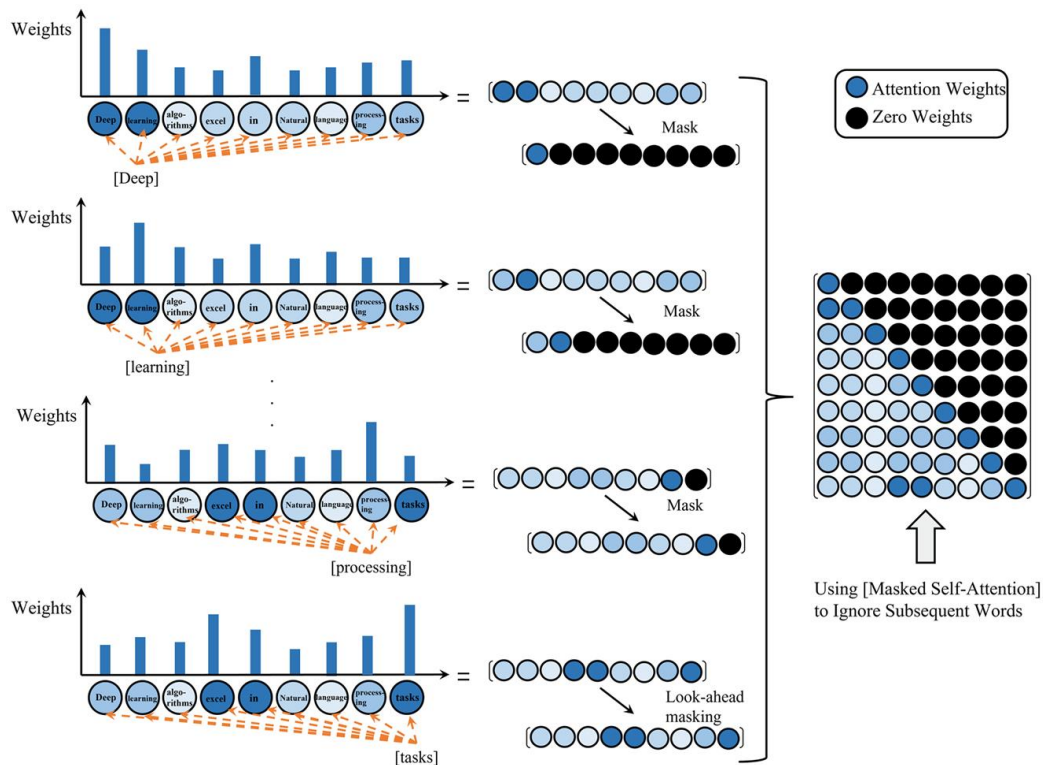


**FIGURE 3.3** Working principle of masked self-attention. Image source: <https://data-science-blog.com/blog/tag/engineer/>.

It shows 4 stages of weights and masking for the words: Deep, learning, algorithms, excel, in, Natural, language, processing, tasks. Each stage has a graph of weights, followed by a masked sequence. The first stage focuses on the word Deep, the second on learning, the third on processing and the fourth on tasks. The right side of the image shows a grid labeled 'Using [Masked Self-Attention] to Ignore Subsequent Words' with attention weights and zero weights.



It's important to note the distinct differences between the GPT model and the original Transformer model. In the original Transformer, the encoder's self-attention is not masked, allowing the model to focus on any position within the sequence. This setup is advantageous for some NLP tasks like machine translation. However, for GPT, the lack of masking can result in text that improperly relies on future information, violating the principles of natural language generation.

Additionally, in the traditional Transformer model, both the encoder and decoder are crucial, with the encoder encoding the input sequence and the decoder producing the target sequence based on the encoder's output. In contrast, GPT aims to learn a universal language representation, not to translate an input sequence to a target sequence. Thus the GPT model exclusively employs the decoder architecture, streamlining its focus on language understanding and generation.

### **3.3 The generative pretraining process**

GPT is a method that uses vast amounts of unlabeled text data to learn how to predict the structure and latent meanings within texts. Its objective is to develop a language model that can produce text sequences resembling those in the training data. Specifically, given a text sequence  $x=(x_1, x_2 \dots, x_l)$ , the model is tasked with learning the probability distribution  $P(x)$  of the entire sequence. This

distribution is understood as a series of conditional probabilities:

$$P(x) = \prod (P(x_t|C)) \quad (3.1)$$

where  $C$  denotes the context, which includes the previous  $t - 1$  words  $(x_1, x_2, \dots, x_{t-1})$ .  $P(x_t|C)$  calculates the likelihood of producing the  $t$ -th word based on this context.

### 3.3.1 The objectives of generative pretraining

In the process of GPT, the main goal of the model is to maximize how likely it is to correctly predict words from the training data, a method known as maximum likelihood estimation. Specifically, given a training sample  $x$  and the model-generated probability distribution  $P(x)$ , the objective function is defined as

$$\mathcal{L}_{LM} = \sum_{i=1}^T T \log P_{\theta}(w_i | w_{<i}) \quad (3.2)$$

where  $\mathcal{L}_{LM}$  is the loss function for the autoregressive language model,  $T$  represents the length of the training sample,  $w_{<i}$  is the sequence of words before position  $i$ , and  $w_i$  is the word at position  $i$ .  $P_{\theta}(w_i | w_{<i})$  is the conditional probability that the model, using parameters  $\theta$ , predicts the next word  $w_i$ .

Autoregressive language models are designed to predict the next word in a sequence based on the previous words. The model calculates the likelihood of a word appearing at the current position using the history of the sequence.

Therefore  $P_\theta(w_i|w_{<i})$  indicates the probability of the word  $w_{<i}$  occurring, given the prior sequence  $w_i$ . The parameter  $\theta$  is a set of model parameters that influence its ability to make these predictions.

Initially, the input text sequence  $u$  is transformed into an embedded representation, where each word is converted into a vector using a token embedding matrix. This matrix assigns each word a fixed-dimensional vector.

Next, to help the model grasp the relationships between words, position information is added to each word's embedding. This is done through a position embedding matrix  $W_p$ , which is combined with the token embedding matrix  $W_e$  to form the initial embedding representation  $h_0$ .

This representation,  $h_0$ , is then processed through several layers of a Transformer block, each consisting of self-attention and feedforward layers. The output  $h_i$  from the  $i$ -th Transformer layer is given by:

$$h_i = \text{transformer\_block}(h_{i-1}) \quad \forall i \in [1, n] \quad (3.3)$$

where  $n$  represents the total number of layers in the Transformer stack. At the end of the Transformer stack, the final output vector  $h_n$  and a matrix  $W_e$ , containing the embedding vectors for each word in the vocabulary, are used to compute the probability of the next word  $u$  as

$$P(u) = \text{softmax}(h_n W_e^T). \quad (3.4)$$

Here, the `softmax` function converts the output into a probability distribution, with  $P(u)$  indicating the likelihood of each word being the next word in the sequence. The model selects the next word by finding the highest value from the dot product between  $P(u)$  and the word embeddings.

The primary objective of training the GPT-1 model is to minimize negative log-likelihood. This approach essentially reduces the errors the model makes in predicting the next word based on the preceding context. To meet this target, the model not only optimizes the parameters of its Transformer structure but also fine-tunes the word embedding vectors. Throughout each training cycle, which includes both the forward pass and backpropagation, the model calculates and minimizes the loss function using the backpropagation algorithm to refine its parameters. Therefore during the unsupervised pretraining phase, the GPT-1 model continually adjusts both its parameters and the representations of words to improve its predictive accuracy.

### **3.3.2 The error backpropagation process in generative pretraining**

All models in the GPT series use a method known as reversible tokenization, which is based on data compression techniques (refer to Section 4.2.3), to facilitate the explanation of the error backpropagation process in GPT. In this chapter, we simplify the initial embedding of words. It is assumed that the model

generates its initial embeddings by multiplying the one-hot encoding of the input text sequence with a word embedding matrix  $W$ , represented as  $W_e$ :

$$W_e = X \times W. \quad (3.5)$$

Here,  $x$  stands for the one-hot encoded matrix of the input text sequence, while  $W$  is the word embedding matrix. The model then enhances these embeddings by applying multihead attention, residual connections, and layer normalization within each Transformer stack layer to produce contextually enriched embeddings  $h_i$ :

$$h_i = \text{transformer\_block}(h_{i-1}) \quad \forall i \in [1, n]. \quad (3.6)$$

In the final step, the output from the last Transformer stack layer is multiplied by a weighting matrix and processed through a softmax function to compute the probability distribution for the next word in the sequence.

When calculating its loss function, the model evaluates the difference between the actual next word's one-hot encoding and its predicted probability distribution to compute the loss. This loss is then sent back through the Transformer stack layers using the backpropagation algorithm, which updates the embedding representations for each word. The gradient calculation for the embedding vector  $v$  of any particular word during backpropagation is

$$\frac{\partial L}{\partial v} = \frac{\partial L}{\partial p} \times W. \quad (3.7)$$

In this formula,  $L$  is the model's loss function,  $p$  is the predicted probability distribution of the next word by the model, and  $W$  is the embedding matrix. By applying the gradient descent method, the model uses this gradient to update the value of  $v$ , thus improving the accuracy of future predictions. Once training is complete, these refined word embeddings are utilized as the initial embedding representations for subsequent tasks.

During GPT, the strategies for adjusting the learning rate and methods of regularization play a crucial role in enhancing model performance. Commonly, strategies such as a warmup phase for the learning rate followed by a gradual decrease are implemented to optimize learning rates, aiming to increase the stability of the training process. Furthermore, to avoid overfitting—a common challenge in model training—techniques like Dropout and weight decay (for instance, L2 regularization) are used to maintain the simplicity of the model structure.

Once pretraining is complete, the GPT model becomes capable of handling various NLP tasks, including text classification, sentiment analysis, and question answering. To tailor the model for these specific tasks, supervised fine-tuning is conducted. In this phase, the model is trained with labeled data pertinent to specific tasks, which allows it to acquire task-specific knowledge. This stage is considered a form of transfer learning, as the model utilizes the

extensive linguistic knowledge it gained during pretraining to improve its effectiveness on downstream tasks.

GPT is an approach where language models learn from a vast amount of unlabeled text data through unsupervised learning. The GPT model uses a Transformer architecture powered by self-attention mechanisms and employs a causality mask during self-attention calculations to ensure that the model only accesses information from previous segments. The objective is to maximize the likelihood on the training data (**Eqs. 3.2–3.3**) to master a generative probability distribution (**Eqs. 3.1–3.3**). After this phase, the model can undergo supervised fine-tuning to adapt to a variety of NLP tasks. This method of GPT has proven to significantly boost performance across numerous language processing applications, highlighting its substantial potential.

## **3.4 Supervised fine-tuning**

GPT equips the GPT model with a fundamental understanding of grammar and vocabulary. During the next phase, supervised fine-tuning, the model's parameters are further refined using annotated data from specific tasks. This enables the model to perform specialized NLP tasks like sentiment analysis, text classification, and question answering.

### **3.4.1 The principles of supervised fine-tuning**

This section introduces two new equations to describe the supervised fine-tuning process. Consider a training example  $(x, y)$  for a specific task, where  $x$  is the input sequence and  $y$  is the associated label. The aim during this phase is to minimize the loss function defined as

$$L'(\theta) = -E[\log P(y|x;\theta)]. \quad (3.8)$$

Here,  $\theta$  denotes the model parameters,  $E$  is the expected value, and  $P(y|x;\theta)$  represents the probability of generating the label  $y$ , given the input  $x$  and parameters  $\theta$ . This loss function, while similar to that described in [Section 3.3.1 \(Eqs. 3.2–3.3\)](#), specifically targets the probability of producing task-related labels, not just words. The objective for fine-tuning the model for specific tasks is set as follows:

$$\theta^* = \operatorname{argmin}_{\theta} \sum L'(\theta) + \lambda R(\theta). \quad (3.9)$$

Here,  $\theta^*$  are the adjusted parameters after fine-tuning,  $L'(\theta)$  is the loss function for task-specific data,  $\lambda$  is a regularization coefficient, and  $R(\theta)$  is the regularization term, usually the L2 norm of the parameters. The goal is to minimize the loss on task-specific data while using regularization to prevent overfitting.

### 3.4.2 Supervised fine-tuning for specific tasks

In the GPT phase, the GPT model diligently works to grasp the various aspects of natural language, enabling it to produce text similar to the input sequences it receives. At



this stage, the GPT operates as a universal language model, capturing extensive knowledge encompassing vocabulary, grammar, and semantics. During the supervised fine-tuning phase, the model is trained with annotated data, allowing it to acquire task-specific knowledge. This resembles the process of transfer learning, where the model leverages the extensive linguistic insights gained during pretraining to boost its performance on specific tasks.

The representations learned during pretraining effectively capture the fundamental structures of the input data. These are then leveraged in the fine-tuning phase to enhance the model's understanding of task-specific features. For particular NLP tasks, the input sequence must undergo specific modifications to ensure the model focuses on relevant information. These modifications are referred to as task-specific input transformations.

The main purpose of these transformations is to tailor the input sequence to fit the requirements of specific tasks. This typically involves incorporating special tokens within the sequence to emphasize task-related details. For instance, in a question-answering task, both the question and its contextual information need to be combined into a single input sequence to enable the model to generate appropriate answers. Inserting a special separator, such as "[SEP]," between the question and context helps the model to more accurately interpret the input structure and deliver more precise predictions.

**Fig. 3.4** illustrates the application of the GPT model to specific NLP tasks. This process is divided into two main stages: first, the unsupervised pretraining stage, where the model learns language models through generative training; second, the supervised fine-tuning stage, where the model is tailored to specific NLP tasks. On the left side of **Fig. 3.4**, the model learns the structure of language by maximizing the likelihood of input sequences, while on the right side, it shows how model parameters are optimized for specific tasks to improve task performance. The four NLP tasks are highlighted in **Fig. 3.4**:

1. **Text classification**: In text classification tasks, the model classifies input text into one or several predefined categories based on content. Task-specific input transformations are straightforward here, involving only a text segment. The original text is input directly into the model. Post fine-tuning, the GPT model can predict category labels based on the input content. A fully connected layer is typically added to the output layer for task-specific predictions. To help the model distinguish different tasks, a special classification token (e.g., “[CLS]”) might be placed at the start of the sequence.
2. **Natural language inference (NLI)**: This task involves determining if a hypothesis is derivable from a given premise, with potential outputs being entailment, contradiction, or neutrality. The input

sequence includes both the premise and the hypothesis, separated by a special symbol (e.g., “[Delim]”). A special inference token (e.g., “[NLI]”) at the sequence’s start signals the inference task to the model.

3. **Similarity:** Here, the model evaluates the similarity between two text sequences. A special delimiter (e.g., “[Delim]”) is typically placed between the texts to be compared, and a similarity marker (e.g., “[SIM]”) might be added at the start to signify a similarity evaluation task.
4. **Multiple choice:** This task requires selecting the most appropriate answer from multiple options based on a question. A special delimiter (e.g., “[Delim]”) is used between the question and each option to differentiate them. A multiple-choice token (e.g., “[MCH]”) at the start indicates the task type.

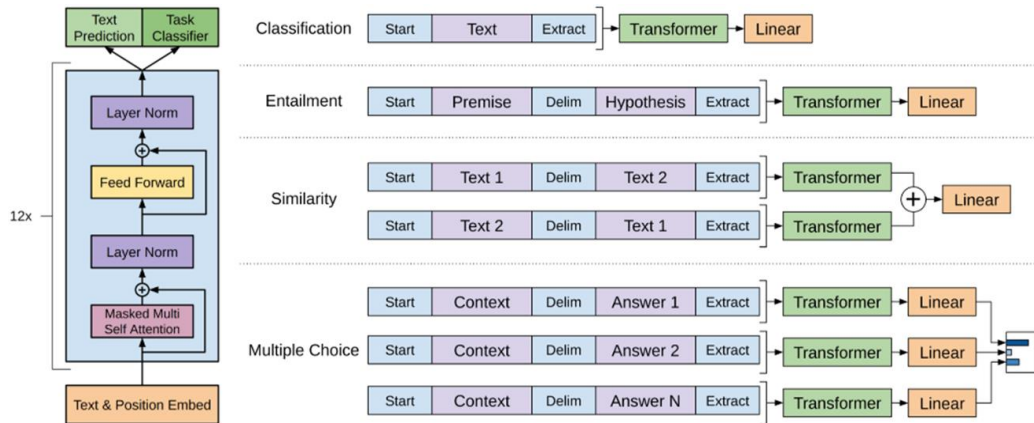


**FIGURE 3.4** Application process of generative pretraining models. Image source: Paper “Improving language understanding by generative pretraining”.

At the top, there are two boxes labeled Text Prediction and Task Classifier. Below these, there is a block labeled Text and Position Embed, which connects to a larger block containing Layer Norm, Feed Forward and Masked Multi Self Attention, repeated 12 times. The tasks shown are Classification, Entailment, Similarity and Multiple Choice. Each task has a sequence starting with Start, followed by various elements like Text, Premise, Hypothesis, Text 1, Text 2, Context, Answer 1, Answer 2 and Answer N, with delimiters and Extract. These sequences are processed by a Transformer and then a Linear layer. For Similarity, there is an additional step where outputs from two Transformers are combined before the Linear layer.

Task-specific input transformations significantly enhance the GPT model's ability to conduct effective transfer learning across a range of NLP tasks. By incorporating special symbols within the input sequences, the model gains a better grasp of information pertinent to specific tasks, which boosts its performance in downstream applications. It is noteworthy that the precise form of these special markers can vary depending on the design of the model and the requirements of the specific tasks. For instance, the original GPT paper indicates that every task-specific input transformation includes the addition of randomly initialized start and end tokens.

For these transformations to be effective, it is vital to maintain consistency in the vocabulary and encoding schemes used during both the pretraining and fine-tuning phases. This involves setting aside some vocabulary space for special symbols during pretraining and employing the same vocabulary and encoding scheme for encoding input sequences during fine-tuning. Such consistency ensures that the model continues to exhibit a uniform understanding and representation of language throughout the transfer learning process.

In conclusion, tailoring input sequences to different NLP tasks not only helps the model comprehend task-specific information more effectively but also enhances its performance in subsequent tasks. Through these input transformation techniques, GPT models can effectively

engage in transfer learning across various NLP tasks, achieving superior NLP performance.

### 3.4.3 Fine-tuning steps

The supervised fine-tuning process involves several crucial steps:

1. **Target task selection:** Determine a specific NLP task based on requirements, such as sentiment analysis, text classification, or named entity recognition.
2. **Training data preparation:** Collect annotated data pertinent to the chosen task, typically comprising input sequences (like text snippets) and their corresponding labels (e.g., sentiment or category labels).
3. **Model initialization:** Start with the model parameters acquired during GPT to ensure the model has robust language representation capabilities from the beginning of the fine-tuning phase.
4. **Model fine-tuning:** Utilize task-related data and the loss function  $L'(\theta)$  to fine-tune the model. Use optimization algorithms like SGD or others to update the parameters, incorporating regularization techniques such as L2 regularization to adjust the regularization coefficient  $\lambda$  and prevent overfitting.

5. **Model performance evaluation:** Measure the fine-tuned model's performance on a task-specific validation set using metrics such as accuracy or F1 score.
6. **Hyperparameter adjustment:** Fine-tune the model's hyperparameters, like the learning rate or regularization coefficient, based on performance metrics from the validation set to enhance the model's efficacy.
7. **Model application:** Deploy the fine-tuned model to tackle real-world problems, process new inputs, and produce predictions for specific tasks.

The model can be tailored according to the task and data characteristics. For instance, in text classification tasks, a fully connected layer can be added to the final layer to generate task-specific predictions. In named entity recognition, a conditional random field layer might be integrated into the output layer to better capture label sequence dependencies. This adaptability makes the GPT model versatile across various NLP tasks.

GPT's fine-tuning typically delivers robust performance with relatively few annotated data, benefiting significantly from the rich linguistic insights gained during pretraining. Thus supervised fine-tuning is effectively refining the pretrained model for specific tasks using limited annotated data.

In essence, supervised fine-tuning is pivotal for leveraging GPT models in specific NLP tasks. It involves

optimizing the model on task-relevant data using learned language representations to achieve optimal outcomes. This process includes fine-tuning the loss function  $L'(\theta)$  (Eqs. 3.3–3.7) and employing regularization to prevent overfitting. Furthermore, the model's architecture can be modified to suit different NLP tasks, enhancing its adaptability.

## 3.5 Summary

The GPT models that form the basis of the GPT series, discussed in this chapter, have been instrumental in shaping the technological framework of ChatGPT. GPT's training regimen is structured into two principal phases: unsupervised pretraining and supervised fine-tuning. This dual-phase approach offers an effective training paradigm for ChatGPT. During unsupervised pretraining on vast quantities of unlabeled text, ChatGPT develops a nuanced understanding of language. In the subsequent supervised fine-tuning phase, it is tailored to excel at specific tasks, significantly boosting its efficacy in targeted areas.

The architecture used by GPT, which consists solely of a Transformer decoder and masked self-attention mechanisms, serves as a critical blueprint for ChatGPT's design. These elements are not only capable of advanced representation learning and efficient parallel processing, but they also ensure that the generation of text is influenced only by the current position and preceding



context. This focus helps ChatGPT generate responses that are both coherent and meaningful.

---

## CHAPTER 4

# Unsupervised multitask and zero-shot learning

---

### Abstract

This chapter delves into unsupervised multitask learning (MTL) and zero-shot learning, focusing on their implementation in OpenAI's GPT-2 model. MTL enhances model generalization by training on multiple related tasks, while zero-shot learning enables task performance without specific training data. The chapter explores the Transformer architecture, contrasting BERT's bidirectional encoder with GPT-2 unidirectional decoder. Key advancements in GPT-2 include layer normalization for stability, orthogonal initialization for gradient preservation, byte-pair encoding for efficient tokenization, and learnable relative positional encodings for better context understanding. The unsupervised pretraining of GPT-2 allows it to perform various natural language processing tasks without fine-tuning, demonstrating the synergy between MTL and zero-shot learning. The chapter concludes by highlighting GPT-2's

autoregressive generation process and its impact on subsequent models like GPT-3.

## **Keywords**

Unsupervised learning; multitask learning; zero-shot learning; transformer architecture; GPT-2; bidirectional encoder representations from transformers (BERT); layer normalization; orthogonal initialization; byte-pair encoding (BPE); autoregressive generation

Multitask learning (MTL) is a strategic approach in machine learning that focuses on training models on several related tasks at once. By sharing insights across these tasks, the model's ability to generalize and adapt to new scenarios is significantly enhanced. OpenAI highlighted this technique, alongside zero-shot learning, in their GPT-2 paper.

Zero-shot learning, a subset of unsupervised MTL, allows models to tackle tasks without specific training data. Instead, these models utilize a broad language model learned beforehand. The beauty of zero-shot learning lies in its cost-effectiveness, as it skips the need for task-specific tuning, making the model versatile across unfamiliar tasks. This chapter dives deep into the modifications of the Transformer architecture in GPT-2, explores the principles behind unsupervised MTL and zero-shot learning, and unpacks these concepts in depth.

## **4.1 Encoder and decoder**

In October 2018, OpenAI unveiled GPT, followed 6 months later by Google with BERT. These models are based on the Transformer architecture, which consists of encoders and decoders stacked in multiple layers. **Fig. 4.1** illustrates BERT as a pretrained language model built on Transformer's encoder architecture, while GPT leverages the decoder side.

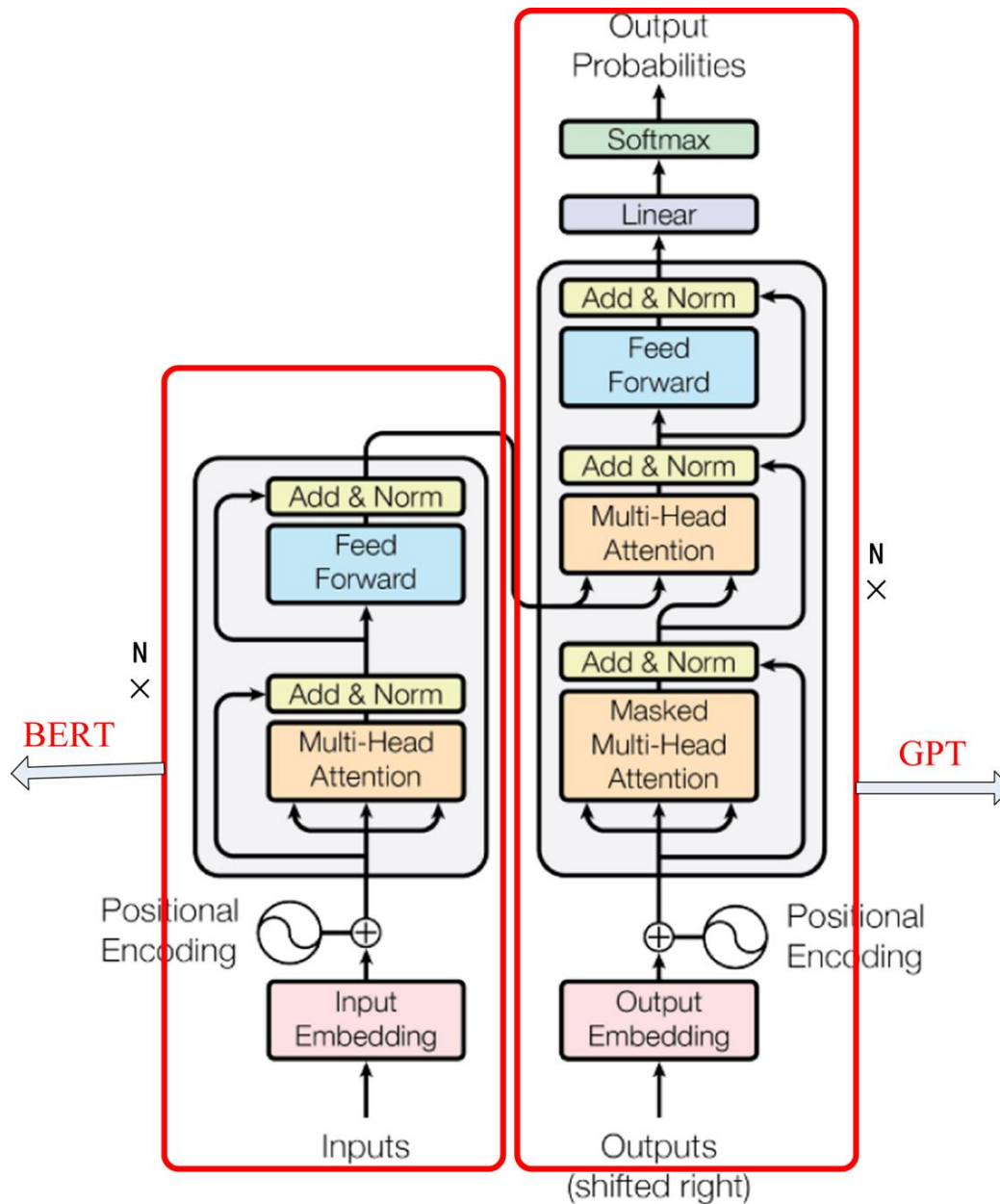


**FIGURE 4.1** Transformer encoder (left) and decoder architecture (right).

On the left, the BERT model is shown with input embeddings followed by positional encoding. The architecture includes layers of multi-head attention, add and norm and feed-forward components. On the right, the

GPT model is depicted with output embeddings followed by positional encoding. The architecture includes masked multi-head attention, add and norm, multi-head attention, feed-forward layers and output probabilities. Both models have N layers, indicated by N times. The BERT model processes inputs, while the GPT model processes outputs shifted right.

BERT features a bidirectional approach, analyzing the words before and after a target word to predict its context. It employs a strategy known as masked language model, where random words in a sentence are hidden to make the model infer them based on the whole sentence's context. This technique enhances BERT's ability to grasp context from both directions, leading to notable improvements across various natural language processing (NLP) tasks.

After BERT's success, OpenAI introduced GPT-2, an advanced version of GPT. GPT-2 maintains a unidirectional flow from left to right during its unsupervised pretraining. The model size of GPT-2 medium version matches that of BERT large, yet their architectures and training techniques greatly differ, affecting their performance in specific tasks. As depicted in **Fig. 4.2**, BERT utilizes a bidirectional encoder using self-attention to process text context, excelling in tasks that demand an understanding of sentence structure and contextual relationships, like question answering, entity recognition, and relation extraction.

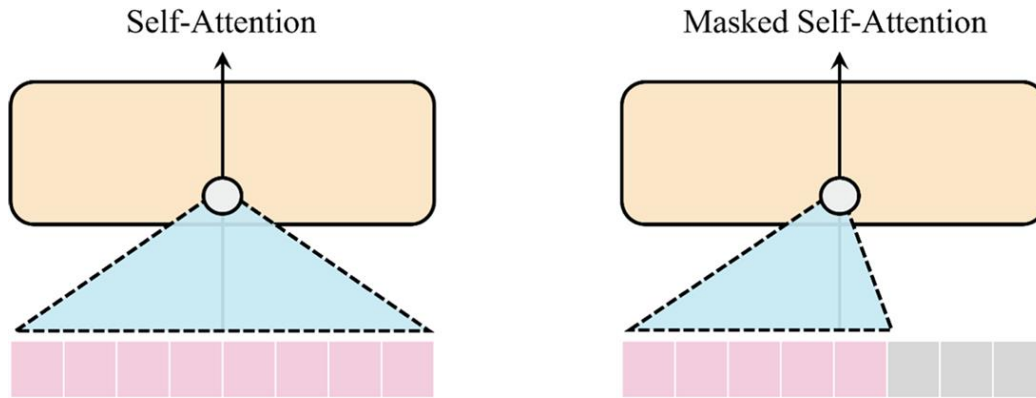


**FIGURE 4.2** Differences between self-attention and masked self-attention. Image source: <http://jalammar.github.io/illustrated-gpt2/>.

The left diagram is labeled Self-Attention and shows a rectangular block with an arrow pointing upwards from a central circle. Below the circle, a triangular area is connected to a series of smaller rectangles. The right diagram is labeled Masked Self-Attention and is similar to the left diagram but with some of the smaller rectangles at the bottom right being shaded differently, indicating masking.

Conversely, GPT-2 features a unidirectional decoder that uses a masked self-attention mechanism to focus primarily on learning through language modeling, which might limit its bidirectional context comprehension compared to BERT. However, GPT-2 performs exceptionally well in various language generation tasks, including text summarization, translation, and dialog creation. Despite their differences,

both models are recognized as robust tools in NLP, each excelling in specific tasks where they are most effective.

## 4.2 GPT-2

GPT-2, the follow-up to the original GPT, retains the decoder architecture from the Transformer model but expands massively on the training data used. It was trained on a hefty 40 GB dataset named WebText, gathered by the OpenAI team. **Fig. 4.3** highlights the varied sizes of GPT-2 models used during training, ranging from 117 MB (small) to 1542 MB (extra-large) in parameter size. The smallest version comprises 12 decoder layers, needing 500 MB of storage for its parameters. On the other hand, the largest version features 48 decoder layers, with its parameter volume 13 times greater than the smallest, necessitating 6.5 GB of storage. **Fig. 4.4** illustrates that GPT-2 architecture has significantly evolved from that of GPT-1, reflecting major advancements in its design and capabilities.



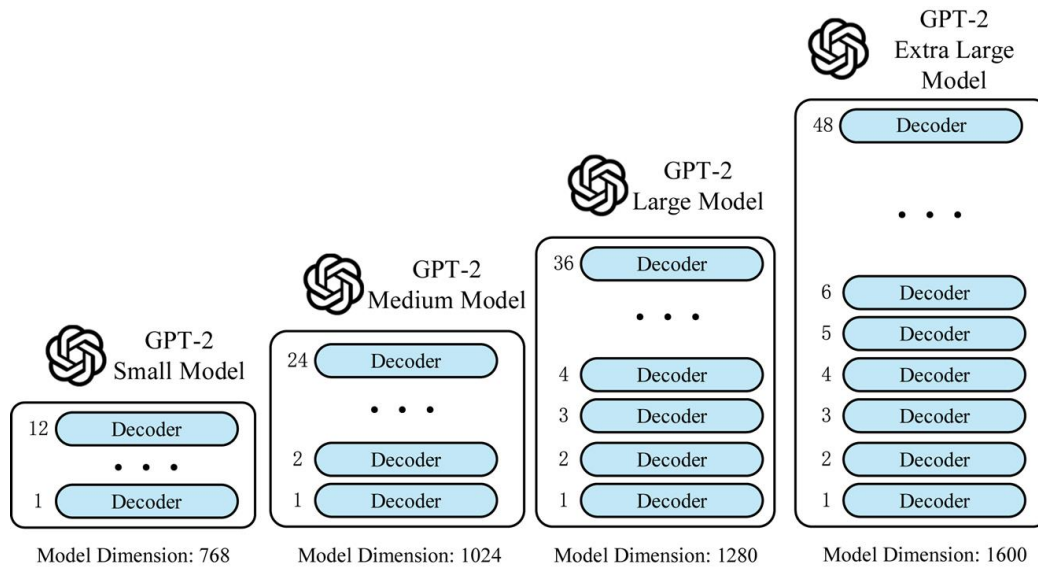


**FIGURE 4.3** Four different sizes of GPT-2 models.

The image compares four GPT-2 models: Small Model, Medium Model, Large Model and Extra Large Model. The GPT-2 Small Model has a model dimension of 768 and includes 12 decoders. The GPT-2 Medium Model has a model dimension of 1024 and includes 24 decoders. The GPT-2 Large Model has a model dimension of 1280 and includes 36 decoders. The GPT-2 Extra Large Model has a model dimension of 1600 and includes 48 decoders.

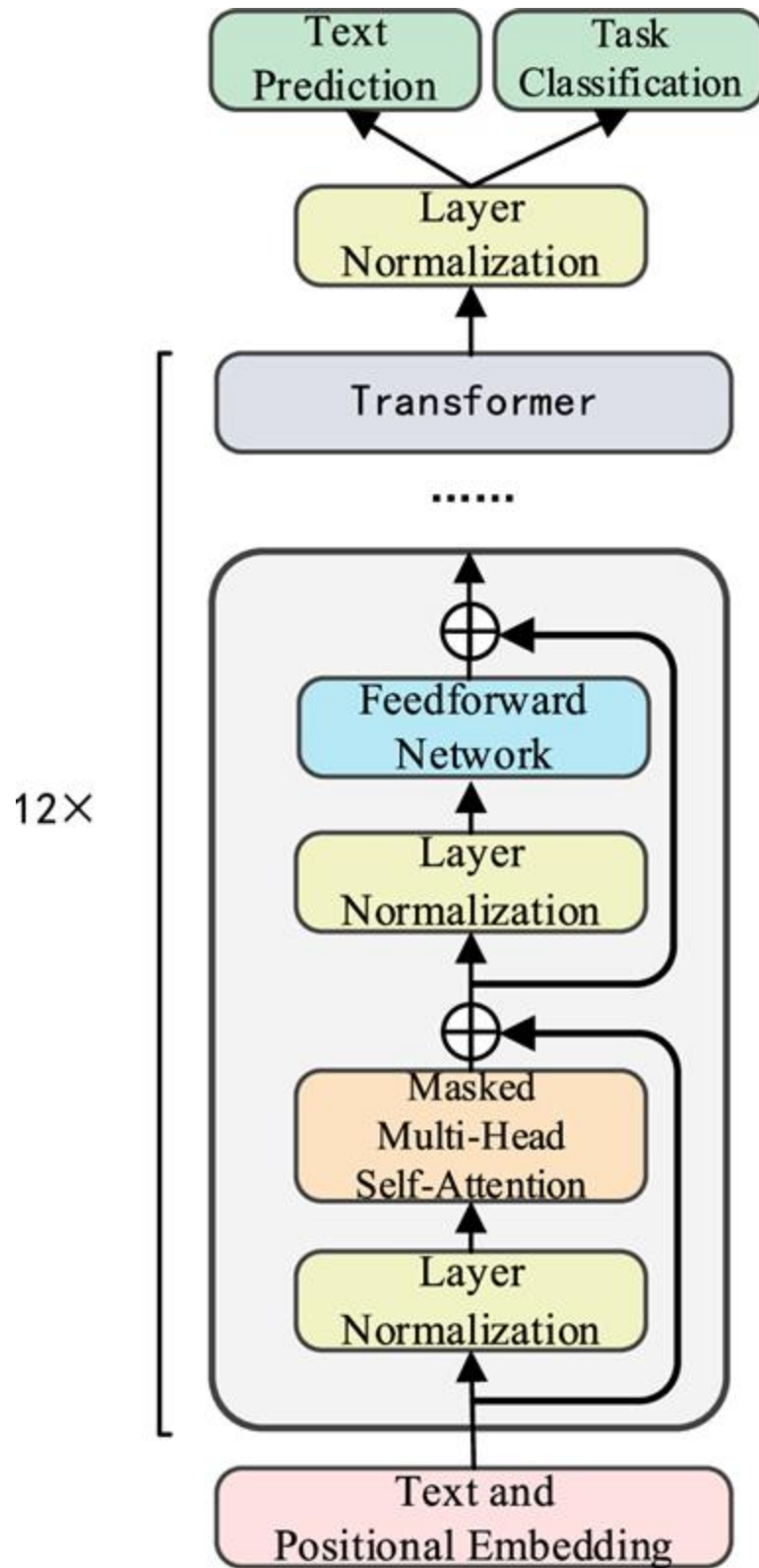


**FIGURE 4.4** Model architecture of GPT-2.

At the bottom, there is Text and Positional Embedding. Above it, there is a block labeled Layer Normalization followed by Masked Multi-Head Self-Attention. Another Layer Normalization block follows, leading to a Feedforward Network. This sequence is repeated 12 times. Above these repeated blocks, there is a Transformer block. At the top, there is another Layer Normalization block, which branches into Text Prediction and Task Classification.

### **4.2.1 Layer normalization**

Layer normalization (LayerNorm) is a technique used to introduce normalization operations within the layers of neural networks, particularly evident in GPT-2's Transformer layers. Unlike its predecessor, GPT-1, GPT-2 incorporates a LayerNorm layer at both the input and output of each Transformer layer, ensuring normalization of these points. This method effectively combats common neural network training challenges such as gradient vanishing and explosion, thereby bolstering training stability and speeding up convergence.

Gradient issues, particularly vanishing and exploding gradients, are prevalent in deep neural networks due to the multilayer gradient propagation. Layer normalization counters this by standardizing the input data's mean and

variance to 0 and 1, respectively, ensuring uniformity across scales.

As illustrated in **Fig. 4.5**, normalization adjusts and balances the input data, setting its mean to 0 and variance to 1, which minimizes scale disparities. This adjustment not only facilitates smoother gradient flow during backpropagation but also stabilizes gradient variations throughout the training phase. Moreover, layer normalization tackles covariate shift—a scenario where training and testing data distributions do not align—by equating the scale of both datasets, thus boosting the model’s generalization capabilities. Ultimately, normalization ensures uniform gradient scaling across all dimensions, aiding optimizers in navigating the optimal direction for weight updates more effectively and accelerating the model’s convergence and overall training performance.

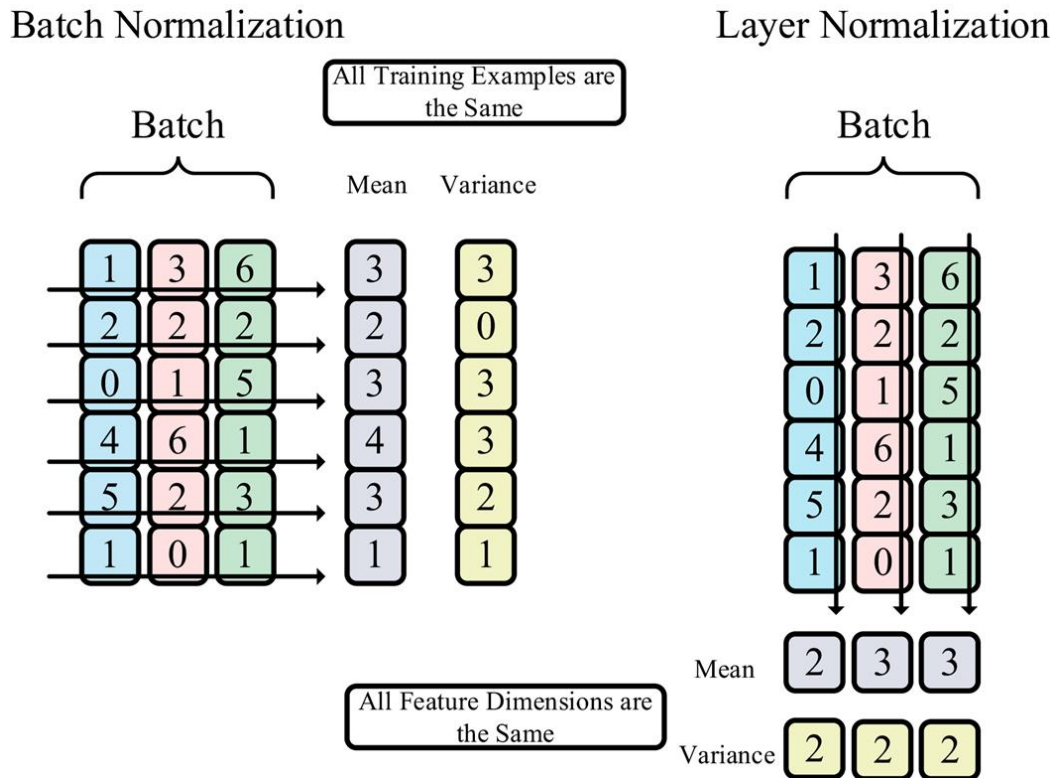


**FIGURE 4.5** Illustration of two types of normalization. Image source: <https://blog.csdn.net/wwangfabei1989/article/details/90547149>.

On the left, Batch Normalization is shown with a batch of training examples. Each example has four features and the mean and variance are calculated for each feature across the batch. The mean values are 3, 2, 4 and 3 and the variance values are 3, 2, 3 and 1. A note indicates All Training Examples are the Same. On the right, Layer Normalization is shown with the same batch of training examples. The mean and variance are calculated for each example across all features. The mean values are 2, 3 and 3

and the variance values are 2, 2 and 2. A note indicates All Feature Dimensions are the Same.

### **4.2.2 Orthogonal initialization**

Orthogonal Initialization is a method employed to set up neural network parameters, mainly focusing on the weight matrices. It initializes these matrices as orthogonal matrices to combat issues like gradient vanishing or explosion during the training phase.

GPT-2 follows a similar weight initialization strategy as GPT-1, specifically utilizing orthogonal initialization to ensure gradient stability throughout training. Orthogonal matrices, by design, consist of row and column vectors that are unit vectors and orthogonal to each other, with the property that the inverse of such a matrix is its transpose ( $A^{-1} = A^T$ ). These matrices are known for their robust numerical characteristics, such as having a condition number of 1, ensuring that they do not exacerbate errors in matrix calculations.

The key benefit of orthogonal initialization lies in its ability to stabilize gradients. During neural network operations, gradients need to traverse through various weight matrices. Improper initialization could result in gradients that either vanish (become too small) or explode (become excessively large), adversely impacting model training and convergence. Orthogonal matrices help mitigate this risk because their products remain

orthogonal, maintaining consistent gradient norms across different layers.

**Fig. 4.6** illustrates an example of orthogonal initialization. The process begins by creating a random matrix with dimensions  $(n, n)$ , each element drawn from a standard normal distribution. This matrix is then subjected to  $QR$  decomposition ( $A = QR$ ), where  $Q$  represents an orthogonal matrix and  $R$  an upper triangular matrix. The  $Q$  matrix from this decomposition serves as the initialized weight matrix, possessing favorable numerical qualities that contribute to gradient stability.

$$\begin{bmatrix} \blacksquare & \blacksquare & \dots & \blacksquare \\ \blacksquare & \blacksquare & \dots & \blacksquare \\ \blacksquare & \blacksquare & \dots & \blacksquare \end{bmatrix} = \begin{bmatrix} | & | & \dots & | \\ \mathbf{q}_1 & \mathbf{q}_2 & \dots & \mathbf{q}_n \\ | & | & \dots & | \end{bmatrix} \begin{bmatrix} \text{shaded triangle} \end{bmatrix}$$

**A**
**Q**
**R**

**FIGURE 4.6** Illustration of  $QR$  decomposition. Note that orthogonal initialization is particularly effective for square weight matrices  $(n, n)$ . For nonsquare matrices  $(n, m)$ , a similar approach involves generating a random matrix and then performing a compact  $QR$  decomposition to extract the first  $m$  columns, resulting in an orthogonal weight matrix suited for initialization.

Matrix  $A$  is represented by a square matrix with dots indicating the continuation of elements. Matrix  $Q$  is represented by a rectangular matrix with columns  $q_1, q_2, \dots, q_n$ , where each column is separated by vertical lines. Matrix  $R$  is represented by an upper triangular matrix shaded in gray.

### 4.2.3 Reversible tokenization

All models in the GPT series use a reversible tokenization method based on data compression known as byte-pair encoding (BPE). This algorithm reduces the vocabulary size



by combining frequently appearing word pairs into new units. Since the tokenized text can be entirely reconstructed during decoding, this method is described as reversible.

The fundamental concept behind the BPE algorithm is to consolidate commonly occurring character combinations, like words or phrases, into new symbols. This process not only shrinks the vocabulary but also enhances the model's ability to generalize, efficiently manages rare words and morphological changes, thereby boosting overall performance. The detailed steps involved in the BPE algorithm are as follows:

1. Vocabulary initialization: Initially, incorporate every word (or character) from the text into the vocabulary, with each entry representing a single character.
2. Frequency counting of character pairs: Assess the frequency of each adjacent character pair within the text. These frequency counts help decide which pairs to merge into new symbols.
3. Merge high-frequency character pairs: Find and merge the most frequently occurring character pair into a new symbol. This new symbol is then added to the vocabulary, and the text is updated to substitute the original character pairs with this new symbol.
4. Iterative merging: Continue the process of counting and merging as needed, until reaching the

predefined number of merges or the desired vocabulary size. Each cycle focuses on merging a new high-frequency pair, progressively enlarging the vocabulary to encompass longer subwords or phrases.

Here's an example to illustrate BPE:

1. Start with a corpus and count the frequency of each word. The frequencies are noted in the format "[frequency]word\_", where "\_" signifies the end of a word. For instance, the corpus might have: Corpus: [7]old\_, [3]older\_, [9]finest\_, [4]lowest\_.
2. Determine the token vocabulary size or the number of iterations to stop the algorithm.
3. Count the frequency of each character, including the end symbol "\_". Refer to **Table 4.1**.
4. Merge the two consecutive characters (or sequences) with the highest frequency. For example, in the first iteration, "e" and "s" might be merged into "es" since they appear together 13 times ( $9 + 4$ ). Add "es" and its frequency to the vocabulary, and decrease the counts for "e" and "s."
5. In the next iterations, "es" and "t" could be merged because "est" appears 13 times, which is the highest frequency. Update the vocabulary to include "est" and reduce "t," keeping the vocabulary length the same.

6. Merge “est\_,” also appearing 13 times. Merging with the termination marker is crucial. It helps distinguish words like “estimate” and “highest.” Both contain “est,” but one ends with “est\_,” and the other starts with it. The algorithm treats “est” and “est\_” differently, recognizing “est\_” as part of “highest” and not “estimate,” as shown in **Table 4.2**.
7. Merge “ol,” which appears 10 times.
8. If the iteration count is set to four, the final vocabulary would be (*\_ o, l, d, r, f, i, n, w, est, ol, est\_*).

**Table 4.1**

**Frequency of character occurrences.**

| Character | Frequency |
|-----------|-----------|
| –         | 23        |
| O         | 14        |
| L         | 14        |
| D         | 10        |
| E         | 16        |
| R         | 3         |
| F         | 9         |
| I         | 9         |
| N         | 9         |
| S         | 13        |
| t         | 13        |
| w         | 4         |

---

**Table 4.2****Example of character merging.**

---

| Character | Frequency  |
|-----------|------------|
| –         | $23-13=10$ |
| O         | 14         |
| L         | 14         |
| D         | 10         |
| E         | 3          |
| R         | 3          |
| F         | 9          |
| I         | 9          |
| N         | 9          |
| W         | 4          |
| est       | $13-13=0$  |
| est_      | 13         |

The description illustrates that the vocabulary length under BPE can increase, decrease, or remain stable. BPE efficiently compresses the original text into a shorter format while preserving patterns that frequently occur between characters. This method shines in NLP tasks where it excels at managing rare words and morphological

variations by decomposing them into smaller subword units, improving model performance. In large-scale models like GPT-2, BPE facilitates better handling of diverse NLP tasks, enhancing both the model's adaptability and effectiveness.

In terms of handling rare words, BPE excels by splitting infrequent words in training data into common subword units, enhancing the model's ability to grasp their meanings. For instance, a seldom-seen word like "neuroscientific" could be segmented into "neuro," "scien," and "tific." These subwords, appearing more frequently in training data, help the model easily acquire their semantic meanings.

BPE also effectively addresses morphological variations, which can inflate vocabulary sizes and complicate computations in many languages. By breaking down words like the English verbs "running" and "ran" into components such as "run," "ning," and "ran," BPE enables the model to more adeptly learn about word roots and their morphological dynamics, simplifying the learning process and reducing computational demands.

The use of the BPE algorithm with Chinese text is notably more intricate than with English due to the fundamental differences in their vocabularies. In Chinese, each character typically has a distinct meaning, and words are often formed by combining these characters, each adding layers of meaning.

However, BPE faces some challenges with Chinese that it doesn't with English. Unlike English, where morphological changes often involve clear prefixes and suffixes, Chinese variations are less straightforward, limiting the effectiveness of BPE in some scenarios. Addressing these challenges requires developing models and algorithms that are specifically geared towards the unique properties of the Chinese language.

In essence, the adoption of BPE in GPT series models significantly shrinks the vocabulary size, which in turn reduces the number of parameters the models need to learn. This reduction eases computational demands and accelerates both training and inference processes. Additionally, by leveraging the shared characteristics among subword units, these models gain enhanced capabilities to understand and represent the semantic details of words, boosting their effectiveness across a variety of NLP applications.

Moreover, BPE improves the GPT models' handling of unfamiliar words. When encountering such words, the models can break them down into recognizable subword components to deduce their meanings, enhancing the models' ability to generalize and manage new or uncommon words effectively.

In the context of transfer learning, BPE's integration makes the GPT series especially potent. The models' ability to learn and utilize commonalities among subword units across different tasks translates into superior performance

in transfer learning scenarios, thus bolstering their flexibility across diverse NLP challenges.

For multilingual tasks, BPE's role is crucial in enabling the GPT models to adeptly manage linguistic diversity. By identifying and utilizing common roots and morphological variations across languages, BPE equips GPT-2 with the ability to grasp and process multiple languages, leading to improved outcomes in multilingual contexts. This capability is particularly beneficial in today's globally interconnected digital landscape.

BPE significantly boosts model efficacy in NLP tasks by reducing vocabulary size and, consequently, the number of parameters a model needs to master. This reduction not only mitigates the risk of overfitting but also allows models to more effectively grasp the semantic nuances of words. This enhanced understanding proves beneficial across a range of applications, from machine translation to semantic analysis and text generation.

Notably, starting with GPT-2, a variation of BPE known as byte-level BPE (BBPE) is employed. This adaptation extends the traditional BPE approach from character level to byte level. For instance, while traditional BPE struggles with the expansive Unicode character set, BBPE simplifies this by treating each byte as a fundamental processing unit, effectively limiting the basic character set to 256 ( $2^8$ ). This approach facilitates vocabulary sharing across languages and reduces vocabulary size. However, a drawback emerges with languages like Chinese, where BBPE can



considerably lengthen text sequences. As a result, while BBPE might outperform BPE on the same dataset, it can also lead to longer training and inference durations due to the increased sequence length.

#### **4.2.4 Learnable relative positional encoding**

GPT-2, built on the Transformer architecture, incorporates a learnable relative positional encoding strategy to precisely capture positional information within input sequences. This approach uniquely considers both the absolute distances and the relative positions between tokens, greatly improving the model's ability to understand long-range dependencies.

In GPT-2, relative positional encodings are seamlessly integrated into the self-attention mechanism and used alongside the  $Q$ ,  $K$ , and  $V$  matrices to capture token relationships accurately. When calculating self-attention scores, the model evaluates not only the content relevance between tokens (through the dot product of  $Q$  and  $K$ ) but also their relative positions (via relative positional encodings). These encodings are expressed through parameterized functions, such as sine and cosine functions, whose parameters are learned and optimized during the model training process.

To highlight the differences between the relative positional encodings in GPT-2 and the traditional absolute positional encodings in the Transformer, consider an example with the sentence "The cat is cute." Initially, this

sentence is tokenized into [The, cat, is, cute,.]. In the traditional Transformer, this process involves adding a precomputed matrix based on sine and cosine functions to the embedding vectors, which imparts absolute positional information to the model.

GPT-2, however, approaches this differently. It doesn't simply add positional encodings to the embeddings but integrates relative positional encodings during the self-attention calculations. These encodings are adaptable and based on the relative distances between tokens—for instance, the relative distance between “cat” and “cute” is 2. In the computation of attention weights, these relative distances are combined with the content relevance between tokens, empowering the model to leverage positional information to refine the effectiveness of its self-attention mechanism.

While the traditional Transformer model utilizes a fixed matrix of absolute positional encodings added directly to embeddings, GPT-2 opts for a dynamic, more sophisticated strategy that intertwines learnable relative positional encodings with the self-attention process. This distinction marks a significant evolution in how positional information is utilized to enhance text processing capabilities in models.

## **4.3 Unsupervised multitask learning**

GPT-2's unsupervised pretraining leverages the extensive WebText dataset, comprising about 450,000 web articles

collected via OpenAI's web crawler. After rigorous cleaning and filtering, this dataset encompasses around 40 GB, equivalent to nearly 8 billion words. This considerable increase in dataset size over GPT-1's dataset facilitates the development of more robust models.

Illustrated in **Fig. 4.7**, while multitask supervised learning focuses on predicting labels for unseen data based on labeled examples, GPT-2's approach involves generating text that mimics the training data, all through unsupervised learning. Although not explicitly detailed in the literature, the core methodologies of GPT-1 and GPT-2 in pretraining are quite similar, both using extensive unlabeled text data within a Transformer framework.

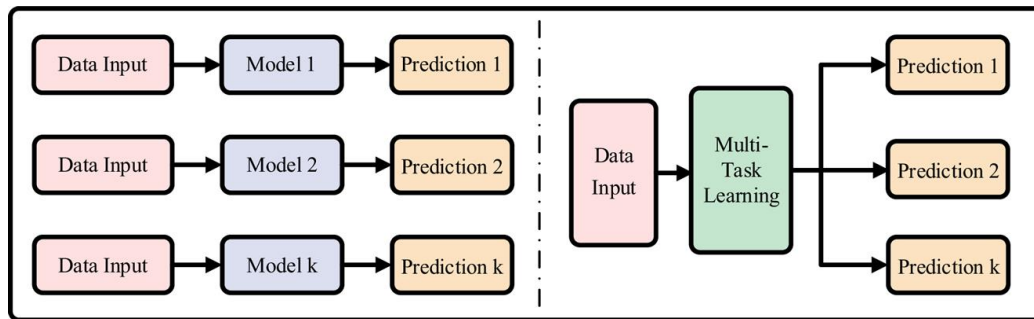


**FIGURE 4.7** Concepts of multitask supervised learning and multitask unsupervised learning. Image source: <https://zhuanlan.zhihu.com/p/348873723>.

On the left side, there are three rows, each with a Data Input leading to Model 1, Model 2, and Model k, respectively, which then lead to Prediction 1, Prediction 2, and Prediction k. On the right side, a single Data Input leads to Multi-Task Learning, which then branches out to Prediction 1, Prediction 2, and Prediction k.

During its pretraining, GPT-2 employs a maximum likelihood estimation technique, guiding the model to predict subsequent words based on the context provided. This phase is crucial as it allows the model to assimilate vast linguistic knowledge and semantic nuances from the text it processes.

GPT-2's strategy of unsupervised MTL is designed to enhance the model's versatility and generalization by extracting and applying shared insights across various tasks from extensive text data. Though inherently

unsupervised, GPT-2 often undergoes supervised fine-tuning for specific applications, enhancing its adaptability to particular tasks. This fine-tuning, while beneficial for targeted applications, typically restricts the model's broader applicability to tasks similar to those it was specifically tuned for.

## **4.4 The relationship between multitask and zero-shot learning**

As depicted in **Fig. 4.8**, zero-shot learning in sample learning allows a model to tackle tasks using prior knowledge without any task-specific supervised training. GPT-2's zero-shot learning ability stems from unsupervised MTL during its pretraining phase. In NLP, zero-shot learning can be viewed as unsupervised transfer across different subdomains. This implies that the knowledge the model gains during pretraining can be applied to various NLP tasks without needing specific supervised fine-tuning. However, zero-shot learning generally underperforms compared with supervised fine-tuning because it doesn't use labeled data for the specific tasks.

---

## Zero-shot

The model predicts answers based solely on the natural language description of the task without performing gradient updates.

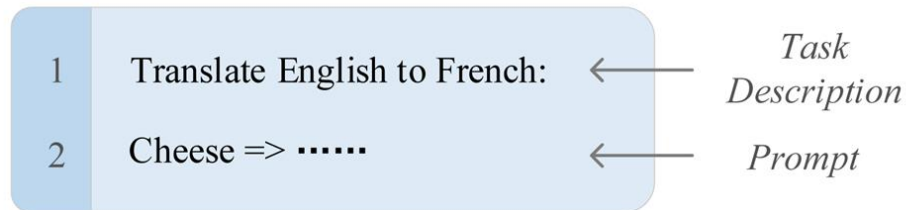


**FIGURE 4.8** Example of zero-shot learning. Image source: Paper “Language models are few-shot learners”.

In GPT-2, MTL involves absorbing a large amount of natural language text during the unsupervised pretraining phase, capturing rich linguistic structures and semantic information. Zero-shot learning enables GPT-2 to handle various NLP tasks without specific task labels. MTL and zero-shot learning in GPT-2 complement each other. Zero-shot learning leverages the knowledge acquired during the unsupervised pretraining phase, allowing it to manage multiple tasks without additional supervised training. This approach offers GPT-2 broad application prospects. The following sections will explore the applications of zero-shot learning in GPT-2 through three specific tasks: reading comprehension, text summarization, and sentiment analysis.

In a reading comprehension task, the model must generate answers to questions based on provided text. With zero-shot learning, GPT-2 does not receive task-specific

fine-tuning. Instead, the question and text are input together, and the model generates the answer. The key lies in organizing the question and text into a sequence effectively. A common method is to concatenate the question with the given text segment, inserting a specific separator between them. For instance:

Question: How many planets are there in the solar system? Text: The solar system includes one star (the Sun) and eight planets. This can be organized as: "How many planets are there in the solar system? <SEP> The solar system includes one star (the Sun) and eight planets." When this organized sequence is provided to the GPT-2 model, it may generate the answer "eight."

For a text summarization task, the model needs to extract key information from a longer text to create a concise summary. In zero-shot learning, this can be achieved by clearly specifying the task in the input sequence. For example, given the following text: Text: Computer science is the study of computers and their systems. Researchers in this field study not only the design and development of computer hardware but also programming languages, software design, algorithms, and other theories and practices related to computing.

You would concatenate the task requirement with the text and input it into the model: "Generate a summary of the following text: Computer science is the study of computers and their systems. Researchers in this field study not only the design and development of computer

hardware but also programming languages, software design, algorithms, and other theories and practices related to computing.”

At this point, the GPT-2 model might produce a summary like: “Computer science studies computer systems, hardware design, software design, and algorithms.” Note that since the model hasn’t undergone supervised training specifically for text summarization, the quality of the generated summary might not be as high as that of a model trained with supervision. Nonetheless, this still showcases the model’s robust capability to perform multiple tasks without specific task training.

In a sentiment analysis task, the model must recognize and understand the sentiment expressed in text, such as whether it is positive, negative, or neutral. In zero-shot learning, GPT-2 can generate sentences that capture the sentiment of the input text. For instance, given a product review: “The battery life of this phone is very long, and the camera performance is excellent,” you can prompt GPT-2 with: “The battery life of this phone is very long, and the camera performance is excellent. What is the sentiment of this review?” GPT-2 might respond with: “The sentiment of this review is positive.” This shows that GPT-2 can effectively perform sentiment analysis without specific task training.

GPT-2 has shown outstanding performance across various zero-shot learning tasks. For example, in reading comprehension, GPT-2 achieved top results in the



LAMBADA language modeling task without any supervised fine-tuning. In semantic similarity tasks, the STS-B (Semantic Textual Similarity Benchmark) task highlighted GPT-2's strong zero-shot capabilities. In translation tasks, GPT-2 demonstrated a notable degree of translation ability in English-French and English-German translations, despite not being fine-tuned for these tasks. The paper also discusses GPT-2's zero-shot learning performance in other NLP tasks like question answering and summarization. Although specific experimental setups for these tasks are not detailed in the paper, they were used to validate GPT-2's zero-shot learning abilities.

## **4.5 The autoregressive generation process of GPT-2**

During its unsupervised pretraining phase, GPT-2 absorbed a vast amount of knowledge from extensive natural language texts. This process optimized the model's weight matrix, enabling it to excel in language modeling tasks. In zero-shot learning applications, this pretrained model is used as is, without any task-specific fine-tuning. This means that when tackling NLP tasks, the weight matrix remains unchanged and is not further modified.

Furthermore, a vocabulary was employed during pretraining. Initially, each word's embedding vector was randomly generated, but these vectors were refined during training to maximize performance in language modeling. After pretraining, the updated embeddings for each word

reflect the language patterns the model learned from the large corpus.

For specific NLP tasks, the pretrained GPT-2 model is typically fine-tuned. This fine-tuning process can alter the model's weight matrix to better suit the task. However, in a zero-shot learning scenario, the model's weights stay the same.

### **4.5.1 Subword unit embeddings**

The GPT-2 model employs BBPE for preprocessing input text, breaking it into “subword units” rather than traditional words. BBPE treats each byte as the basic processing unit, so the embedding vector for each subword unit (token) is 1 byte long. Notably, because BBPE splits text into byte-level subword units, a word can be divided into multiple subword units, each with its corresponding embedding vector. Thus the GPT-2 model's vocabulary is generated from the statistical information in the pretraining corpus, accounting for common vocabulary and word structures.

The length of the input token sequence in the GPT-2 model varies with its version. As shown in **Fig. 4.9**, the input token sequence length is 768 in the smaller GPT-2 model (i.e., GPT-2 small), while in larger models, it can extend to 1024 or 2048.

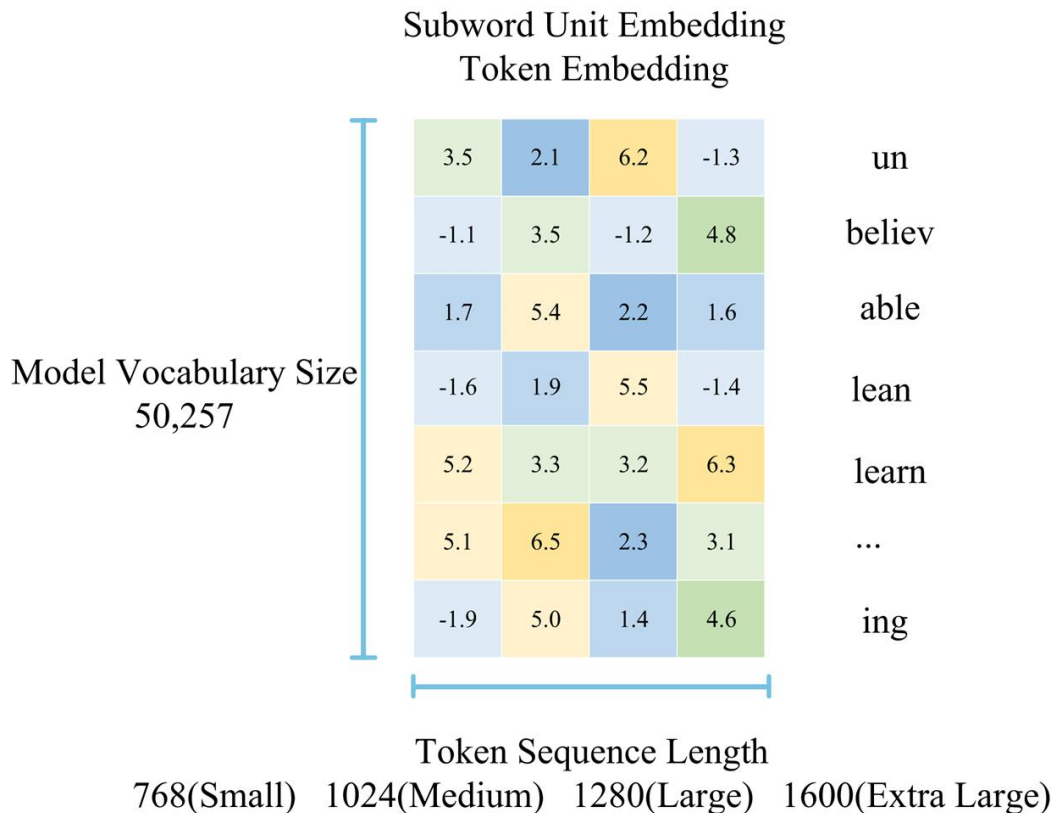


**FIGURE 4.9** Token embedding.

The model vocabulary size is 50,257. The token sequence length is shown as 768 (Small), 1024 (Medium), 1280 (Large) and 1600 (Extra Large). The embeddings are represented in a matrix with values such as 3.5, 2.1, 6.2, negative 1.3, negative 1.1, 3.5, negative 1.2, 4.8, 1.7, 5.4, 2.2, 1.6, negative 1.6, 1.9, 5.5, negative 1.4, 5.2, 3.3, 3.2, 6.3, 5.1, 6.5, 2.3, 3.1, negative 1.9, 5.0, 1.4, 4.6. The subwords listed are un, believ, able, lean, learn, dot dot dot, ing.

During GPT-2's pretraining, the embedding representations of the vocabulary are trained through autoregressive prediction on the input sequence. This

means that the model adjusts the embedding representations of tokens in the vocabulary based on the context and prediction targets to optimize prediction performance. This training process updates the embedding parameters, allowing them to better capture semantic and syntactic information.

It is crucial to note that after pretraining is complete, the vocabulary's embedding representations are fixed when the model is applied and are not updated further. The aim of pretraining is to provide the model with initial parameters that have strong semantic representation capabilities, not to dynamically update the embeddings during actual use. Since the model is pretrained on a large corpus, these embedding vectors typically adapt well to various NLP tasks.

### **4.5.2 Autoregressive process**

GPT-2 is a trained model that includes the decoder weights of the Transformer, the token embedding matrix, and the positional encoding matrix. As a typical autoregressive generative model, GPT-2 can process inputs up to 1024 tokens long and generate sequences in an autoregressive manner starting from the input token. However, the sequences generated in this way often lack meaningful themes. Therefore in practical applications, GPT-2 often uses zero-shot learning, where it receives an instructional prompt (without examples) and generates text around a specific theme.

For instance, the input prompt might be “Large datasets are crucial for training these models.” As shown in **Fig. 4.10**, before the input sequence enters the Transformer model, it needs to be converted into corresponding token representation vectors by adding the word embedding vector to the positional encoding. The Transformer model comprises multiple decoder components, with each layer of the token sequence first processed by a self-attention layer and then by a neural network layer before being passed to the next decoder layer. Although the processing flow of each decoder component is identical, the self-attention and neural network layers of each component have independent weights.

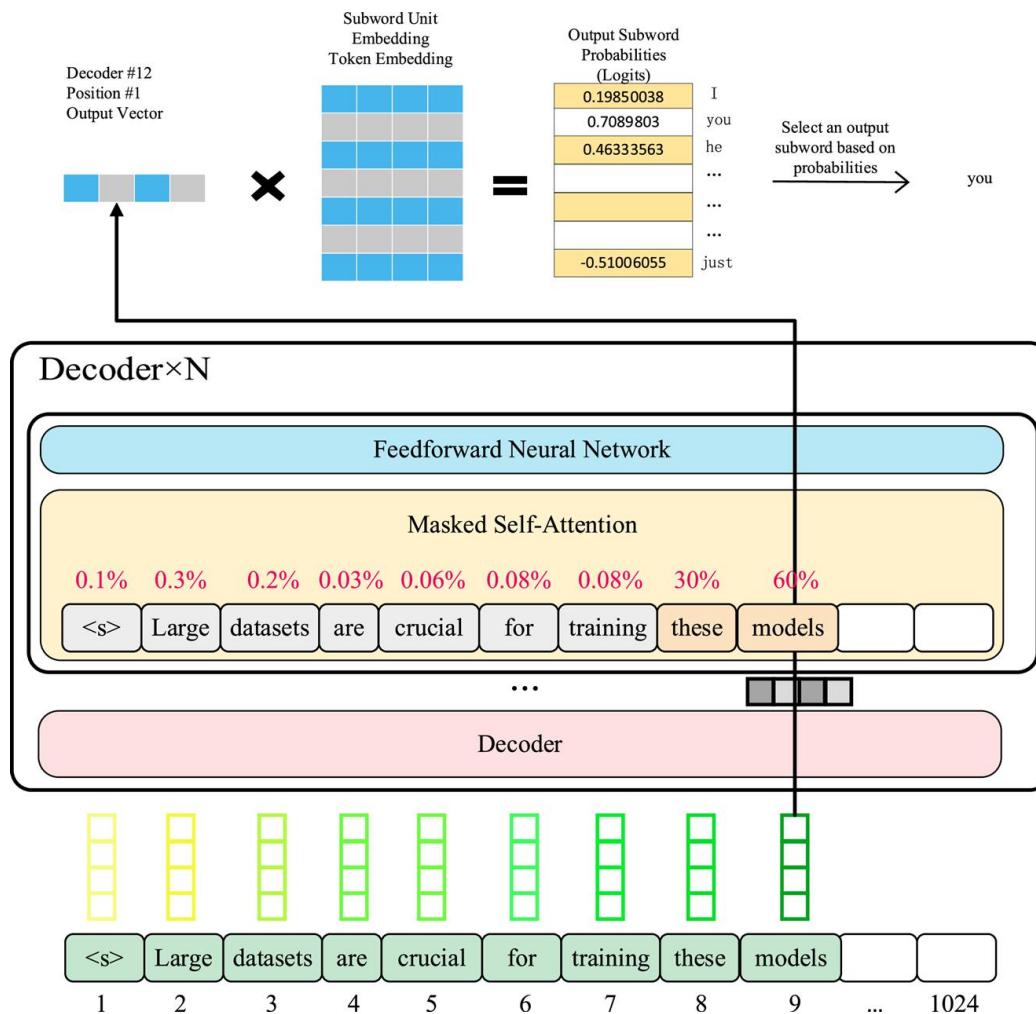


**FIGURE 4.10** Autoregressive generation process of GPT-2.

At the top, there is a representation of Decoder number 12, Position number 1, Output Vector being multiplied by Subword Unit Embedding Token Embedding to produce Output Subword Probabilities (Logits). The probabilities are shown for different subwords like I, you, he and others, with the highest probability subword being selected. Below this, a larger box labeled Decoder times N contains a Feedforward Neural Network and a Masked Self-Attention

layer. The Masked Self-Attention layer shows percentages for different words in a sentence: Large datasets are crucial for training these models. The percentages indicate the attention weights for each word. At the bottom, there is a sequence of words with corresponding positions from 1 to 1024, showing the input sentence Large datasets are crucial for training these models.

When the model's final decoder produces an output vector, this vector is multiplied by the token embedding matrix. The result, after normalization, can be considered the cosine similarity between the two vectors. This implies that the closer the output vector is to a vector in the token embedding matrix, the larger their dot product result will be. The normalized result can be interpreted as the probability or score of selecting a vector from the token embedding matrix. Typically, the word with the highest score is selected ( $\text{top}_k=1$ ). However, to achieve better results, other high-scoring words can also be considered. Thus a common strategy is to perform random sampling based on these scores, increasing the likelihood of selecting high-scoring words. Another common strategy is to set  $\text{top}_k$  to  $N$  and choose the top  $N$  highest-scoring words. This way, the model completes one iteration and outputs a word. The model continues to iterate until it generates the entire context (up to 1024 tokens for GPT-2) or until it generates the token indicating the end of the sequence `<e>`.

## 4.6 Summary

The introduction of GPT-2 did not resolve the ongoing debate over encoder versus decoder architectures, but its success in MTL has offered the research community a fresh perspective. There is now a growing emphasis on effectively utilizing vast amounts of unstructured text data in unsupervised learning environments. Researchers are also looking into optimizing model architectures and refining training strategies to boost performance in MTL and zero-shot learning.

Shortly after GPT-2, OpenAI launched the more advanced GPT-3 model, which has shown exceptional performance across various NLP tasks. Astonishingly, GPT-3 contains 175 billion parameters, an order of magnitude greater than the largest version of GPT-2. Given the impracticality of supervised multitask fine-tuning on such a massive scale, the zero-shot learning approach introduced by GPT-2 has proven crucial for GPT-3, offering new insights and perspectives in the realm of NLP.



---

## CHAPTER 5

# Sparse attention and content-based learning

---

### Abstract

This chapter delves into the architecture and learning strategies of GPT-3, emphasizing its Sparse Transformer design and content-based learning. GPT-3, with 175 billion parameters, introduces a hybrid attention mechanism alternating between dense and locally banded sparse attention patterns, optimizing its efficiency in handling long sequences. The chapter explores the Sparse Transformer's ability to reduce computational complexity, enhancing GPT-3 scalability and performance across various natural language processing tasks. Additionally, it discusses metalearning and in-context learning as key strategies for task adaptation without extensive retraining. The chapter also covers Bayesian inference as a framework for understanding contextual learning and introduces the chain-of-thought approach, which enhances reasoning capabilities through step-by-step problem-solving techniques. These innovations enable GPT-3 to

perform complex reasoning tasks effectively, demonstrating its advanced capabilities in natural language understanding and generation.

## **Keywords**

GPT-3; Sparse transformer; content-based learning; in-context learning; Bayesian inference; chain of thought; meta-learning; sparse attention; few-shot learning

In May 2020, OpenAI unveiled GPT-3, a groundbreaking language model. GPT-3, like its predecessors GPT and GPT-2, is built on a Transformer-based decoder architecture. However, it features an innovative Sparse Transformer pattern that alternates between dense and locally banded sparse attention mechanisms across its layers. This innovation significantly boosts GPT-3's efficiency in processing long sequences.

Boasting an impressive 175 billion parameters, GPT-3 stands as one of the largest Transformer models ever created. Its expanded capacity has notably enhanced its performance across a variety of natural language processing (NLP) tasks. Yet, this scale comes with high training costs—estimated at \$1.4 million by Guosheng Securities—which makes it economically challenging to fine-tune such large models using supervised data for specific tasks. Instead, GPT-3 focuses on in-context learning. In this approach, the model adapts to new tasks using just a few examples, without extensive retraining of its parameters. This allows GPT-3 to quickly assimilate

task-specific knowledge even from minimal data, showcasing remarkable performance on new challenges. This chapter will explore GPT-3 advanced Sparse Transformer pattern and the principles of in-context learning in greater detail.

## 5.1 GPT-3

OpenAI's GPT-3 builds on the foundational architecture of the GPT series, incorporating techniques such as orthogonal initialization, layer normalization, and the byte-level byte-pair encoding (BBPE) tokenization method. These features collectively enhance the training efficiency and generative power of the model. A notable upgrade in GPT-3 lies in its attention mechanism: as depicted in **Fig. 5.1**, it employs a hybrid of dense and locally banded sparse attention within the Transformer's decoder layers. This method, inspired by Sparse Transformers, deviates from the traditional fully connected attention used in GPT-2. Such improvements allow GPT-3 to handle longer sequences more efficiently, reducing both computational demands and memory footprint, thereby boosting the model's scalability.



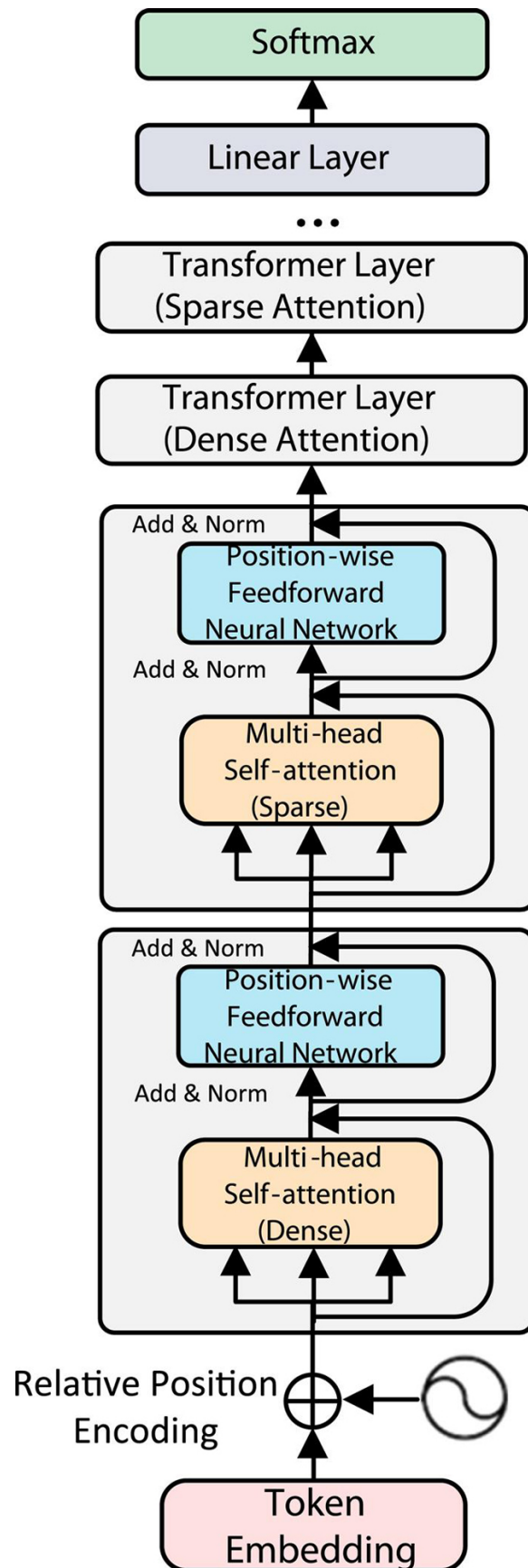


**FIGURE 5.1** Diagram of the GPT-3 model architecture.

The neural network architecture starting with Token Embedding. Next is Relative Position Encoding, followed by Add and Norm. Then, there is a Transformer Layer (Dense Attention) with Multi-head Self-attention (Dense) and Position-wise Feedforward Neural Network. Another Add and Norm follows, leading to a Transformer Layer (Sparse Attention) with Multi-head Self-attention (Sparse) and Position-wise Feedforward Neural Network. The final components are Linear Layer and Softmax.

GPT-3, with its colossal 175 billion parameters, dwarfs GPT-2, which has only 1.5 billion. This vast increase in parameters has enabled GPT-3 to outperform on numerous tasks. Both models share a Transformer-based decoder structure, encompassing input, embedding, Transformer layers, and an output stage. Despite adopting GPT-2's orthogonal initialization, layer normalization, and reversible tokenization, GPT-3 innovates with alternating dense and locally banded sparse attention across its Transformer layers. Additionally, GPT-3 extensive training dataset further boosts its ability to generalize across diverse NLP tasks. The architecture of GPT-3 includes the following:

1. Input and BBPE tokenization: Utilizing BBPE, a method that efficiently reduces vocabulary size by

- merging frequent word pairs into single units.
2. Embedding layer: This layer maps discrete tokens into continuous vector representations.
  3. Positional encoding: Adds vital positional data to vectors, enabling the model to differentiate word positions and understand contextual relationships.
  4. Transformer layers: Each layer is a stack of encoders comprising a sparse self-attention mechanism, alternating between dense and sparse patterns, and a feedforward network for abstracting higher-level features.
  5. Layer normalization: Integrated at every Transformer layer's input and output, it normalizes these points to stabilize training and speed up convergence.
  6. Output layer: Transforms Transformer outputs into a probability distribution via a linear layer and a softmax function, predicting the next token by selecting the highest probability.

The primary distinction between GPT-3 and GPT-2 lies in the implementation of sparse attention within the model architecture. Additionally, GPT-3 adopts many of the design elements and techniques used in GPT-2. The significant increase in both the scale of GPT-3 and the size of its training dataset have contributed to its enhanced performance.

## **5.2 The sparse transformer**

The Sparse Transformer represents a specialized adaptation of the standard Transformer model, designed to optimize the processing of long sequences. This model incorporates a sparse attention mechanism that cleverly cuts down on both computational complexity and memory demands, facilitating efficient management of lengthy data streams.

### **5.2.1 Characteristics of the sparse transformer**

Traditional Transformer models struggle with processing long sequences due to the inherent fully connected setup of the self-attention mechanism, which causes both computational complexity and memory usage to scale quadratically with the sequence length, expressed as  $O(n^2)$ . This level of complexity becomes prohibitive for extensive sequences. The Sparse Transformer addresses this issue by not calculating attention scores for every pair of elements, thus enabling a more computationally sparse approach. There are two main strategies for this optimization: first, by limiting the reach of the attention mechanism with a defined attention window; second, by employing specific sparse connection patterns that can reduce the computational load and memory requirements to  $O(n)$ .

#### **5.2.1.1 Sparse attention patterns**

Illustrated in **Fig. 5.2**, the Sparse Transformer capitalizes on sparse attention patterns. Under this framework, each input token selectively focuses on a subset of contextual



information instead of every possible token, dramatically cutting the computational complexity from  $O(n^2)$  to  $O(n)$ . Sparse attention can be implemented in several ways, such as local banded attention, which concentrates on closely positioned tokens, and strided (sparse fully connected) attention, which samples the sequence at fixed steps throughout its entire length.

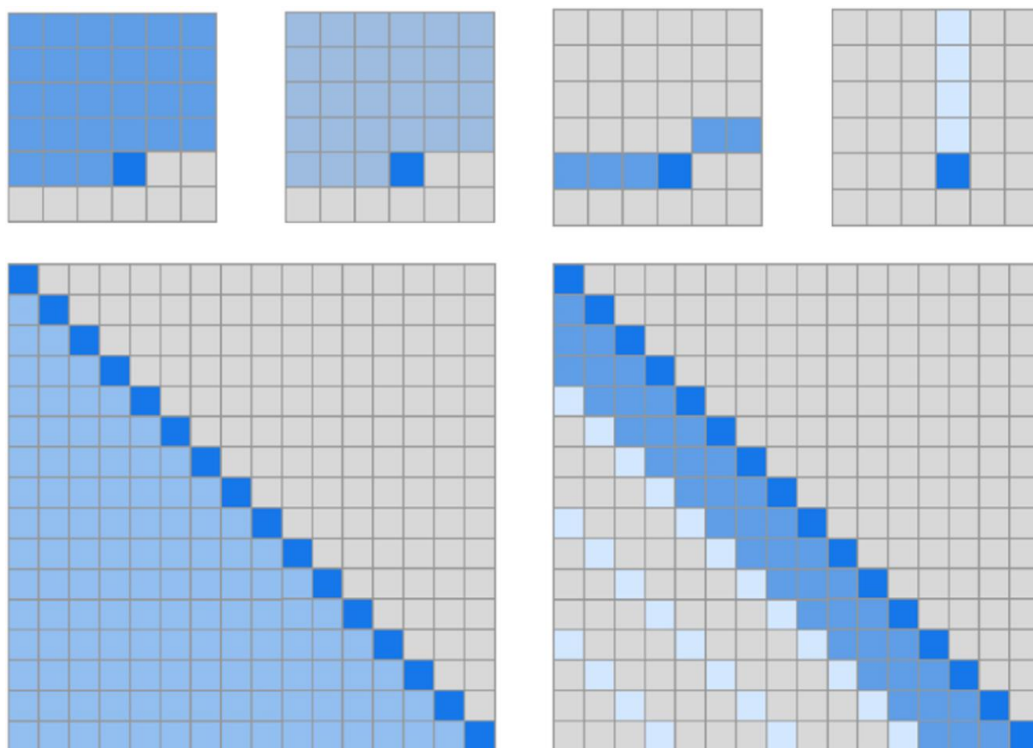


**FIGURE 5.2** Traditional transformer (left) and sparse transformer (right).

The Sparse Transformer capitalizes on sparse attention patterns. The image shows six grid patterns with different highlighted areas, representing various sparse attention patterns. The top row has four smaller grids and the bottom row has two larger grids. Each grid has a unique pattern of highlighted squares.

### 5.2.1.2 Alternating dense and sparse attention patterns

GPT-3 enhances its performance and generalizability by alternating between dense and sparse attention patterns in different Transformer layers, leveraging principles from

Sparse Transformer designs. This strategic variation enables the model to effectively capture long-range dependencies without sacrificing computational efficiency. Practically, the balance between dense and sparse patterns can be tailored to fit the specific requirements of a task and the available computational resources.

### **5.2.1.3 Learnable relative positional encodings**

GPT-3 employs learnable relative positional encodings that account for both the absolute and relative distances between tokens. This innovative approach significantly bolsters the model's capability to understand and process long-range dependencies.

### **5.2.2 Local banded attention**

Local banded attention in GPT-3 is designed to reduce the computational load and memory usage typical of traditional Transformer models. This sparse attention mechanism focuses each input token on just a portion of its neighbors instead of every token in the sequence. The implementation involves several key steps:

1. Determine window size: First, establish a window size that dictates the range of context each token considers. This parameter can be adapted to fit the needs of the task and the available computational power. Larger window sizes provide richer contextual details but require more computational resources.

2. Build the attention matrix: Construct a one-dimensional attention vector for each token that contains the weights for the adjacent tokens within its window. This process mirrors the conventional self-attention approach but is confined to the specified window. These vectors are then stacked to form a two-dimensional attention matrix.
3. Execute local banded attention: Multiply the attention matrix by the input sequence's hidden states to generate outputs that include contextual information for each token, focusing only on immediate neighbors.

Local banded attention notably reduces the computational complexity from  $O(n^2)$  to  $O(n \times \text{window size})$ , efficiently balancing performance and contextual awareness. However, it might struggle with long-range dependencies. To overcome this limitation, it can be integrated with other sparse attention mechanisms, such as sparse fully connected attention, or used alternately with dense attention patterns across different Transformer layers.

### **5.2.3 Cross-layer sparse connections**

Cross-layer sparse connections in GPT-3 are designed to forge longer-range dependencies between layers, optimizing the model's handling of extended sequences. This approach improves the attention mechanism's efficiency, enabling processing of larger inputs. GPT-3

implements cross-layer sparse connections through a blend of dense and locally banded sparse attention patterns, detailed as follows:

1. Local banded sparse connections: These connections focus each token on a nearby subset of tokens, facilitating the capture of immediate contextual relationships within a constrained scope.
2. Dense connections: Contrasting with the local approach, dense connections allow each token to engage with every other token in the sequence, broadening the scope to global dependencies.

Alternating these connection styles throughout the model layers, GPT-3 adeptly manages both near and far dependencies. The local banded approach targets adjacent layer interactions for nearer relationships, while dense connections bridge longer distances. This structured versatility not only boosts GPT-3 efficiency in processing lengthy sequences but also extends its capacity to recognize extensive relational patterns.

Furthermore, GPT-3 employs these cross-layer sparse connections to enhance its overall effectiveness in NLP tasks, simultaneously conserving computational resources. It achieves this through a strategic alternation of dense and sparse Transformer layers, where dense attention and local banded sparse attention layers interchange, optimizing both performance and computational efficiency.

By integrating sparse attention mechanisms, alternating attention patterns, and adaptive positional encodings, GPT-3 significantly alleviates the challenges faced by traditional Transformers with long sequences, positioning Sparse Transformer as a robust framework for extensive data handling. The model's specific attention strategies, akin to those in Sparse Transformer, significantly elevate its capability in processing long texts.

## **5.3 Meta-learning and in-context learning**

Due to the vast number of parameters in GPT-3 (175 billion), fine-tuning every single one for different downstream tasks has become unfeasible. Consequently, metalearning and in-context learning have emerged as the key strategies for leveraging this model. Meta-learning involves pretraining the model on an extensive text corpus to enhance its natural language generation skills and overall linguistic knowledge and comprehension.

Throughout this phase, GPT-3 absorbs the intricacies of grammar across different language elements—words to paragraphs—and garners a foundational understanding of common knowledge.

In-context learning, alternatively, focuses on making slight adjustments to the model using a small subset of labeled data to tailor it to specific tasks. This method of fine-tuning doesn't retrain the model extensively; instead, it leverages the broad capabilities developed during

pretraining to meet the needs of particular applications. Examples of such data include question-answer pairs or sentiment-labeled sentences. This approach allows GPT-3 to achieve high performance in a variety of NLP tasks with just minimal labeled data.

### **5.3.1 Meta-learning**

In the GPT-3 paper, the authors reinterpreted the concept of metalearning, differentiating it from its traditional definition. Traditionally, metalearning refers to training a model to quickly learn new tasks with a small amount of labeled data. However, the GPT-3 paper expands the definition of metalearning to include the process of unsupervised pretraining.

In this context, large pretrained language models like GPT-3 itself are considered metalearners because they have already acquired a vast amount of knowledge and language understanding during the pretraining phase. By pretraining on a large corpus of unlabeled text data, the model learns the structure of language and common-sense knowledge, thus gaining some generalization ability.

This new interpretation of metalearning is viable because GPT-3 has demonstrated strong generalization capabilities during its pretraining phase. This enables GPT-3 to perform excellently on new tasks with minimal labeled data (or even no data), similar to traditional metalearning methods. However, in GPT-3's case, this capability primarily stems from the unsupervised pretraining phase, rather than from

a specialized training process tailored for metalearning tasks.

### 5.3.2 In-context learning

To minimize adjustments to its parameters during the fine-tuning phase, GPT-3 focuses more on in-context learning strategies. In the task adaptation stage, the model adopts zero-shot, one-shot, and few-shot learning approaches, circumventing the constraints associated with traditional fine-tuning techniques.

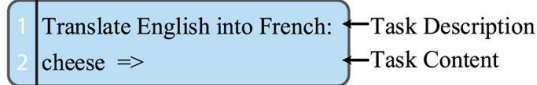
As depicted on the right side of **Fig. 5.3**, the fine-tuning process is initiated after the model's pretraining phase is complete. For each specific task, a subset of training samples is provided, such as the single-sample batch used in this instance. By computing the loss and updating its weights, the model is able to achieve commendable performance on new tasks. Fine-tuning generally demands fewer data compared to starting from scratch and permits lower learning rates. This efficiency is possible because fine-tuning starts from a pretrained model's parameters, which are already close to the optimal solution for the tasks at hand.



### Content-based Learning in GPT-3

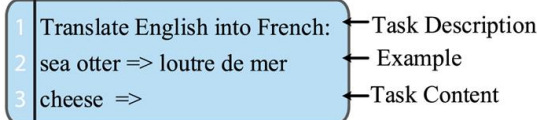
#### Zero-shot Learning:

The model predicts answers solely based on a natural language description of the given task, without performing gradient updates.



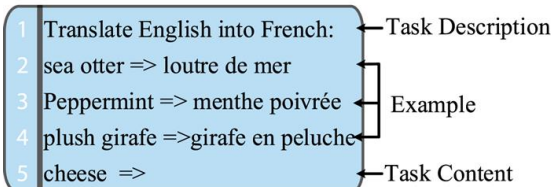
#### One-shot Learning:

In addition to the task description, a single example is provided, without performing gradient updates.



#### Few-shot Learning:

In addition to the task description, the model is exposed to some task examples, without performing gradient updates.



### Conventional Fine-tuning

#### Fine-tuning:

The model is trained through repeated gradient updates using a large number of example tasks.

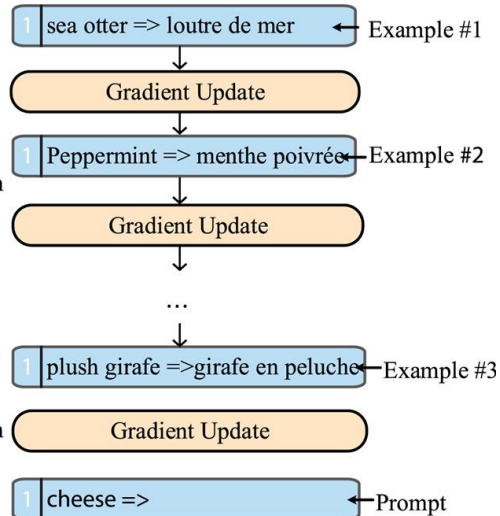


**FIGURE 5.3** Three different types of contextual learning versus traditional fine-tuning.

The model predicts answers solely based on a natural language description of the given task, without performing gradient updates. Task Description: Translate English into French colon Task Content colon cheese becomes cheese. One-shot Learning colon In addition to the task description, a single example is provided, without performing gradient updates. Task Description colon Translate English into French colon Example colon sea otter becomes loutre de mer Task Content colon cheese becomes cheese. Few-shot Learning colon In addition to the task description, the

model is exposed to some task examples, without performing gradient updates. Task Description colon Translate English into French colon Example colon sea otter becomes loutre de mer Example colon Peppermint becomes menthe poivrée Example colon plush giraffe becomes girafe en peluche Task Content colon cheese becomes cheese. Conventional Fine-tuning colon Fine-tuning colon The model is trained through repeated gradient updates using a large number of example tasks. Example number one colon sea otter becomes loutre de mer Gradient Update Example number two colon Peppermint becomes menthe poivrée Gradient Update Example number three colon plush giraffe becomes girafe en peluche Gradient Update Prompt colon cheese becomes cheese.

**Fig. 5.3** on the left illustrates how GPT-3 employs varying degrees of example-based in-context learning techniques—zero-shot, one-shot, and few-shot—to boost its performance on new tasks without modifying its parameters. This strategy greatly simplifies the use of large language models, making it easier for users to apply the model across diverse tasks with minimal setup.

Zero-shot learning in GPT-3 involves tackling tasks without prior exposure to any related labeled data, relying entirely on knowledge gleaned from its extensive pretraining. In this mode, the model must interpret the task’s description and generate appropriate responses

without example prompts. The main hurdle here is enabling the model to grasp the task requirements thoroughly and produce suitable responses solely based on its pretrained intelligence.

GPT-3 prowess in zero-shot learning stems from its colossal architecture, which incorporates up to 175 billion parameters, and its pretraining on vast quantities of unlabeled text. This extensive background allows it to understand intricate language patterns and common-sense contexts, enabling it to generate pertinent outputs without specific training on the task at hand.

One-shot learning, by contrast, involves the model making inferences from a single related example. Here, the challenge lies in empowering the model to quickly decode the task's demands from just one example and respond accurately, demonstrating the model's ability to adapt swiftly based on minimal data.

In the one-shot learning scenario illustrated in **Fig. 5.3**, GPT-3 uses a single example to bridge the gap between the task description and execution. This requires the model to process longer sequences than in zero-shot scenarios, extracting valuable information to improve its performance on translation tasks. Importantly, no gradients are calculated or weights updated during this process.

GPT-3 achieves high performance on one-shot learning tasks by leveraging the knowledge acquired during its pretraining phase, combined with the given single example. This demonstrates GPT-3 ability to rapidly adapt to new

tasks with minimal labeled data, thus reducing the training costs and time required for supervised learning on specific tasks.

Few-shot learning involves training the model to make inferences after being provided with a small number of labeled samples related to the target task. In this case, the model needs to understand the specific requirements of the task based on several provided examples and generate the correct answer accordingly. The challenge of few-shot learning is how to enable the model to effectively learn and adapt to new tasks from limited information while improving prediction accuracy. Few-shot learning extends one-shot learning by providing several examples after the task description. However, the processing capacity of the GPT-3 model is limited, so submitting overly long text sequences for few-shot learning may not facilitate the extraction of useful information. This explains why context learning with too many examples is not widespread in GPT-3 models.

GPT-3 also excels in few-shot learning tasks. By taking multiple task-related examples as input, GPT-3 can better adapt to new tasks, achieving high performance across various NLP tasks. This indicates GPT-3 strong generalization ability, capable of producing high-quality outputs with a small amount of labeled data.

Compared to zero-shot, one-shot, and few-shot learning, traditional fine-tuning methods focus on supervised learning of the pretrained model with a large amount of

labeled data. While this approach has achieved significant results in many NLP tasks, its limitations include the need for extensive labeled data and computational resources (to adjust model parameters). Moreover, fine-tuning on specific tasks may lead to overfitting, limiting the model's generalization capability.

In contrast, zero-shot, one-shot, and few-shot learning strategies offer clear advantages. First, since GPT-3 has already acquired a vast amount of knowledge and language understanding during its pretraining phase, it does not require costly supervised learning on specific tasks. This significantly reduces training costs and time. Secondly, GPT-3 can perform well with only a small amount of labeled data, making it easier to adapt to a variety of NLP tasks, especially those where large amounts of labeled data are difficult to obtain.

## **5.4 Bayesian inference of concept distributions**

Contextual learning is a key emergent capability of large language models like GPT, and grasping its principles is essential for crafting effective prompts. **Fig. 5.2** illustrates this point: prompts for zero-shot learning include only a task description and content, while those for one-shot and few-shot learning also incorporate examples.

### **5.4.1 Implicit fine-tuning**

Language models are interpreted as metaoptimizers, treating contextual learning as a form of implicit fine-tuning—an approach that establishes a link between GPT’s context-dependent learning and traditional explicit fine-tuning methods. Illustrated in **Fig. 5.4**, GPT produces metagradients from forward computations related to the target task labels and utilizes these gradients within the given task through its attention mechanism. This metaoptimization mirrors the fine-tuning process, where model parameters are explicitly updated using backward propagation gradients.

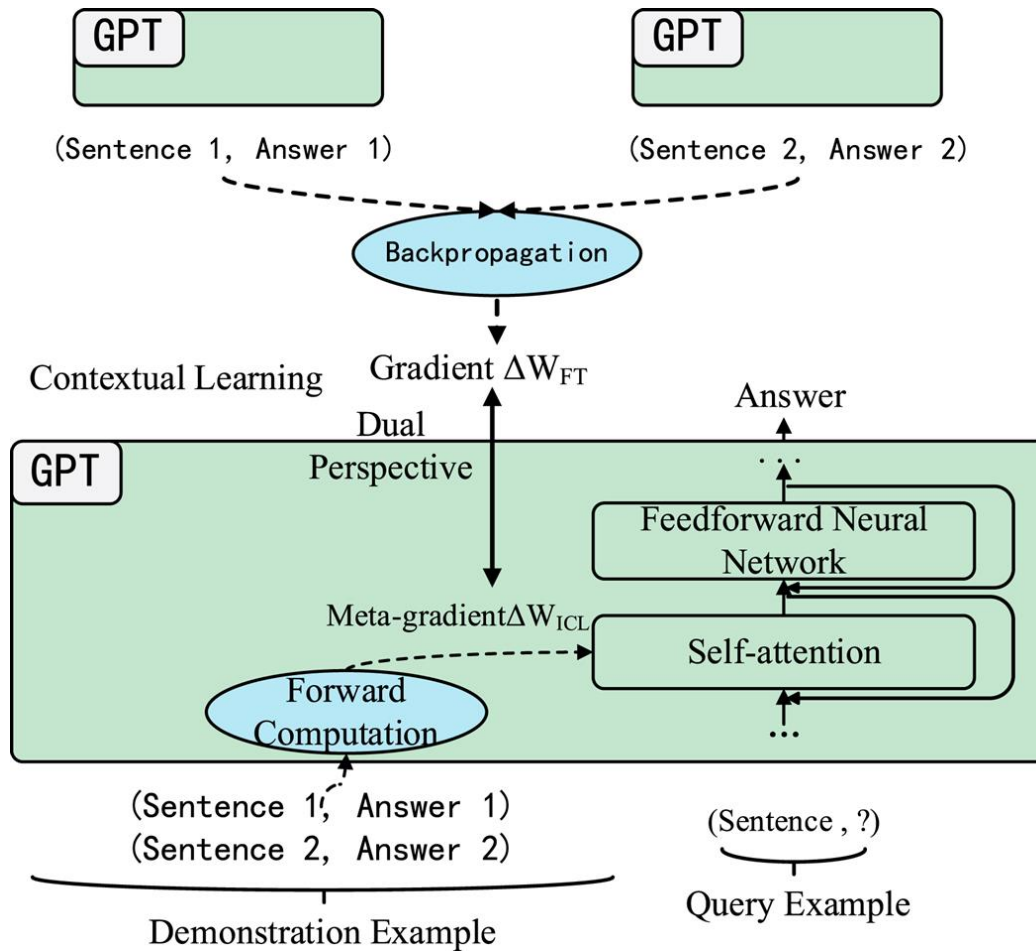


**FIGURE 5.4** Process of meta-optimization and fine-tuning in GPT.

At the top, there are two GPT models labeled with Sentence 1, Answer 1 and Sentence 2, Answer 2. These are connected to a Backpropagation process, which generates Gradient  $\Delta W_{FT}$ . This gradient is used in the Contextual Learning process within a larger GPT model. The larger GPT model includes components such as Perspective, Feedforward Neural Network and Self-attention. The Forward Computation process generates Meta-gradient  $\Delta W_{ICL}$ . Demonstration

Examples (Sentence 1 prime, Answer 1) and (Sentence 2, Answer 2) are fed into the Forward Computation process. A Query Example (Sentence, question mark) is also fed into the model to generate an Answer.

At various levels—predictive accuracy, representational depth, and attention dynamics—contextual learning and explicit fine-tuning are akin, both rooted in gradient descent optimization. Yet, the key distinction lies in how these gradients are produced: contextual learning relies on forward computations for metagradients, whereas fine-tuning uses backpropagation.

In few-shot learning, do large language models like GPT implicitly learn a function  $y=f(x)$  when presented with a small set of task-relevant examples? If this is true, these sample examples  $\langle x_i, y_i \rangle$  have a profound influence on context-based learning performance. As illustrated in **Fig. 5.5** from the original GPT-3 paper, without any examples, the model struggles to learn or adjust, but performance significantly improves in scenarios with one or a few examples.



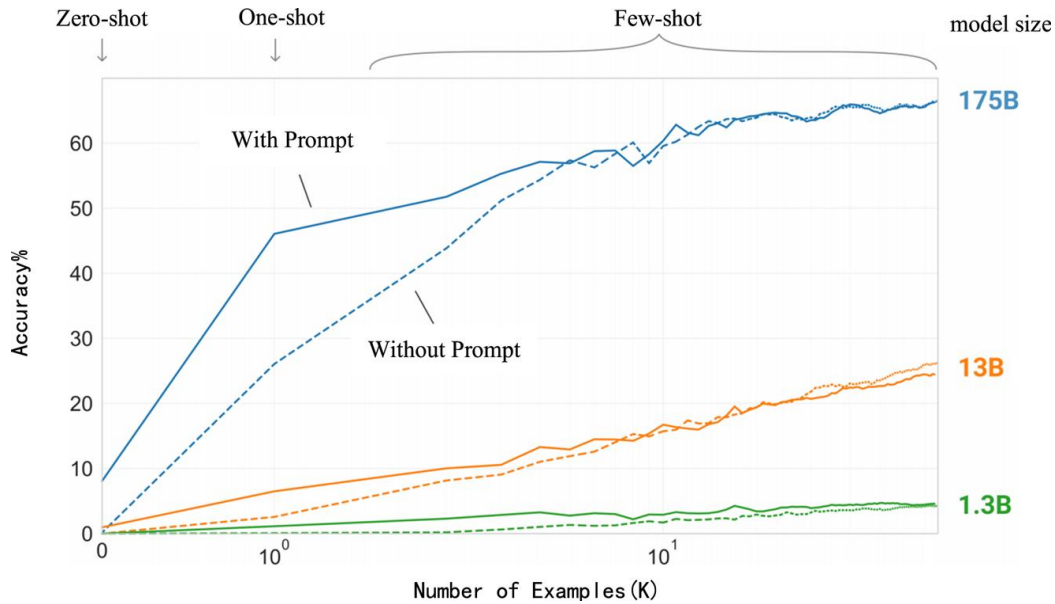


**FIGURE 5.5** Impact of sample size and parameter scale on model performance. Image source: Paper “Language models are few-shot learners”.

The x-axis represents the number of examples (K) on a logarithmic scale from 0 to 10<sup>1</sup> end superscript. The y-axis shows accuracy percentage from 0 to 70 in increments of 10. Three model sizes are compared: 175B, 13B and 1.3B. Each model size has two lines: one solid line for With Prompt and one dashed line for Without Prompt. The 175B model (top lines) shows the highest accuracy, starting around 10 percent for zero-shot and reaching about 70 percent for few-shot learning. The With Prompt line consistently outperforms Without Prompt. The 13B model (middle lines) shows moderate performance, starting near 0 percent for zero-shot and reaching about 25 percent for few-shot learning. Again, With Prompt performs

better than Without Prompt. The 1.3B model (bottom lines) shows the lowest accuracy, barely rising above 0 percent even with many examples. The difference between With Prompt and Without Prompt is minimal for this model size. The graph is labeled with Zero-shot, One-shot and Few-shot at appropriate points along the x-axis, indicating different learning scenarios. Overall, the graph demonstrates that larger models perform better and that using prompts generally improves accuracy across all model sizes and learning scenarios.

However, recent research offers a fresh view on the significance of these examples in contextual learning. The study reveals that the input-label mapping within these examples affects model performance less than previously thought. Surprisingly, when actual labels are swapped with random ones or when examples are replaced with random values, the loss in performance is relatively minor. This finding challenges the intuitive belief in few-shot learning's reliance on accurate input-label mappings in examples.

**Fig. 5.6** showcases how large language models like GPT perform on downstream tasks with no examples, correct labels, and random labels. This figure highlights two distinct inference methods used for performance evaluation: direct and channel. In direct inference, the model assesses the probabilities of different categories based on the given context. In contrast, channel inference calculates the probabilities of the context occurring under

specified categories. For instance, in sentiment analysis, for the context  $x =$  "This movie is great," Direct inference evaluates the likelihood of "This movie is great positive" versus "This movie is great negative." Channel inference, however, assesses "positive This movie is great" against "negative This movie is great." The data illustrates that, both in classification (upper part of the figure) and multiple-choice tasks (lower part), models with random labels (backslash) perform comparably to those with correct labels (net-like) and distinctly outperform the no-example zero-shot cases (gray).

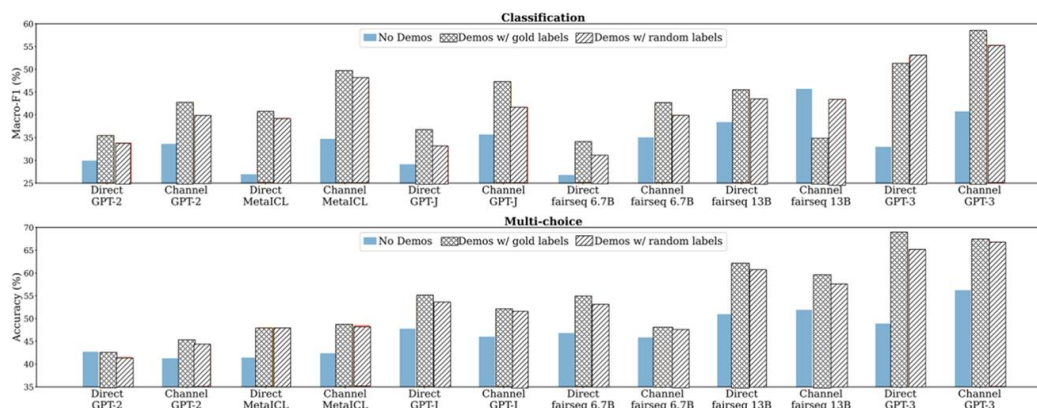


**FIGURE 5.6** Performance of different labels in downstream tasks.

For the Classification graph: The x-axis lists different models: Direct GPT-2, Channel GPT-2, Direct MetaICL, Channel MetaICL, Direct GPT-J, Channel GPT-J, Direct fairseq 6.7B, Channel fairseq 6.7B, Direct fairseq 13B, Channel fairseq 13B, Direct GPT-3 and Channel GPT-3. The y-axis represents Accuracy (percent) from 25 to 60 in increments of 5. For each model, there are three bars representing: 1. No Demos 2. Demos w/ gold labels 3. Demos w/ random labels The Multi-choice graph has the same x-axis layout as the Classification graph. The y-axis for Multi-choice represents Accuracy (percent) from 35 to 70 in increments of 5. For both graphs, the Demos w/ gold labels generally show the highest accuracy, followed by Demos w/ random labels and No Demos typically showing the lowest accuracy across most models. The Channel GPT-3 model appears to have the highest accuracy in both Classification and Multi-choice tasks across all three demo conditions. The graphs illustrate how different language

models perform on classification and multi-choice tasks under various demo conditions, showcasing the impact of providing examples (demos) with correct (gold) or random labels compared to no examples at all.

According to **Fig. 5.6**, several insights emerge:

1. The definition of label spaces and the distribution of input texts set by sample examples are pivotal for contextual learning, regardless of the accuracy of individual labels. The distributions of  $x$  and  $y$ —the input texts and their potential corresponding answers—play a major role in influencing contextual learning outcomes. In scenarios with random labels, such as  $\langle x_i, y_i \rangle$ , even if  $y_i$  is incorrect, it remains within the set of possible answers  $\mathcal{Y}$ . Altering these distributions, like substituting  $\mathcal{Y}$  with options outside the expected answer set, would drastically impair the learning process.
2. The specification of the overall format is equally vital. In contexts where the label space is undefined, using random English words as labels proves far more effective than not using any labels at all.
3. Employing a contextual learning objective in metatraining enhances these effects, encouraging the model to focus more on the straightforward elements of the sample examples, like their format,

rather than on the complexities of input-label mapping.

Large language models might not derive a specific mapping function  $y=f(x)$  from examples for tackling new tasks; instead, they often tap into their extensive pretrained knowledge. Yet, if we broaden the definition of task learning to encompass adapting to specific inputs, label distributions, and formats implied by examples—and refining predictions accordingly—then it’s plausible that these models do indeed extract task-relevant information from examples. Pre-trained language models come equipped with an inherent ability to execute tasks, demonstrating that language modeling goals inherently support robust zero-shot capabilities. However, this also implies that contextual learning might struggle with tasks where the necessary input-label relationships are not preencoded.

The influence of sample examples and instructional cues on language models appears to be analogous. This insight applies to instruction-following models too, indicating that such models reactivate existing capabilities rather than learning new semantic nuances of tasks. Experimental findings further reveal that pairing each unlabeled input with a random label and using it as an example can achieve performance nearing that of few-shot scenarios, even without labeled data. This suggests that our current benchmarks for zero-shot capabilities might be set significantly lower than actual potential, marking an

important avenue for future enhancements in zero-shot performance.

### **5.4.2 Bayesian inference**

Using a Bayesian framework to interpret contextual learning offers a clearer understanding of few-shot learning principles. This approach posits that during extensive language pretraining, elements like sentences, paragraphs, and tables within a document share underlying semantic themes and structured formats—for example, how questions and answers are alternately presented in Q&A formats. The training data at the document level embody latent concepts of coherence over long distances, which the model taps into during pretraining to deduce these underlying ideas.

Contextual learning, within this Bayesian framework, is seen as a problem of conditional probability, where the task is to predict the likelihood of the next word or phrase based on the pretrained model’s knowledge and a sequence of prompts. The “knowledge of the pretrained model” corresponds to what is learned about cross-document coherence during the extensive text pretraining, which becomes part of the model’s internalized understanding of the world and lays the groundwork for its context comprehension. The “series of prompt information” pertains to the specific task-oriented concept distributions learned through contextual learning with training samples,

serving as critical adjustments that refine the model’s task-specific responses.

Together, the pretrained and prompted distributions underpin contextual learning in the Bayesian paradigm, offering a foundational world understanding while specifically honed for particular tasks. Bayesian inference applied to the concepts introduced in few-shot learning enhances the learning process. Successfully deducing these concepts enables the model to accurately predict outcomes in test scenarios.

$$p(\text{output}|\text{prompt}) = \int_{\text{concept}} p(\text{output}|\text{concept}, \text{prompt}) p(\text{concept}|\text{prompt}) d(\text{concept}) \quad (5.1)$$

**Eq. (5.1)** illustrates that prompts serve as evidence, bolstering the certainty of the posterior probability distribution  $p(\text{concept}|\text{prompt})$ . If this distribution is mainly focused on the prompted concepts, the model can effectively “learn” from the prompts, thus enhancing its likelihood of selecting the relevant prompted concepts accurately.

There’s a logical challenge, though: the language model derives prompted concepts based on context examples, yet these prompts are drawn from a prompt distribution that may greatly differ from the model’s training pretraining distribution. Prompts connect independent examples, making the transitions between different examples under the language model and its pretraining distribution



unlikely, potentially introducing noise during inference. For example, concatenating independent sentences from different news topics could result in unusual texts due to the lack of sufficient context in any individual sentence. Yet, as demonstrated in GPT-3, despite these distributional discrepancies, the language model can still execute Bayesian inference effectively.

Effective contextual learning by the language model is feasible as long as the signal's strength surpasses the noise's disruption. Here, "signal" refers to the KL divergence between the prompted concept and other concepts under the given prompt, while "noise" is the error from transitions between examples. If the prompts make it straightforward for the model to differentiate between the prompted concept and others, the signal is considered strong. This indicates that removing certain information (like eliminating input-output mappings) is permissible if the prompt's format is constant and the mapping information is already present in the pretrained data, a stark contrast to traditional supervised learning, which fails without intact input-output mappings (e.g., when labels are randomized).

This further clarifies the discussion on implicit fine-tuning earlier, noting that language models are not dependent on the specific  $\langle x_i, y_i \rangle$  pairings in prompts. The model can engage in contextual learning as long as the output  $y_i$  is part of the candidate set  $y$ , even if it doesn't directly correspond to  $x_i$ . This indicates that in contextual

learning, every aspect of the prompt—including input distribution, output space, and format—acts as a “signal” that aims to define the task’s conceptual distribution, thus enhancing the model’s ability to accurately infer the concepts learned during pretraining. While randomizing input-output mappings in the prompts introduces “noise” by linking random sequences, the model can still undertake Bayesian inference effectively if there is a sufficient signal, like the correct input distribution, output space, and format. Proper input-output mappings can further facilitate this process by offering more precise evidence and minimizing noise, particularly when such mappings are rare in pretrained data.

The Bayesian framework thus illuminates the underlying mechanics of contextual learning and offers guidance on how to design prompts and their elements—input distribution, output space, and format. These components help define the task’s conceptual distribution through the prompts, enabling the model to more accurately infer the concepts learned during pretraining.

## **5.5 Thought chains**

To further boost large language models’ abilities in mathematical reasoning, members of the Google Brain team, including Jason Wei and Xuezhi Wang, introduced the chain-of-thought (CoT) approach in January 2022. This method employs a divide-and-conquer strategy tailored to various contexts of learning—zero-shot, one-shot, and few-

shot—training the models to mimic human-like progressive thinking and reasoning. This technique grants models the capacity to tackle complex reasoning challenges effectively.

The CoT's most straightforward implementation involves incorporating auxiliary step-by-step instructions into prompts, a method known as zero-shot learning CoT. Despite its simplicity, it has shown effectiveness across many fields. This approach involves two phases (refer to **Fig. 5.7**): initially, the prompt “Let’s think about this step-by-step” is added to a zero-shot prompt, prompting the model to output its reasoning process. In the second phase, this reasoned output is followed by another prompt, “Therefore, the answer (in Arabic numerals) is,” leading the model to deliver the final answer.

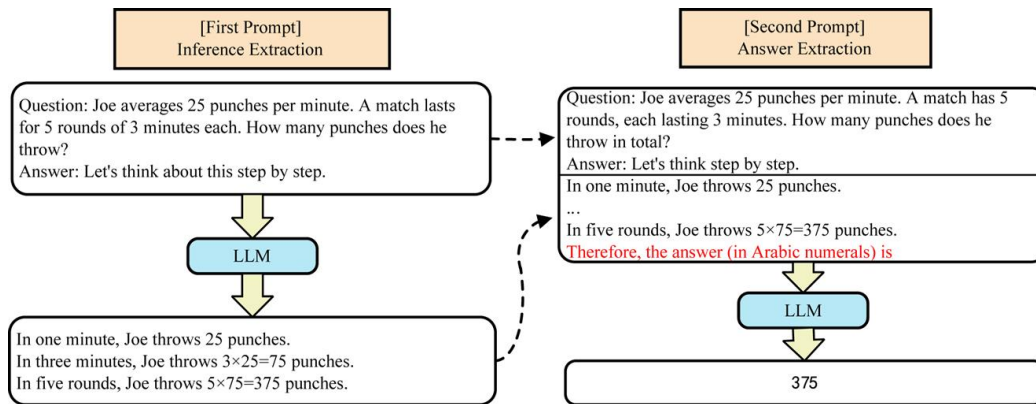


**FIGURE 5.7** Zero-shot learning CoT.

The flowchart shows two prompts: [First Prompt] Inference Extraction and [Second Prompt] Answer Extraction. The first prompt reads: Question: Joe averages 25 punches per minute. A match lasts for 5 rounds of 3 minutes each. How many punches does he throw? Answer: Let's think about this step by step. This connects to a box labeled LLM. Below it, the steps are: In one minute, Joe throws 25 punches. In three minutes, Joe throws 3 times 25 equals 75 punches. In five rounds, Joe throws 5 times 75 equals 375 punches. The second prompt reads: Question: Joe averages 25 punches per minute. A match has 5 rounds, each lasting 3 minutes. How many punches does he throw in total? Answer: Let's think step by step. In one minute, Joe throws 25 punches. In five rounds, Joe throws 5 times 75 equals 375 punches. Therefore, the answer (in Arabic numerals) is. This connects to a box labeled LLM, which then connects to a box with the number 375.

This straightforward technique significantly enhances the performance of large language models on various

reasoning tasks. For instance, on the GSM8K mathematical reasoning dataset, adding this specific prompt improved the reasoning accuracy from an initial 10.4% to 40.4%, as detailed in [Table 5.1](#).

**Table 5.1**

|                            | Mathematical ability   |                     |                       |                       |                    |              |
|----------------------------|------------------------|---------------------|-----------------------|-----------------------|--------------------|--------------|
|                            | SingleEq               | AddSub              | MultiArith            | GSM8K                 | AQUA               | SVAMP        |
| Zero-shot learning         | 74.6/78.7              | 72.2/77.0           | 17.7/22.7             | 10.4/12.5             | 22.4/22.4          | 58.8/58.7    |
| Zero-shot chain-of-thought | 78.0/78.7              | 69.6/74.7           | 78.7/79.3             | 40.4/40.5             | 33.5/31.9          | 62.1/63.7    |
|                            | Common sense           |                     | Other reasoning tasks |                       | Symbolic reasoning |              |
|                            | Common sense questions | Strategic questions | Data understanding    | Out-of-order problems | Last letter        | Coin tossing |
| Zero-shot learning         | 68.8/72.6              | 12.7/54.3           | 49.3/33.6             | 31.3/29.7             | 0.2/-              | 12.8/53.8    |
| Zero-shot chain-of-thought | 64.6/64.0              | 54.8/52.3           | 67.5/61.8             | 52.4/52.9             | 57.6/-             | 91.4/87.8    |

Why does adding “Let’s think about this step-by-step” to prompts enable large language models to elaborate detailed reasoning steps and compute answers? Based on the contextual learning’s Bayesian inference of conceptual distribution discussed earlier, it’s likely because the pretraining data contain an abundance of examples that begin with “Let’s think step by step,” followed by detailed reasoning and culminating in an answer. During pretraining, large language models internalize these patterns. Thus, when prompted with “Let’s think step by step,” these models can more effectively infer the step-by-step reasoning concepts they learned during pretraining, allowing them to methodically reason through tasks and derive answers.

The second application of the CoT approach targets one-shot learning prompts by incorporating manually crafted reasoning examples into the sample scenarios. These examples extend beyond simple question-answer pairs to include detailed, step-by-step reasoning processes tailored to each question. For a detailed illustration of how this is applied, refer to the CoT response depicted in **Fig. 5.8**.

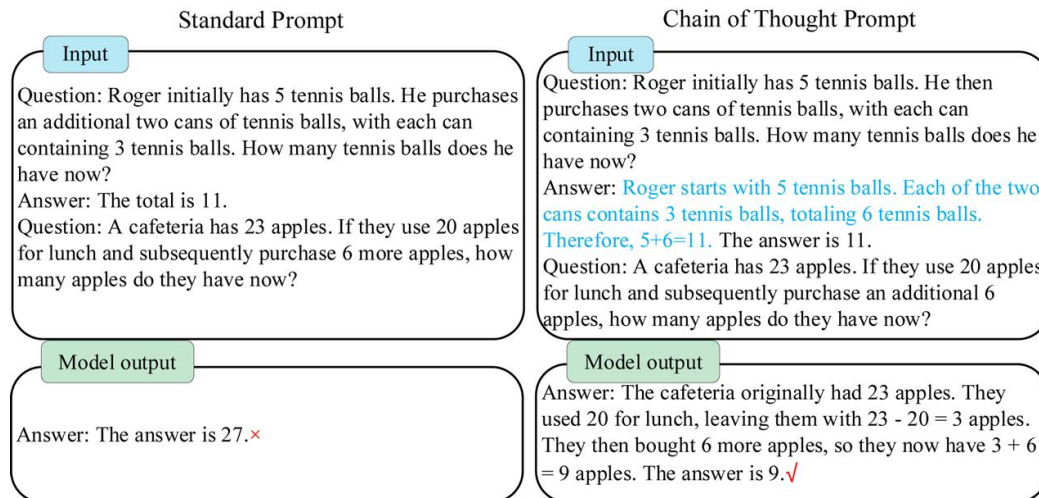


**FIGURE 5.8** Standard prompts versus reasoning chain prompts in single-shot learning.

In the standard prompt, the input question is: Roger initially has 5 tennis balls. He purchases an additional two cans of tennis balls, with each can containing 3 tennis balls. How many tennis balls does he have now? The answer is: The total is 11. The second question is: A cafeteria has 23 apples. If they use 20 apples for lunch and subsequently purchase 6 more apples, how many apples do they have now? The model output is: Answer: The answer is 27. In the chain of thought prompt, the input question is: Roger initially has 5 tennis balls. He then purchases two cans of tennis balls, with each can containing 3 tennis balls. How many tennis balls does he have now? The answer is: Roger starts with 5 tennis balls. Each of the two cans contains 3 tennis balls, totaling 6 tennis balls. Therefore, 5 plus 6 equals 11. The answer is 11. The second question is: A cafeteria has 23 apples. If they use 20 apples for lunch and

subsequently purchase an additional 6 apples, how many apples do they have now? The model output is: Answer: The cafeteria originally had 23 apples. They used 20 for lunch, leaving them with 23 minus 20 equals 3 apples. They then bought 6 more apples, so they now have 3 plus 6 equals 9 apples. The answer is 9.

The CoT for one-shot learning prompts is demonstrated by enriching sample examples with detailed reasoning processes, showing how to break down large problems effectively. The paper assesses the performance of standard and CoT-based prompts on three major language models—LaMDA, GPT, and PaLM. While GPT is developed by OpenAI, Google has released the other two. Testing revealed that Google’s PaLM model, boasting 540 billion parameters, reached approximately 60.1% accuracy on the elementary-level math reasoning dataset GSM8K, initially introduced by OpenAI in October 2021.

Following CoT’s introduction, the self-consistency technique, unveiled in March 2022, enhanced GSM8K’s accuracy to 74.4%. Illustrated in **Fig. 5.9**, self-consistency diverges from previous CoT methods by not just outputting a single task’s reasoning process and answer. Instead, it mandates the large language model to generate multiple reasoning processes and answers for each prompt, selecting the optimal solution through a voting mechanism. This approach integrates three advancements: extending from a single to multiple prompt questions, validating the



accuracy of intermediate reasoning steps, and applying weighted voting to the responses, boosting GSM8K's accuracy to approximately 83%.

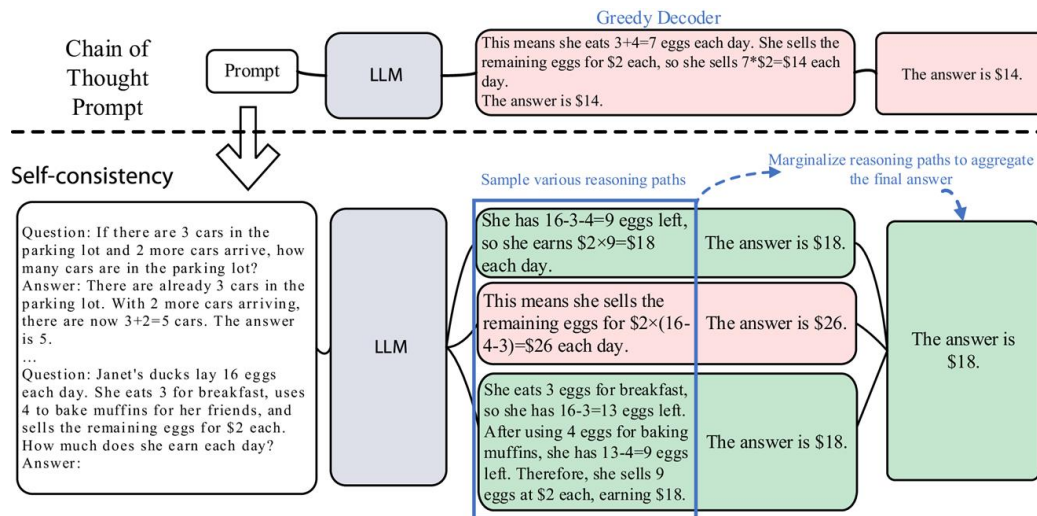


**FIGURE 5.9** Improving reasoning chains in language models with self-consistency.

It shows a prompt being processed by the LLM, leading to different reasoning paths. The example question is: Janet's ducks lay 16 eggs each day. She eats 3 for breakfast, uses 4 to bake muffins for her friends and sells the remaining eggs for dollar2 each. How much does she earn each day? The greedy decoder path concludes: This means she eats 3 plus 4 equals 7 eggs each day. She sells the remaining eggs for dollar2 each, so she sells 7 times dollar2 equals dollar14 each day. The answer is dollar14. The self-consistency approach samples various reasoning paths: 1. She has 16 minus 3 minus 4 equals 9 eggs left, so she earns dollar2 times 9 equals dollar18 each day. The answer is dollar18. 2. This means she sells the remaining eggs for dollar2 times (16 minus 4 minus 3) equals dollar26 each day. The answer is dollar26. 3. She eats 3 eggs for breakfast, so she has 16 minus 3 equals 13 eggs left. After using 4 eggs for baking muffins, she has 13 minus 4 equals 9 eggs left. Therefore,

she sells 9 eggs at dollar2 each, earning dollar18. The answer is dollar18. The final answer is aggregated as dollar18.

The CoT applications all reflect the divide-and-conquer approach, where complex reasoning challenges are segmented into simpler subtasks. By resolving these subtasks, the overall problem solution is derived from the individual answers. This strategy reaffirms the concept of “Bayesian inference of conceptual distribution,” using prompts to define task-specific conceptual distributions and enabling the model to more effectively infer the concepts learned during pretraining. With inherent reasoning abilities, large language models can enhance their step-by-step problem-solving accuracy by employing properly crafted prompts and methodical guidance.

For corpora in various languages, appropriate step-by-step prompts can differ significantly. For instance, in Chinese corpora, a phrase equivalent to “Let’s think step by step” could be better expressed as “The detailed problem-solving approach is as follows,” rather than a literal translation. Consequently, when employing different CoT approaches, it is crucial to tailor prompts based on the specific linguistic context. By doing so, prompts can effectively enhance the posterior conceptual distribution for task decomposition, enabling the large language model to dissect a complex reasoning issue into a series of simpler subtasks, thus improving its inferential prowess.

## 5.6 Summary

GPT-3 is an advanced model built on an optimized sparse Transformer decoder architecture, boasting up to 175 billion parameters. As the model has expanded, it has exhibited substantial improvements across a variety of NLP tasks; however, this enhancement comes at the cost of increased training expenses. To mitigate these costs, GPT-3 employs a content-based learning strategy, especially leveraging few-shot learning. This approach does not require adjusting model parameters but instead relies on acquiring task-specific knowledge from a limited number of examples, thus delivering strong performance on new tasks. This chapter delves into the workings of GPT-3's content-based learning through the lenses of Bayesian inference and conceptual distributions, and it discusses the divide-and-conquer philosophy underpinning the CoT approach. Lastly, **Section 5.5** investigates the CoT's capacity for coherent reasoning and innovation. By incorporating this advanced reasoning into its workflow, the model enhances its ability to comprehend and produce complex outputs.

---

## CHAPTER 6

# Pretraining strategies for large language models

---

### Abstract

This chapter provides an in-depth exploration of pretraining strategies for large language models (LLMs), focusing on the critical aspects of dataset selection, data processing, and distributed training techniques. It begins by discussing the importance of diverse, high-quality datasets and methods like deduplication and augmentation to enhance model performance and generalization. The chapter then delves into distributed training strategies, including data parallelism, model parallelism, and pipeline parallelism, which are essential for efficiently handling the massive computational demands of LLMs. Tools such as Pathways, Megatron-LM, and DeepSpeed Zero are highlighted for their roles in optimizing distributed training processes. The chapter concludes with a comparison of various training strategies employed by models like GPT-3, OPT, PaLM, and BLOOM, emphasizing the importance of stable parameters, optimized training settings, and hardware considerations such as BF16 precision. These insights are crucial for improving the efficiency and effectiveness of pretraining large-scale language models.

### Keywords

Large language models; pre-training strategies; dataset selection; distributed training; data parallelism; model parallelism; pipeline parallelism; Megatron-LM; DeepSpeed ZeRO; BF16 precision

In the initial stages of developing large language models (LLMs), key challenges such as the amount of data, the size of parameters, the design of the training framework, and how the training is managed play crucial roles. These issues might lead to problems like sudden spikes in loss or the model struggling to properly converge. To overcome these, integrating various training approaches, including handling of pretraining data, selection of training frameworks, and adjustments in training parameters, is essential to improve both the stability and efficiency of the pretraining phase.

From the release of GPT-3 onwards, OpenAI has implemented a partly proprietary approach, keeping certain essential technical details confidential. Details such as the methodology for creating the training dataset, the computational resources required, and specifics about hyperparameters like the learning rate, number of iterations, or the type of optimizer remain undisclosed. This chapter aims to delve deeper into the pretraining strategies of ChatGPT, drawing comparisons and insights from the disclosed information of other major natural language processing models.

## 6.1 Pre-training datasets

The ChatGPT model has been developed and refined based on GPT-3.5, whereas InstructGPT builds upon GPT-3.0. Both have been fine-tuned with the aid of human feedback to optimize performance. As outlined in [Table 6.1](#), during its pretraining, GPT-3.0 extensively used sources like Wikipedia, a variety of books, Internet-sourced web pages such as Common Crawl, and datasets like WebText2. GPT-3.5, on the other hand, expanded its sources to include code-

related datasets from GitHub, conversational forums such as StackExchange, and datasets of video subtitles.

1. **Wikipedia:** Serving as a free, multilingual online encyclopedia, Wikipedia brings together the knowledge and efforts of over 300,000 volunteers. By April 2022, the English Wikipedia featured more than 6.4 million articles, totaling over 4 billion words. Wikipedia's text is immensely valuable, noted for its meticulous content, precise references, clear descriptions, and extensive scope, which is why many research laboratories prefer to use its English version as their primary dataset.
2. **Books:** This category includes datasets from both fiction and nonfiction genres, aimed at enhancing the model's capabilities in storytelling and responsiveness. These datasets are primarily drawn from sources like Project Gutenberg and Smashwords (Toronto BookCorpus/BookCorpus). Notably, OpenAI's GPT-3 paper did not reveal the exact origins of the Books1 (12 billion tokens) and Books2 (55 billion tokens) datasets. Researchers speculate that these might come from collections similar to those on LibGen18 and Sci-Hub, though their vast sizes might not perfectly match the expected dataset characteristics of GPT-3.
3. **Scholarly Journals:** This dataset comprises articles from both preprints and journals that have been formally published, providing a robust and precise academic grounding. Academic texts are typically characterized by strong logical structuring, rationality, and detailed exposition. Key sources for these datasets include ArXiv and the National Institutes of Health.

4. WebText: WebText is a substantial dataset sourced from all outbound links on Reddit that have gathered at least three upvotes. This approach captures the trends of popular content online, which helps in curating quality links and generating related textual content.
5. Common Crawl: Operational since 2008, Common Crawl offers an extensive collection of web pages, metadata, and text extractions. It covers a diverse array of text in various languages and domains. Researchers often use the English-only, filtered version (C4) of this dataset for their projects.
6. Other Datasets: This includes datasets like GitHub for code, StackExchange for dialogues, and collections of video subtitles. These are vital in developing the model's proficiency in understanding code, managing dialogues, and processing multimodal information.

---

**Table 6.1**

**Dataset used for training GPT-3.**

| Dataset                 | Order of magnitude (tokens) | Weights in the training mix |
|-------------------------|-----------------------------|-----------------------------|
| Common Crawl (filtered) | 410B                        | 60%                         |
| WebText2                | 19B                         | 22%                         |
| Books1                  | 12B                         | 8%                          |
| Books2                  | 55B                         | 8%                          |
| Wikipedia               | 3B                          | 3%                          |

To summarize, the GPT-3.0 and GPT-3.5 models have harnessed an extensive variety of datasets during their training phases,



encompassing a broad range of fields, languages, and content types. This diversity in training resources has equipped the models with a more comprehensive understanding and improved efficiency, resulting in enhanced performance in real-world applications. The variety and depth of these datasets enable the models to effectively comprehend and produce a wide array of textual outputs, thereby increasing their usability and adaptability across various applications.

## 6.2 Processing of pretraining data

GPT-3 underwent its pretraining on a massive dataset of about 300 billion tokens, with around 60% derived from a meticulously selected portion of Common Crawl, and the rest from sources including WebText2, Books1 and Books2, and Wikipedia. The GPT-3.5 update further integrated code-related datasets such as GitHub Code. The way these datasets are sampled isn't strictly based on size but is optimized for quality. Although the open-source community has made several attempts to replicate ChatGPT, truly successful efforts are quite rare. Models like OPT-175B and BLOOM-176B, while comparable or even larger in parameter size than GPT-3, have not reached the performance level of the original GPT-3. Still, these models serve as valuable case studies for addressing the critical aspects of handling pretraining data discussed in this chapter.

1. Quality and quantity of pretraining data. The amount of data used for training is crucial for the effectiveness of pretrained models. For example, the volume of data used for PaLM and GPT-3's pretraining far exceeds that of OPT and BLOOM (refer to [Table 6.2](#)), indicating that large, high-quality datasets are pivotal to their success. Moreover, the quality of data significantly affects model performance. GPT-3, for

instance, utilizes an efficient classifier to sift through its pretraining dataset, a step not taken by models like OPT and BLOOM during their training.

2. De-duplication of Pre-training Data. Removing duplicates from the dataset is essential to prevent models from overly adapting to the same data, which improves their ability to generalize to new data. GPT-3 implements de-duplication at the document level to combat overfitting, while the Pile corpus used in OPT's pretraining still harbors a considerable amount of repeated data.
3. Diversity of Pre-training Datasets. For large-scale language models, it is vital to train on diverse datasets that cover different domains, formats (like text, code, and tables), and languages. OPT-175B incorporates sources such as Pushshift.io Reddit and Common Crawl, which are predominantly English. However, through a combination of various sources including Common Crawl, the corpus integrates a small portion of nonEnglish data. In contrast, BLOOM utilizes the ROOTS corpus, which encompasses data in 46 natural languages and 13 programming languages from hundreds of sources, providing it with a much richer diversity than OPT-175B.

**Table 6.2**

**Training dataset for large language models.**

|       | Number of tokens in the pre-training corpus | Number of tokens seen during pre-training |
|-------|---------------------------------------------|-------------------------------------------|
| PaLM  | 780B                                        | 770B                                      |
| GPT-3 | 500B                                        | 300B                                      |
| OPT   | 180B                                        | 300B                                      |
| BLOOM | 341B                                        | 366B                                      |

It is crucial to recognize that while having a diverse range of data is essential for training broadly capable LLMs, the specific make-up of the pretraining data has a profound effect on how well these models perform in certain tasks. For example, BLOOM, with its higher proportion of data in multiple languages, excels in tasks like multilingual processing and translation. OPT’s extensive use of conversational data, such as from Reddit, likely contributes to its effectiveness in dialogue-based tasks. GPT-3.5’s emphasis on coding datasets not only enhances its performance in coding-related tasks but may also improve its ability in chain-of-thought (CoT) reasoning. However, despite BLOOM’s inclusion of code data in pretraining, it struggles with coding and CoT tasks, indicating that code data alone isn’t enough to excel in these areas.

To push the boundaries of pretrained language models even further, researchers should consider the following:

1. Finer filtering of training data to ensure high quality.
2. More efficient de-duplication of training data to minimize redundancy and prevent overfitting.
3. Enhancing the diversity of training data to boost the models’ ability to generalize across various tasks and fields.

4. Investigating how data distribution affects model performance in specific applications to optimize the selection and organization of pretraining data.

Effective data de-duplication prevents overfitting, careful data selection ensures quality, and diverse datasets enhance the general applicability of large-scale pretrained language models. However, the detailed processes by which GPT-3 handles these data, or the specifics of the pretraining data itself, remain undisclosed.

## 6.3 Distributed training patterns

One distinct characteristic of LLMs is their ever-increasing size. Take GPT-3, for instance, with its staggering 175 billion parameters. Training this model requires a month, even with 1024 high-capacity (80 GB) A100 GPUs. The Meta LLaMA model, with 65 billion parameters, pushes the envelope further by using 2048 Nvidia A100 GPUs over a span of 21 days. With such growth in model size, significant challenges inevitably follow.

A primary hurdle is the limitations imposed by current hardware technology, notably the memory wall issue. Due to physical constraints, the computing capabilities and memory capacities of single devices are insufficient to support the expanding scale of large models, leading to challenges as follows:

1. VRAM limitations: No existing graphics processing unit (GPU) can fully store the parameters of these massive models. For instance, GPT-3 needs about 700 GB just for the model parameters, another 700 GB for gradients, and 1400 GB for optimizer states, adding up to roughly 2.8 TB.
2. Computational challenges: Even if it were possible to cram an entire model into one GPU (say, by shuttling parameters between the host and device memory), the sheer volume of computations required would make the training process

impractically long. For perspective, training GPT-3 on a single V100 NVIDIA GPU would take an estimated 288 years (**Fig. 6.1**).

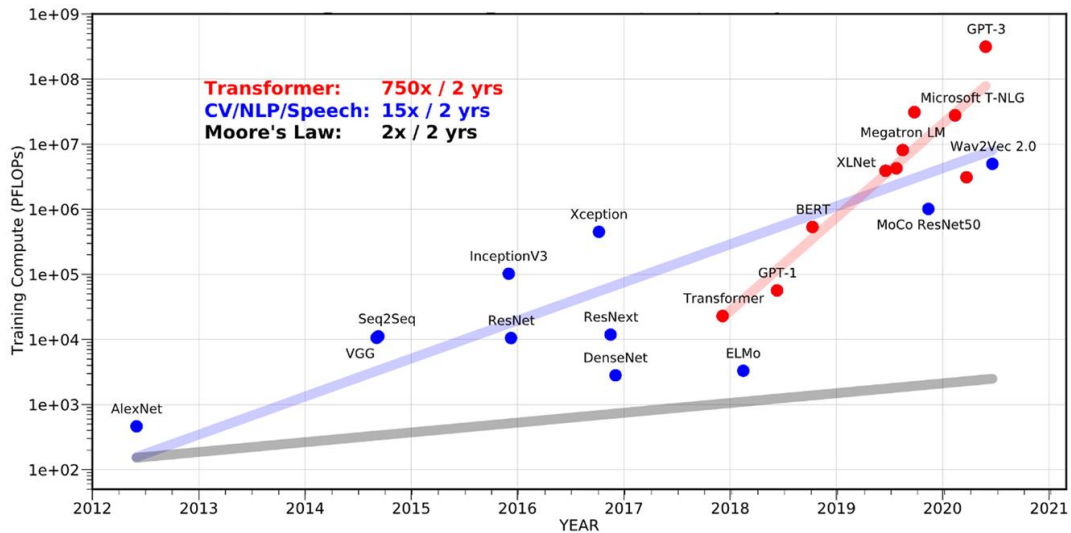


**FIGURE 6.1** Computational requirements for training state-of-the-art models in computer vision, natural language processing, and speech. Image source: <https://medium.com/riselab/ai-and-memory-wall-2cb4265cb0b8>.

The y-axis represents training compute in PFLOPS, ranging from 1eplus02 to 1eplus09 on a logarithmic scale. The x-axis shows years from 2012 to 2021. Three trend lines are shown: 1. Transformer: 750x / 2 yrs 2. CV / NLP / Speech: 15x / 2 yrs 3. Moore's Law: 2x / 2 yrs Various AI models are plotted: AlexNet (2012) VGG, Seq2Seq (2014) ResNet, InceptionV3 (2015) Xception (2016) ResNext, DenseNet (2017) ELMo, Transformer (2018) BERT, GPT-1 (2019) MoCo, ResNet50, XLNet, Megatron LM, Microsoft T-NLG (2020) GPT-3, Wav2Vec 2.0 (2021) GPT-3 is the highest point on the graph, showing significantly higher compute requirements than other models. The graph illustrates the rapid increase in compute requirements for AI models, particularly for transformer models, which show a 750x increase every 2 years, far outpacing Moore's Law (2x every 2 years).

As a result, despite the growing pains of advancing chip technology, distributed training across multiple nodes has emerged as the primary strategy for managing these gargantuan training tasks. Modern approaches include data parallelism, model parallelism, pipeline parallelism, and hybrid methods that combine data and model parallelism.

### **6.3.1 Data parallelism**

Data parallelism is a technique where the training data batch is split across multiple computational devices. Each device processes a portion of the data independently—performing tasks like forward and backward propagation and updating parameters. After each cycle of training, these devices synchronize their parameters to ensure the model remains consistent across all platforms. This method is particularly useful for processing large datasets quickly (**Fig. 6.2**).

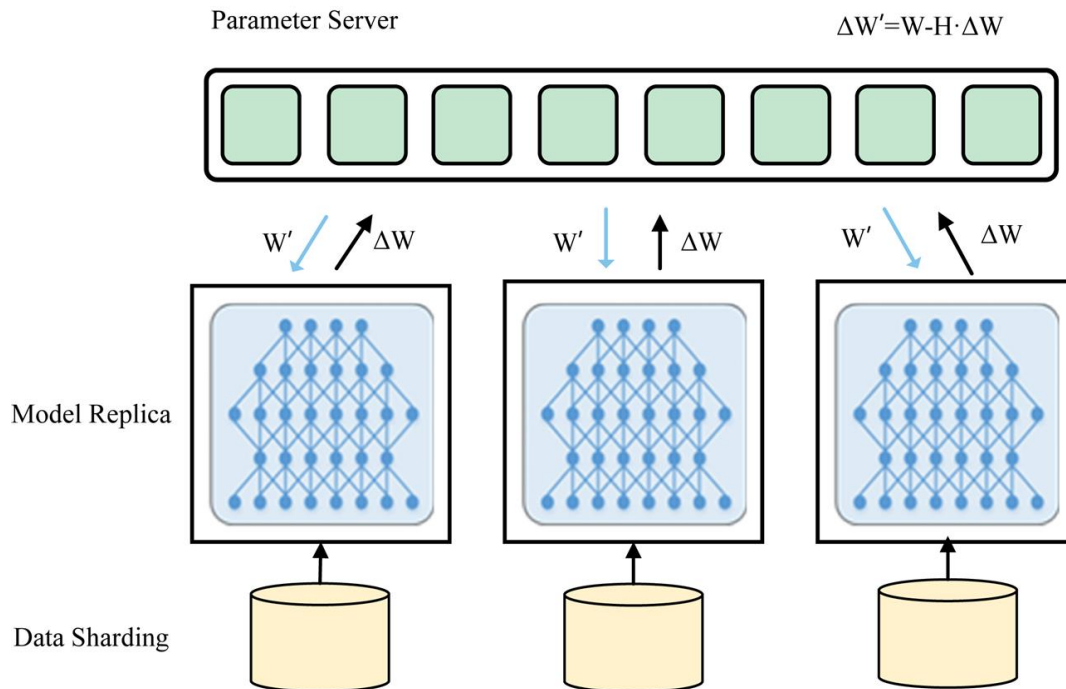


**FIGURE 6.2** Diagram of data parallelism.

Below the parameter server, there are three model replicas, each connected to a data sharding component. The parameter server is labeled with  $\Delta W' = W - H \cdot \Delta W$ . Each model replica is connected to the parameter server with arrows labeled  $W'$  and  $\Delta W$ . The data sharding components are depicted as cylindrical shapes below each model replica.

In practical terms, data parallelism is achieved by dividing the overall batch into smaller subbatches, each handled by a different device. A common framework for this is the parameter server model. Here, every device holds a copy of the model. Each node processes different subsets of data and calculates gradients. These gradients are then sent to a central node, or parameter server, which updates the model and redistributes the updated version back to each node. Despite its efficacy in scaling, data parallelism faces certain challenges:



1. If the batch size per GPU drops below a certain point, it can lead to underutilization of GPU capacities and higher communication overhead.
2. There's a limit to how many devices can be effectively used, dictated by the manageable batch size, which can cap the number of accelerators employed in training.

### 6.3.2 Model parallelism

Model parallelism is designed to bypass the memory restrictions of individual computing units, like GPUs, by distributing a massive model across several devices during training. In this setup, each component of the model is handled by different devices, and every device is involved in all steps of the training process, as illustrated in [Fig. 6.3](#). Model parallelism includes two primary types: tensor parallelism and pipeline parallelism.

1. Tensor parallelism: This approach splits model parameters—such as weights, gradients, and optimizer states—into smaller subtensors, which are then allocated to various computing devices. Each device handles its portion of the computation independently and synchronizes with others to update the entire model. This method effectively reduces the burden on individual device memory, facilitating the training of larger models within memory constraints.

Tensor parallelism specifically involves dividing the model's parameters and distributing them across different devices. This process uses different methods of sharding, with one-dimensional sharding being the most common, where a tensor is split along a single dimension, either horizontally or vertically. Future techniques may include more complex, multidimensional sharding methods. For example, consider matrix multiplication in the format  $XA = Y$ , where  $X$  is the

input,  $A$  the weights, and  $Y$  the output. The process involves breaking down the matrix into segments for calculation and then aggregating the results. The matrix  $A$  could be divided by rows or columns:

$$\begin{aligned} Y &= X[A_1, A_2] = [XA_1, XA_2] \\ Y &= [X_1, X_2] \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} = X_1A_1 + X_2A_2 \end{aligned} \tag{6.1}$$

This setup allows, for example, two different GPUs to handle separate segments of a matrix multiplication task and then communicate to combine the final output. This method can be expanded to include more GPUs and can be applied to other operations that support sharding.

2. Pipeline parallelism. This advanced approach divides a model into several sequential stages, each containing one or more layers, with each stage executed on different computational devices. Similar to a production line, training data flows seamlessly from one stage to the next. Each device processes its assigned stage and then passes the outcome forward, boosting both the efficiency and speed of training (**Fig. 6.4**).

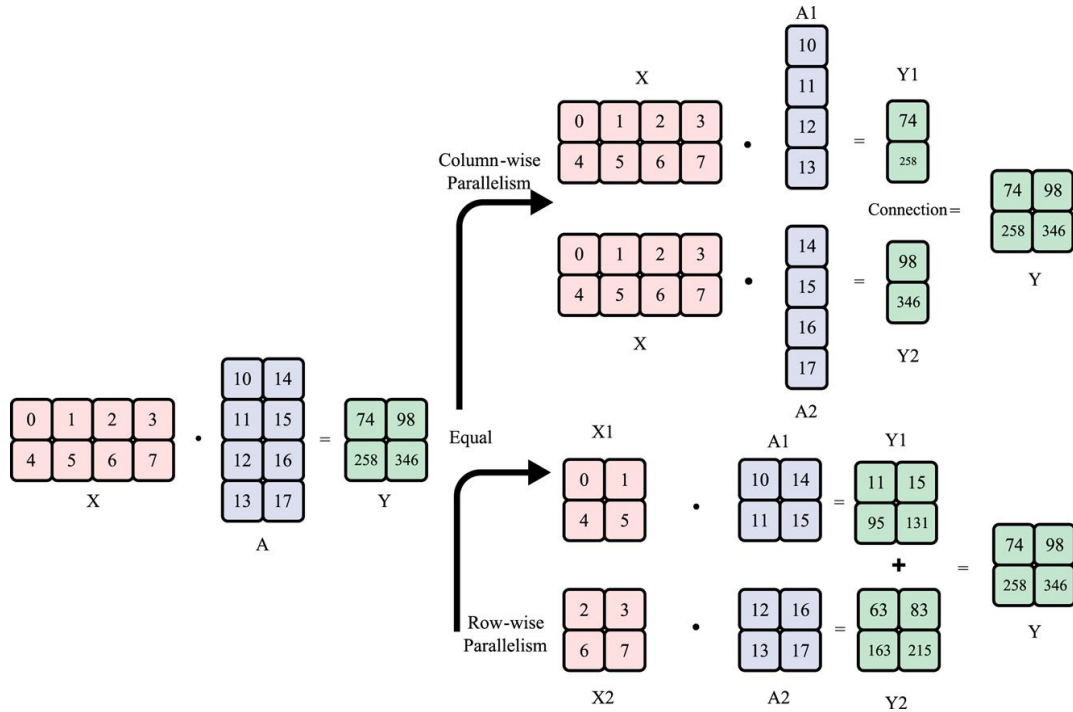


**FIGURE 6.4** The division of matrix multiplication across multiple GPUs.

The initial matrices are X and A. Matrix X is a 2 by 4 matrix with elements 0, 1, 2, 3 in the first row and 4, 5, 6, 7 in the second row. Matrix A is a 4 by 2 matrix with elements 10, 14 in the first row, 11, 15 in the second row, 12, 16 in the third row and 13, 17 in the fourth row. The resulting matrix Y is a 2 by 2 matrix with elements 74, 98 in the first row and 258, 346 in the second row. Column-wise parallelism is shown with matrix X multiplied by sub-matrices A1 and A2. Matrix A1 is a 4 by 1 matrix with elements 10, 11, 12, 13. Matrix A2 is a 4 by 1 matrix with elements 14, 15, 16, 17. The resulting matrices Y1 and Y2 are 2 by 1 matrices with elements 74, 258 and 98, 346 respectively. These are combined to form matrix Y. Row-wise parallelism is shown with matrix X divided into sub-matrices X1 and X2. Matrix X1 is a 2 by 2 matrix with elements 0, 1 in the first row and 4, 5 in the second row. Matrix X2 is a 2 by 2 matrix with elements 2, 3 in the first row and 6, 7 in the second row. Matrix X1 is multiplied by matrix A1 to form matrix Y1, a 2 by

1 matrix with elements 11, 15 in the first row and 95, 131 in the second row. Matrix X2 is multiplied by matrix A2 to form matrix Y2, a 2 by 1 matrix with elements 63, 83 in the first row and 163, 215 in the second row. The resulting matrices Y1 and Y2 are combined to form matrix Y.

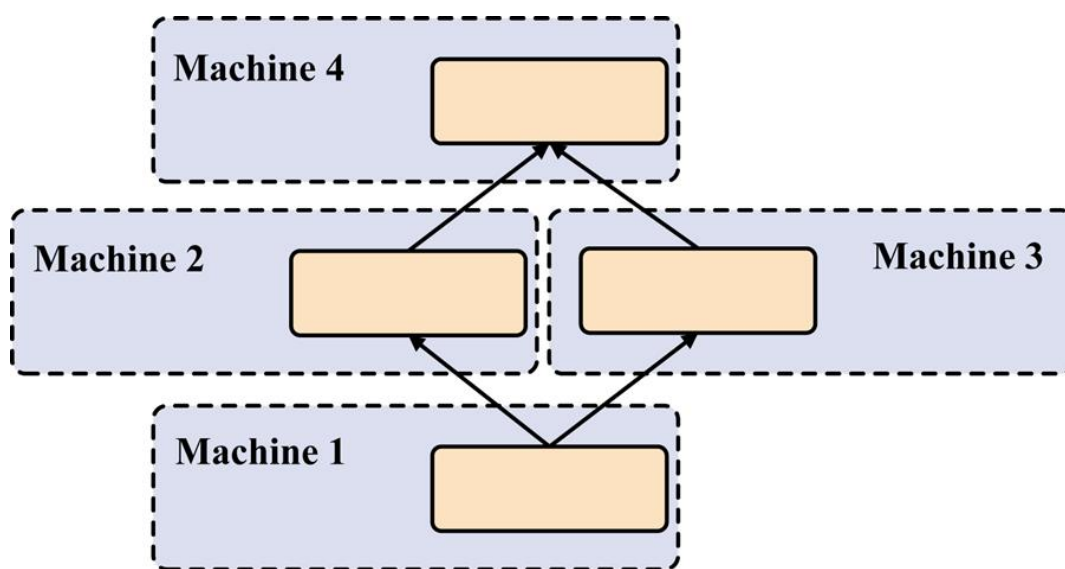


**FIGURE 6.3** Diagram of model parallelism.

Neural networks can benefit from tailored parallel strategies based on their structure. For example, components with numerous parameters might best be managed by splitting the model among devices, whereas segments with fewer parameters could be better served by splitting the data. Mixing parallel strategies in specific network parts can minimize data transfer. In parts of the network with extensive hidden layers, both data and model matrices might be split simultaneously. Unlike data and model parallelism, where devices perform parallel computations, pipeline parallelism organizes tasks into a clear sequence, delegating different stages to various devices. This allows each device to focus on a specific

segment of the network, supporting the training of larger models or handling larger batches more effectively.

In practice, different network layers are distributed across various GPUs, each sequentially processing its designated stage with fresh data inputs at every step, optimizing throughput. As depicted in **Fig. 6.5**, each computing unit is responsible for a specific layer or set of layers, executing in an orderly fashion. For instance, when the initial data input is received, device 0 calculates the first layer,  $F_0$  by the next step, device 0 processes another dataset's first layer, while device 1 moves on to the second layer,  $F_1$ , of the initial dataset. Though pipeline and tensor parallelism typically exist alongside each other and complement one another, pipeline setups can lead to underutilization, as downstream units must wait for their predecessors to finish. Additionally, this method can lead to “pipeline bubbles” where devices may idle between completing forward propagation and starting backward propagation, as illustrated in **Fig. 6.5**.

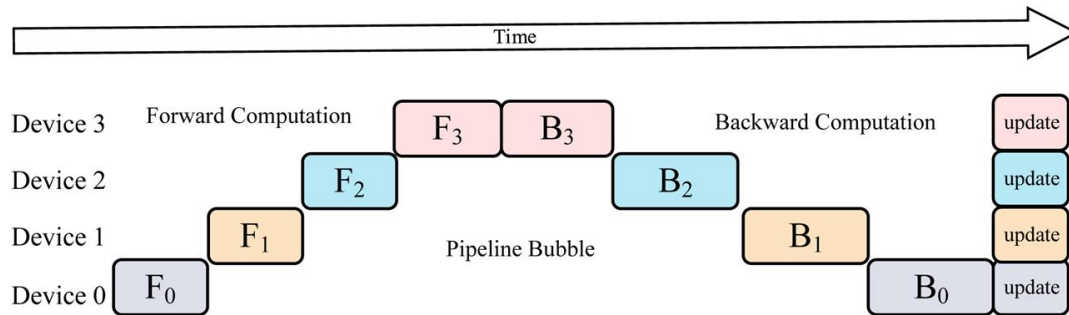


**FIGURE 6.5** Diagram of pipeline parallel training.

The diagram illustrates the computation process across four devices labeled Device 0, Device 1, Device 2 and Device 3 over time. The time axis is represented by a horizontal arrow pointing to the right. The computation is divided into forward computation and backward computation stages. Forward computation includes  $F_0$  through  $F_3$  on Device 0, Device 1, Device 2 and Device 3. Backward computation includes  $B_0$  through  $B_3$  on Device 0, Device 1, Device 2 and Device 3. There is a pipeline bubble between the forward and backward computation stages. After backward computation, each device undergoes an update stage labeled as update.

**Table 6.3** provides a comparative analysis of three parallelism techniques—data parallelism, tensor parallelism, and pipeline parallelism—focusing on aspects like generality, computational utilization, VRAM consumption, and communication overhead. Data parallelism stands out for its versatility and high efficiency in both computation and communication, although it comes with a considerable VRAM overhead. Tensor parallelism, on the other hand, is more VRAM-efficient but incurs additional communication costs and is less flexible. Pipeline parallelism is efficient in terms of VRAM usage and has lower communication overheads but may

suffer from “pipeline bubbles,” which can disrupt the smooth processing flow.

**Table 6.3****Comparison of three types of parallelism.**

|                         | <b>Data<br/>parallelism</b>                                                                     | <b>Tensor<br/>parallelism</b>                                                                           | <b>Pipeline<br/>parallelism</b>                                           |
|-------------------------|-------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| Generality              | Fully general                                                                                   | It can only be used on some models; for example, tensor parallelism cannot currently be applied to CNNs | Generally applicable                                                      |
| Computation utilization | When data distribution is relatively even, the utilization loss of computational devices is low | Frequent transmission of intermediate results incurs some utilization loss of computational devices     | Pipeline bubbles result in some utilization loss of computational devices |
| Memory overhead         | Each card retains the same model and optimizer                                                  | It can evenly distribute memory overhead across                                                         | If the parameter sizes of different layers of the model vary              |



|                      | <b>Data parallelism</b>                                                                | <b>Tensor parallelism</b>                                                               | <b>Pipeline parallelism</b>                                                                                                                   |
|----------------------|----------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
|                      | parameters, resulting in relatively high overhead                                      | different servers                                                                       | significantly, adjustments are required to achieve optimal results                                                                            |
| Communication volume | Only the gradients on different GPUs need to be transmitted, resulting in low overhead | Frequent transmission of intermediate results incurs significant communication overhead | Only the hidden states and backpropagation gradients need to be transmitted between different layers, resulting in low communication overhead |

## 6.4 Technical approaches to distributed training

Distributed training often outperforms training on a single machine with a single GPU because it enables training on a much larger scale. However, each method of distributed training comes with its own set of pros and cons. Simply adding more computing devices doesn't always lead to increased processing power, due to the slower growth rate of memory bandwidth compared to processing power, and even slower growth in interdevice network bandwidth.

Limited by the so-called “memory wall” and “network wall,” the amount of data transferred becomes a critical factor in determining the speed and effectiveness of distributed training.

When comparing data parallelism with model parallelism, it is clear that if a batch of data results in many intermediate outputs but has relatively few parameters, data parallelism is likely the better option. On the other hand, if processing a batch involves managing a large number of parameters with fewer intermediate outputs, model parallelism may prove more beneficial. It is important to note that data parallelism has a distinct advantage because it allows communication tasks to overlap more seamlessly with computations, effectively hiding some of the communication delay. This is more challenging to achieve with model parallelism.

For training massive models, pipeline parallelism is advantageous due to its minimal data transfer requirements, which involve only the sum of data transferred between stages. This is in contrast to data and tensor parallelism, which involve data transfers across the entire computation graph. Therefore, pipeline parallelism may be preferred for devices with limited bandwidth. However, a hybrid approach combining pipeline and model parallelism might offer superior performance compared to using just one of these methods. Additionally, there is potential for optimization in both data and model parallelism, as the time spent computing can obscure the time needed for data transfers.

As LLMs gain prominence, the focus on developing and implementing parallel processing strategies like model parallelism and pipeline parallelism for the foundational Transformer model has become a critical task in both academic and industrial circles. Examples include the GPipe neural network training library built on Lingvo, Microsoft’s PipeDream, NVIDIA’s Megatron-LM, Facebook’s FairScale, Microsoft’s DeepSpeed ZeRO, and Google’s Pathways.

There are primarily two technological paths for training massive language models. First, the approach led by Google utilizes a combination of tensor processing units (TPUs), accelerated linear algebra (XLA), and TensorFlow/JAX, deeply integrated with Google's cloud infrastructure. The second path, supported by industry giants like NVIDIA, Meta, and Microsoft, relies on GPUs combined with PyTorch, Megatron-LM, and DeepSpeed. This latter path represents the most developed open-source strategy and facilitates large-scale model training through parallelism. Both methods are tailored for training substantial models across numerous GPUs and nodes, adeptly overcoming the challenges of memory and computation that typically accompany LLM training, enabling the scaling of models despite hardware limitations.

### **6.4.1 Pathways**

Pathways is an innovative asynchronous distributed data flow architecture developed by Google, tailored for machine learning (ML) applications. Illustrated in **Fig. 6.6**, the architecture organizes the model's computation into a directed acyclic graph (DAG), where nodes signify computation tasks and edges denote data dependencies. Pathways aims to enhance distributed training efficiency, showing particular strength when handling expansive models and datasets. The architecture boasts four main features:

1. **Flexibility.** Pathways supports various forms of parallel computing, including data parallelism, model parallelism, and pipeline parallelism, making it adaptable to different ML challenges and hardware setups.
2. **Asynchronous execution.** Pathways enables tasks to start processing as soon as their input data dependencies are satisfied, without the need for the entire batch to be ready.

This approach minimizes idle time for computing resources, boosting training efficiency.

3. Distributed processing. The architecture allows tasks to be spread across multiple devices, maximizing the use of available hardware and enhancing both the speed and scalability of training.
4. Fault tolerance. Pathways is designed with fault-tolerance capabilities, ensuring continuous operation even in the face of device failures. This feature is crucial for long-term training tasks within large-scale distributed systems, where such failures are more common.

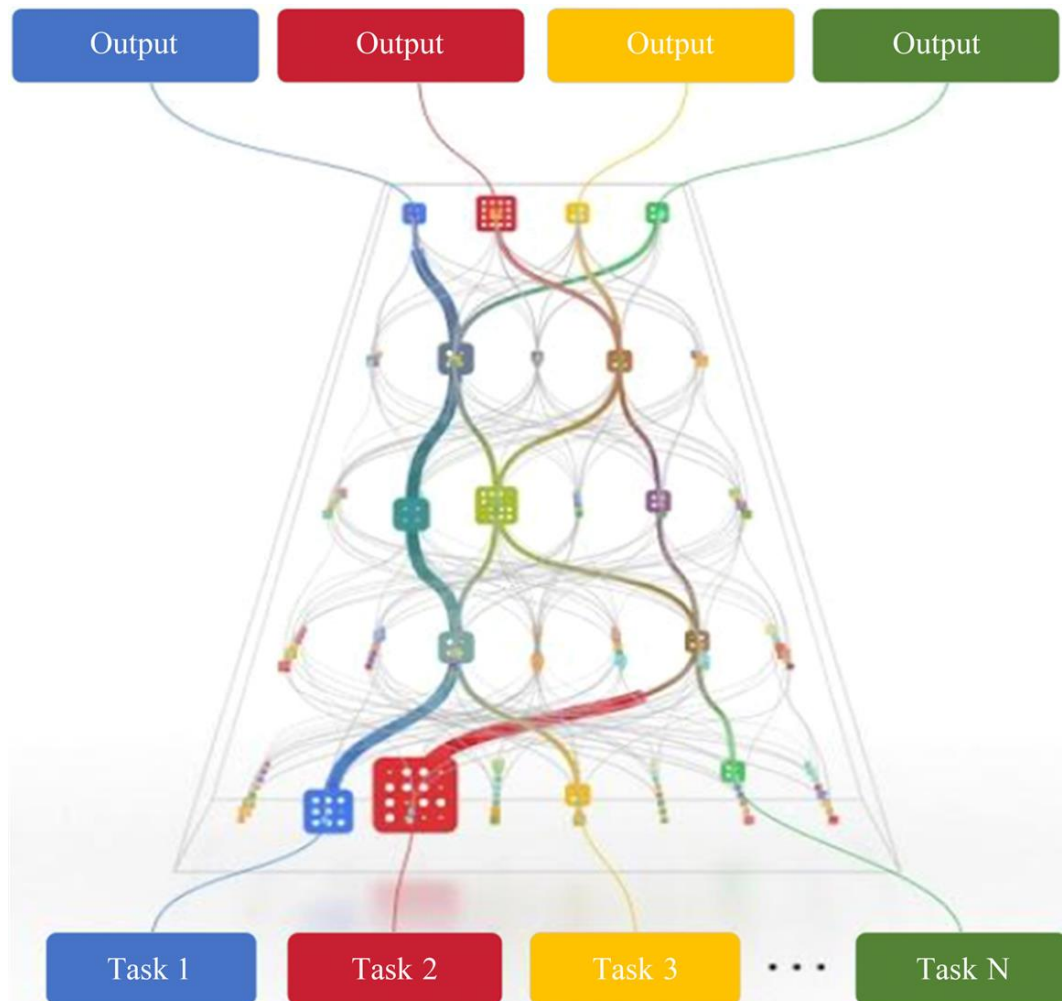


**FIGURE 6.6** Google pathways asynchronous distributed data flow. Image source: Paper “Introducing Pathways: A next-generation AI architecture”.

The diagram shows a network of interconnected nodes representing the flow from tasks to outputs. At the bottom, there are labels for Task 1, Task 2, Task 3 and Task N. These tasks are connected through a series of nodes and pathways to the outputs at the top, labeled as Output, Output, Output and Output.

**Fig. 6.7** provides a comprehensive view of the Pathways architecture, encompassing three primary components:

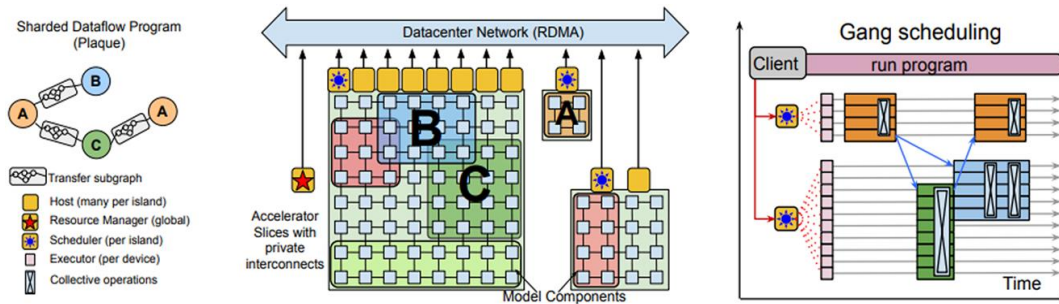


**FIGURE 6.7** Overview of the pathways system.

Image source: Paper “Pathways: Asynchronous Distributed Dataflow for ML”.

The image consists of three primary components of the Pathways architecture. The first component, labeled Sharded Dataflow Program (Plaque), shows a diagram with nodes A, B and C connected by transfer subgraphs. It includes symbols for Host (many per island), Resource Manager (global), Scheduler (per island), Executor (per device) and Collective operations. The second component, labeled Datacenter Network (RDMA), depicts a network with accelerator slices with private interconnects and model components labeled A, B and C. Hosts, resource managers, schedulers and executors are represented within the network. The third component, labeled Gang scheduling, shows a client running a program over time with various tasks scheduled and executed in parallel. The diagram includes arrows indicating the flow of tasks and operations.

Distributed computing represented as DAG (left diagram): This diagram depicts each node as an individual compiled function, with edges between nodes illustrating the flow of data between these functions. This setup forms the core of the Pathways system, where all computations are structured as a DAG.

Resource manager distributes island accelerator subsets (middle diagram): The resource manager assigns subsets of island

accelerators, termed “virtual slices,” to each compiled function. This allocation method ensures that each function operates on designated subsets of accelerators, significantly improving the efficiency of computations.

Centralized scheduling by each island's scheduler (right diagram): Each island features a central scheduler that orchestrates gang scheduling for all computations within the island. These computations are subsequently assigned to the executors on each slice for execution. Arrows emanating from the client module in the diagram denote control messages, whereas the other arrows depict the transmission of data paths.

The Pathways architecture enables components of a model to be processed in parallel across multiple devices, facilitating model parallelism. Within Pathways’ DAG framework, each node symbolizes a segment of the model, allowing the entire computation to occur concurrently on various devices, thereby accelerating model training. Additionally, Pathways supports data parallelism by processing different data subsets simultaneously across devices. In this configuration, each node processes a data subset and integrates the results into a unified model. Pathways is versatile in its parallel strategies and can fine-tune the level of parallelism as needed, offering the flexibility to employ model parallelism, data parallelism, or their combinations based on specific requirements.

Pathways aims to leverage existing hardware assets like GPUs and TPUs to boost ML tasks’ performance. While it doesn’t require specialized hardware, its effectiveness is tightly linked to the distributed computing power of the underlying hardware. Pathways can assign tasks across multiple devices and capitalize on interdevice communications for distributed computing, making it suitable for a variety of hardware setups, including clusters of multiple GPUs, TPUs, or a mix of both.

However, to fully harness the capabilities of the Pathways architecture, it is crucial to use hardware with robust communication facilities. High-speed networking options such as InfiniBand or high-speed Ethernet can dramatically enhance data transfer rates between devices, improving distributed computing performance.

Pathways is designed to operate on existing hardware, but for best results, choosing hardware with superior communication capabilities is essential. It is noteworthy that Google develops the specific implementation of Pathways as an internal project, and the details and source code are not entirely public.

### **6.4.2 Megatron-LM**

Megatron-LM is a deep learning tool developed by NVIDIA and is available as an open-source project on GitHub, along with its documentation. It is compatible with other deep learning frameworks, such as PyTorch and TensorFlow, enabling the training of large-scale models across different hardware and software setups. The tool is designed to train massive models efficiently on multiple GPUs and nodes by utilizing model parallelism strategies, primarily tensor and pipeline parallelism.

By integrating tensor and pipeline parallelism, Megatron-LM optimizes the training process for extensive Transformer models. It achieves this by distributing the weights and layers of the model across several GPUs, allowing the training of larger, more complex models even though it necessitates transferring data between GPUs and synchronizing gradients. Therefore, having an effective communication strategy is essential to reduce communication overhead significantly.

Megatron-LM doesn't establish a new parallel modeling framework or compiler but instead enhances the existing implementation of PyTorch Transformers with precise



modifications. This straightforward method avoids the need for new compilers or extensive code rewriting, using simple primitives to implement model parallelism. Specifically, Megatron-LM divides operations within the Transformer's masked multihead attention and feedforward sections, facilitating their parallel processing.

**Fig. 6.8** illustrates this process using masked multihead attention as an example. Megatron-LM leverages the inherent parallelism of multihead attention operations, dividing the associated small matrix multiplications (GEMM) for keys (K), queries (Q), and values (V) in a column-parallel fashion. This division allows each GPU to handle the matrix multiplications for corresponding attention heads locally, splitting the parameters and workload across GPUs, with each one producing parts of the final output.

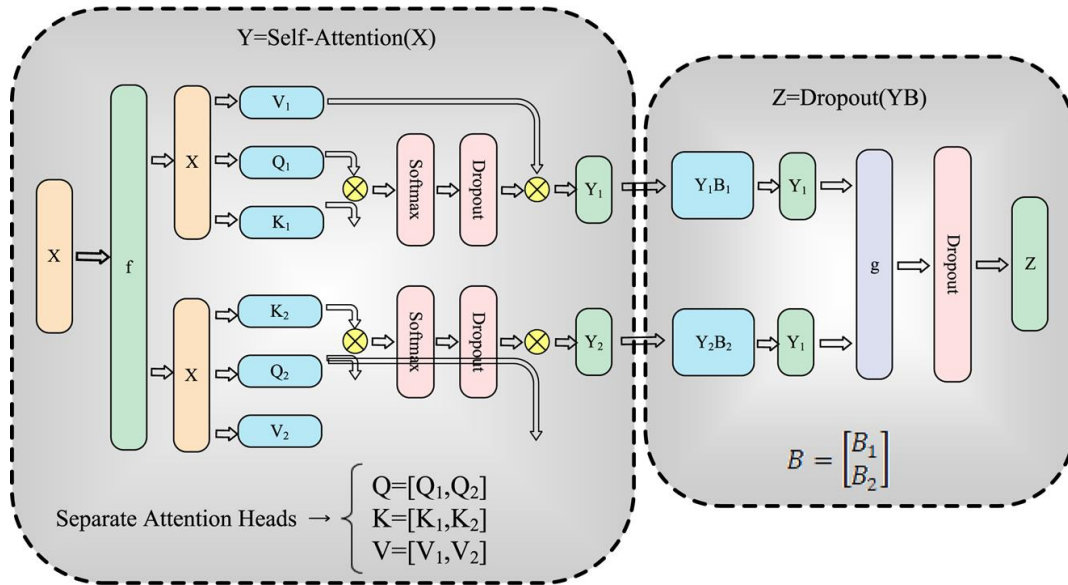


**FIGURE 6.8** Transformer block with model parallelism.

On the left, the self-attention mechanism is shown with input  $X$  passing through a function  $f$ , resulting in separate attention heads. Each head has components  $Q_1, K_1, V_1$  and  $Q_2, K_2, V_2$ . The heads are combined as  $Q = [Q_1, Q_2]$ ,  $K = [K_1, K_2]$  and  $V = [V_1, V_2]$ . The outputs  $Y_1$  and  $Y_2$  are obtained through softmax and dropout operations. On the right, the dropout process is shown with  $Y = \text{Dropout}(YB)$ . The outputs  $Y_1B_1$  and  $Y_2B_2$  are combined as  $B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$ . The final output  $Z$  is obtained after another dropout operation.

In the subsequent fully connected layers of the model, each GPU handles a portion of the output, allowing the weight matrix  $B$  to be

row partitioned for direct computations with inputs  $Y_1$  and  $Y_2$ . The final output,  $Z$ , is then derived through an all-reduce operation combined with Dropout. In real-world setups, it is unnecessary to dedicate a GPU to each attention head; instead, multiple heads can efficiently share a single GPU without breaching the principle of independent operations on individual GPUs. Hence, the total number of attention heads is ideally set to be divisible by the number of GPUs available.

Megatron-LM employs a comparable approach to split operations in the feedforward section, with decisions on which operators to divide being based on workload characteristics such as the amount of computation, parameters, and required communications. In the process of both forward and backward propagation within a single model parallel Transformer layer, Megatron-LM involves four instances of communication. Notably, the subsequent GEMMs of the linear layer outputs post-self-attention are parallelized along rows and seamlessly gather outputs from the parallel attention layer, circumventing the need for inter-GPU communication. This integration strategy, applied to both multi-layer perceptron (MLP) and self-attention layers, merges two sets of GEMM operations and removes intermediate synchronization points, thus boosting the model's scalability. This configuration enables all GEMM operations within a straightforward Transformer layer, utilizing two all-reduce operations each in both the forward and reverse paths (as depicted in **Fig. 6.9**).

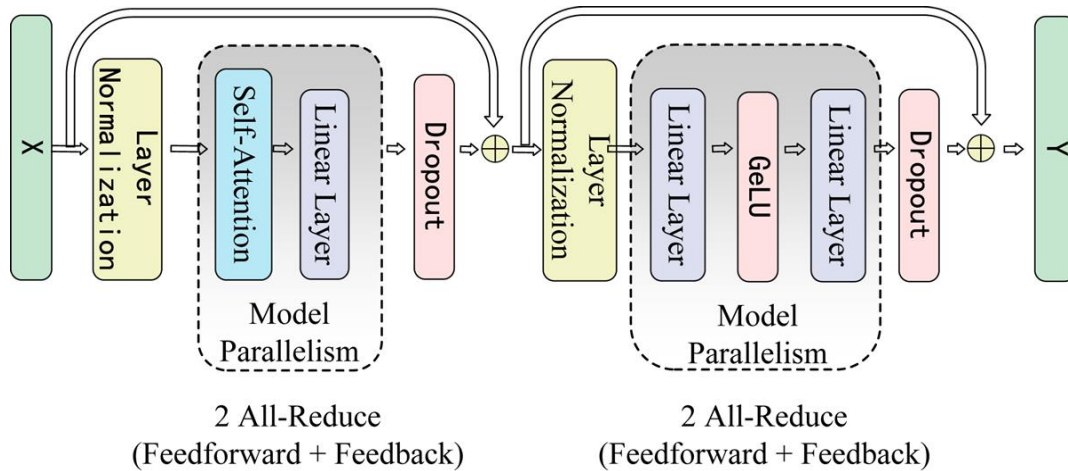


**FIGURE 6.9** Communication operations in transformer layers within the Megatron architecture.

The diagram illustrates a neural network model with two stages of model parallelism. The first stage starts with input  $X$ , followed by layer normalization, self-attention, a linear layer and dropout. The second stage includes layer normalization, a linear layer, GeLU, another linear layer and dropout, ending with output  $Y$ . Both stages involve 2 All-Reduce operations for feedforward and feedback.

Megatron-LM's strategy for splitting operations focuses on minimizing communication overhead and maintaining manageable computation loads across GPUs. It avoids isolating operations such as dropout, layer normalization, or residual connections to a single GPU with results then broadcasted across the network. Instead, it promotes simultaneous replicated computations across multiple GPUs.

In its architectural design, Megatron-LM treats model parallelism and data parallelism as independent dimensions. Illustrated in [Fig. 6.10](#), the architecture integrates pipeline, tensor, and data parallelism, making the majority of operations compute-bound rather than limited by memory constraints. Megatron-LM adeptly

divides the computation graph across devices, minimizing both the data transmitted over networks and the idle time of devices. This method, termed pipeline, tensor, and data parallelism (PTD-P), leverages pipeline parallelism across several GPU servers and both tensor and data parallelism within these servers. It operates within a well-optimized cluster featuring high-bandwidth connections for both intra- and interserver communications. This setup enables the training of models with up to a trillion parameters on 1000 GPUs, showcasing robust scalability and efficient computational performance, achieving up to 52% of the peak throughput of the devices.

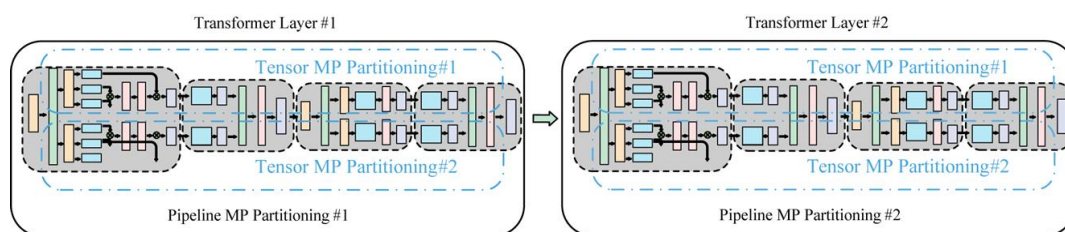


**FIGURE 6.10** Combination of tensor and pipeline model parallelism.

The diagram illustrates two transformer layers labeled as Transformer Layer number 1 and Transformer Layer number 2. Each transformer layer is enclosed within a dashed line and further divided into two sections labeled as Tensor MP Part itioning number 1 and Tensor MP Part itioning number 2. The entire transformer layer is also labeled as Pipeline MP Part itioning number 1 for the first layer and Pipeline MP Part itioning number 2 for the second layer. The diagram shows the internal structure and connections within each partitioning.

### 6.4.3 ZeRO

DeepSpeed features the Zero Redundancy Optimizer (ZeRO) as its primary component, a strategy of data parallelism specifically aimed at decreasing both the memory demands and the communication load during model training. ZeRO's primary objective is to reduce the amount of redundant data, such as model parameters, optimizer states, and gradients. It achieves this by distributing these elements among various training processes, facilitating more effective training of larger models within constrained hardware setups.

For training language models, the combination of mixed precision training and the Adam optimizer has become the norm. The Adam optimizer dynamically adjusts learning rates based on the first and second moments of the gradients for each parameter. According to **Fig. 6.11**, during the training phase, ZeRO divides storage into model states and residual states. The model state encompasses model parameters (fp16), model gradients (fp16), and the Adam states, which include fp32 backups of model parameters, and fp32 first and second moments. If the total amount of model parameters is  $\Phi$ , then the storage requirement is  $2\Phi + 2\Phi + (4\Phi + 4\Phi + 4\Phi) = 4\Phi + 12\Phi = 16\Phi$  bytes, with Adam states making up 75% of this total. The residual state comprises activation values, a variety of temporary buffers, and fragments of VRAM that cannot be utilized effectively.

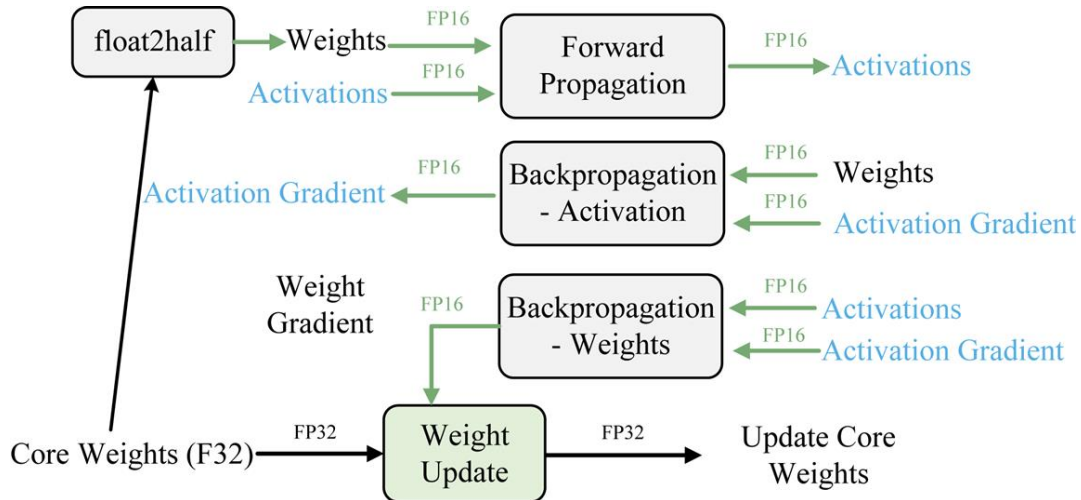


**FIGURE 6.11** Mixed precision training iteration.

The diagram illustrates a neural network training process. It starts with Core Weights (F32) which are converted to float2half. The Weights in FP16 are used in Forward Propagation to produce Activations in FP16. These Activations are then used in Backpropagation - Activation to generate Activation Gradient in FP16. The Weights and Activation Gradient are used in Backpropagation - Weights to produce Weight Gradient in FP16. The Weight Gradient is then used in Weight Update along with Core Weights (F32) to Update Core Weights in FP32.

ZeRO reduces memory redundancy in data parallel processes by distributing model states—parameters, gradients, and optimizer states—across processes rather than duplicating them. It employs a dynamic communication strategy during training to distribute essential states across devices effectively, which helps balance the demands of computational granularity with the communication needs of data parallelism. With ZeRO, each device's memory usage increases linearly with the level of data parallelism, and the communication volume mirrors this scalability. ZeRO can handle any size model, provided the devices in the cluster have sufficient memory to share the model state.

**Fig. 6.12** depicts ZeRO’s three-tiered system, where each level represents a progressively greater degree of model state partitioning:

- ZeRO-1: This level partitions the optimizer state.
- ZeRO-2: This level extends partitioning to both the optimizer state and model gradients.
- ZeRO-3: This most extensive level partitions the optimizer state, model gradients, and model parameters.

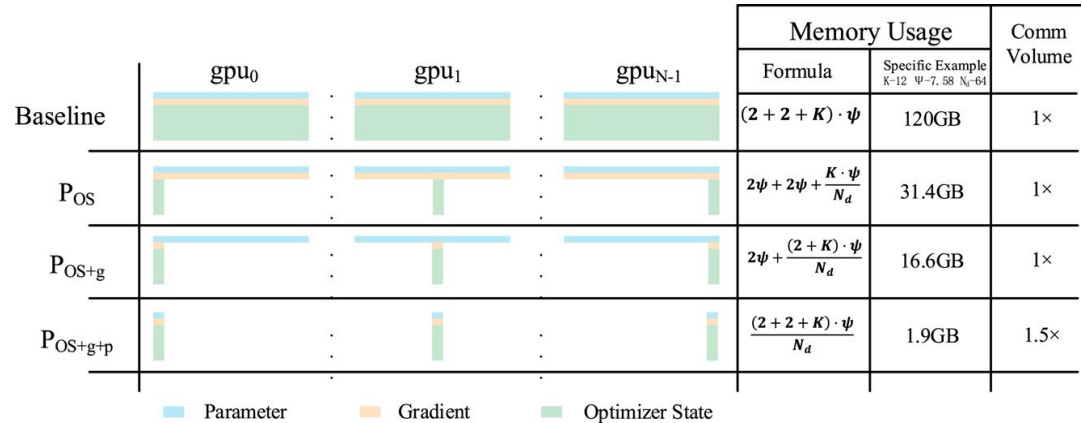


**FIGURE 6.12** Memory savings and communication volumes at the three levels of ZeRO.

The table consists of five columns: unnamed column, gpu0, gpu1, gpuN-1, Memory Usage and Comm Volume. It reads as follows: Row 1. Baseline: (2 plus 2 plus K) times psi, 120GB, 1 times. Row 2. POS: 2 times psi plus 2 times gamma plus K times psi divided by Nd, 31.4GB, 1 times. Row 3. POSplusg: 2 times psi plus (2 plus K) times psi divided by Nd, 16.6GB, 1 times. Row 4. POSplusgplusp: (2 plus 2 plus K) times psi divided by Nd, 1.9GB, 1.5 times.

The ZeRO paper not only quantifies the potential memory savings at each level but also notes that the first two levels do not increase



the communication overhead, whereas the third level boosts communication costs by about 50%. Additionally, ZeRO optimizes residual states, which include activation values, temporary buffers, and VRAM fragments, though the specific methods for this optimization are not detailed here.

ZeRO uses a combination of partitioning and checkpointing techniques for handling activation values. In the forward phase of model propagation, checkpointing preserves certain intermediate variables to assist in gradient calculations during the backward phase. This approach essentially opts for increased computational tasks in lieu of higher storage requirements. For instance, in matrix multiplication  $C=A \cdot B$ , the gradients for  $A$  and  $B$  are calculated as follows:

$$\begin{aligned}\nabla A &= \nabla C \cdot B^T \\ \nabla B &= A^T \cdot \nabla C.\end{aligned}\tag{6.2}$$

Storing variables  $A$  and  $B$  for backward propagation occupies a considerable amount of memory. The key to checkpointing is maintaining only the gradients at specific checkpoints, with gradients between these points recalculated during the backward pass by re-executing forward operations. This method effectively replaces storage demands with additional computations.

Regarding temporary buffers, the model training process necessitates the creation of buffers of various sizes, such as those used for the All-Reduce operation on gradients. A practical solution is to establish a fixed-size buffer at the outset, which prevents the need for dynamic buffer creation during training. Additionally, for smaller data transfers, efficiently grouping multiple data sets into buckets for simultaneous transmission can improve performance.

VRAM fragmentation often arises from the frequent creation and deletion of gradients that are not stored, a consequence of employing gradient checkpointing. A practical approach to mitigate

this issue is to allocate a continuous segment of VRAM in advance. This segment is used to store the model states that need to remain in memory consistently, along with the values from checkpoints, while the rest of the VRAM is available for the dynamic handling of obsolete activation values.

ZeRO-Offload is a technique that shifts part of the computational workload and model parameters from the GPU's VRAM to the CPU's memory. This involvement of the CPU in processing tasks substantially reduces the demand on GPU VRAM, making it feasible to train larger models with limited hardware.

To counteract the potential communication overhead between the GPU and CPU and the generally lower computational speed of the CPU, ZeRO-Offload strategically allocates computation tasks. By analyzing the performance of the Adam optimizer in fp16 mode, it selectively places only those parameters that require computation on the CPU, while keeping the rest on the GPU. Typically, it delegates the task of updating the model to the CPU, effectively turning the CPU into a parameter server. To boost efficiency further, as depicted in **Fig. 6.13**, ZeRO-Offload runs communication and computation tasks in parallel, thereby reducing the impact of communication delays on the overall computational workflow.

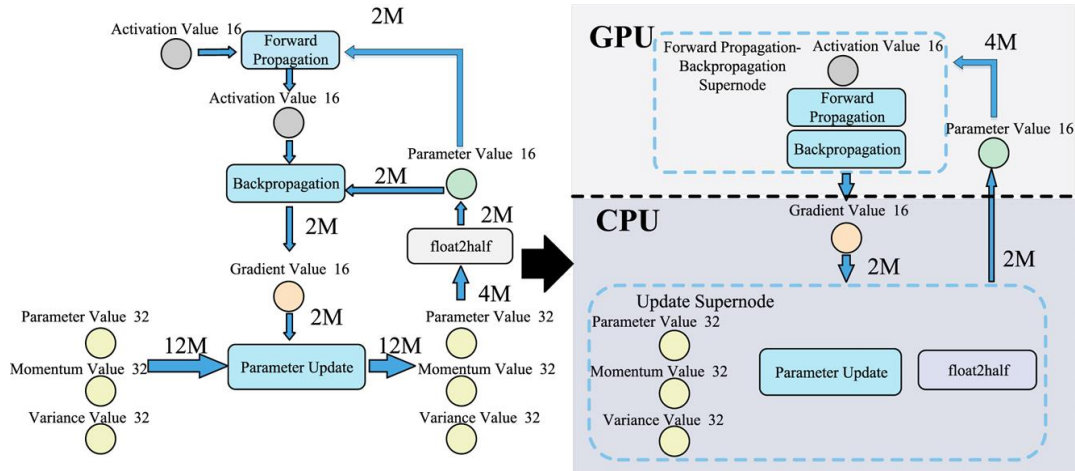


**FIGURE 6.13** Diagram of process parallelism for communication and computation.

The diagram illustrates the process of forward propagation, backpropagation and parameter update in a neural network. On the left side, the process is shown without specifying the hardware. It starts with an activation value of 16, which goes through forward propagation, resulting in another activation value of 16. This value is then used in backpropagation, producing a gradient value of 16. The gradient value is used in the parameter update process, which involves parameter values of 32, momentum values of 32 and variance values of 32. The data flow includes 2M for forward propagation, 2M for backpropagation and 2M for gradient value. The parameter update process involves 12M for parameter value, 12M for momentum value and 12M for variance value. The right side of the diagram specifies the hardware, with the GPU handling forward propagation and backpropagation and the CPU handling the parameter update. The GPU section shows forward propagation and backpropagation with an activation value of 16, resulting in a gradient value of 16. The data flow includes 4M for parameter value and 2M for gradient value. The CPU section shows the update supernode with parameter values of 32, momentum values of 32 and variance values of 32. The parameter update process involves

12M for parameter value, 12M for momentum value and 12M for variance value. The float2half process is also shown, converting parameter values of 16 to 2M.

In the context of backward propagation, GPUs are designed to fill up a storage “bucket” with gradients. As it computes new gradients, it concurrently sends these buckets to the CPU. By the time backward propagation concludes, the CPU typically has updated gradients. Likewise, during parameter updates, the CPU efficiently transfers the updated parameters back to the GPU, ensuring continuous synchronization, as depicted in **Fig. 6.14**.

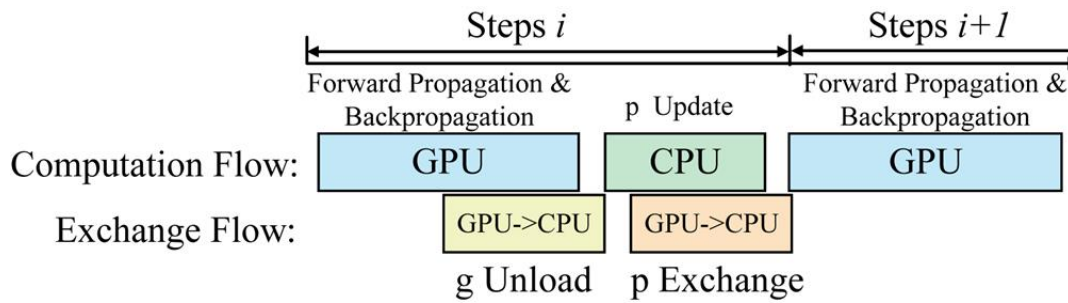


**FIGURE 6.14** Training process of ZeRO-Offload on a single GPU.

The diagram illustrates the computation and exchange flow between GPU and CPU during steps  $i$  and steps  $i$  plus 1. In steps  $i$ , forward propagation and backpropagation occur on the GPU, followed by an update on the CPU. The exchange flow shows data being unloaded from GPU to CPU (**g Unload**) and exchanged from GPU to CPU (**p Exchange**). In steps  $i$  plus 1, forward propagation and backpropagation occur again on the GPU.

ZeRO-Infinity is essentially an enhanced version of ZeRO stage-3, which utilizes non-volatile memory express (NVMe) technology. This system enables the offloading of all model parameters not just

to the CPU, but also to NVMe storage. Leveraging the capabilities of the NVMe protocol allows ZeRO to make additional use of solid-state drives, which helps substantially cut down on memory demands and boosts the speed of data transfers.

## **6.5 Examples of training strategies**

Since the release of GPT-3, OpenAI has chosen to keep its technology private. However, other models like OPT, PaLM, and BLOOM, which are similar in size to GPT-3, openly share their training strategies in their academic papers. These documents offer valuable insights into how they prepare their data before training, choose their architecture, set their training parameters, and manage other elements that are crucial for ensuring the training runs smoothly and reaches its goals effectively.

### **6.5.1 Training framework**

Training LLMs usually involves complex techniques like model parallelism, which includes methods such as tensor and pipeline parallelism, and advanced distributed optimization methods like ZeRO. For instance, OPT utilizes ZeRO's FSDP and Megatron-LM's approach for model parallelism. BLOOM uses ZeRO's Deepspeed and also Megatron-LM's model parallelism strategies. PaLM, on the other hand, employs a system called Pathways which combines model and data parallelism on TPU technology. While the specifics of GPT-3 training system are still not fully disclosed, it is known to incorporate some form of model parallelism.

### **6.5.2 Parameter stability**

Maintaining stable parameters during the training of models like OPT and PaLM is key to their success. For OPT, frequent adjustments such as changing gradient norms, learning rates, and optimizers, and restarting training from the latest checkpoint were

made. These continual changes may have hindered OPT from reaching its full potential. On the other hand, PaLM followed a more consistent strategy. It only restarted training from a checkpoint about 100 steps prior to any sudden increase in loss and skipped around 200–500 batches of data. This approach of minimal intervention helped PaLM achieve remarkable results.

### **6.5.3 Optimizing training settings**

Optimizing training settings is essential for stable model training. PaLM, for example, improved its stability through various enhancements, including using a better version of the Adafactor optimizer, adjusting the logits before the softmax function, adding auxiliary losses that help keep normalization close to zero, using different initialization methods for word vectors and layer weights, omitting bias in certain layers, and avoiding dropout during pretraining. Techniques like applying postlayer normalization with DeepNorm and restricting gradient flow in the word vector layer further stabilize large model training. On the other hand, OPT and BLOOM have not employed these advanced methods, which might explain why they lag behind PaLM in terms of training stability and overall performance.

### **6.5.4 BF16**

In model training, the stability can often be affected by the choices made around training systems and hardware. An important decision in this realm is using bfloat16 (BF16) for storing model weights and activations, which is key because it handles large numbers better than FP16, particularly during spikes in loss, due to having the same exponent size as FP32. Despite these benefits, FP16 limited capacity can still lead to issues, often requiring models to keep weights smaller to avoid frequent overflows, a challenge especially in large models ([Fig. 6.15](#)).

## Floating Point Format

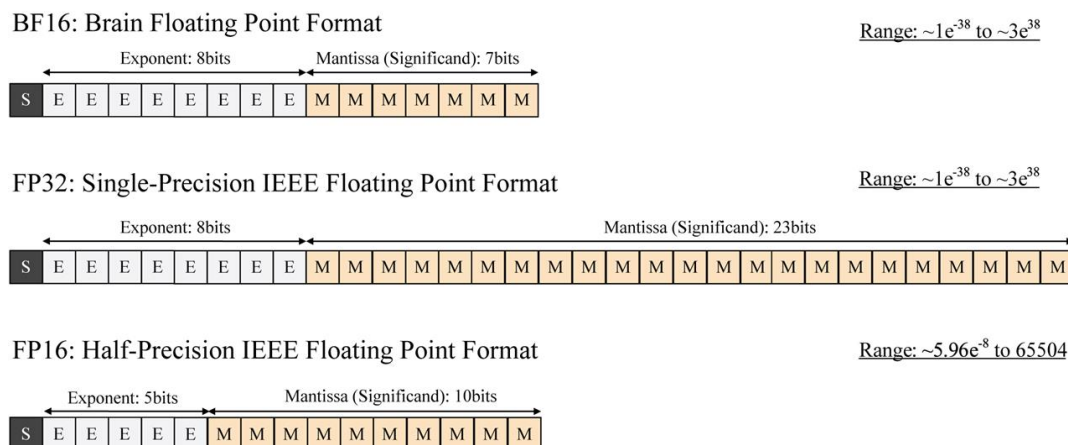


**FIGURE 6.15** Diagram of floating-point formats.

Floating Point Format BF16: Brain Floating Point Format Exponent: 8bits Mantissa (Significand): 7bits Range: approximately  $1e^{-38}$  end superscript to approximately  $3e^{38}$  end superscript FP32: Single-Precision IEEE Floating Point Format Exponent: 8bits Mantissa (Significand): 23bits Range: approximately  $1e^{-38}$  end superscript to approximately  $3e^{38}$  end superscript FP16: Half-Precision IEEE Floating Point Format Exponent: 5bits Mantissa (Significand): 10bits Range: approximately  $5.96e^{-8}$  end superscript to 65504.

Both BF16 and FP16 occupy the same amount of space (2 bytes), but BF16 can be less precise. In practice, whether using BF16 or FP16, a version of the weights is always kept in FP32 to aid in optimization updates. The 16-bit formats are mainly used for calculations, with updates done in full precision FP32 before being converted back for the next processing cycle.

However, GPUs typically prefer FP16, as seen with NVIDIA's V100, which uses it by default for mixed precision training. OPT's choice of FP16 might be contributing to its instability in training.

Meanwhile, BLOOM has addressed the instability seen with FP16 by switching to BF16, which has improved its training process.

### **6.5.5 Other factors**

While not always the main factor, certain choices can significantly influence the overall performance of language models. For instance, both PaLM and GPT-3 gradually increased their batch sizes during training—a strategy that’s known to boost performance for big language models. In contrast, OPT and BLOOM kept their batch sizes constant. The type of activation function used also plays a role; OPT uses ReLU, while PaLM goes with SwiGLU, and both GPT-3 and BLOOM prefer GeLU. Typically, GeLU and SwiGLU are seen to perform better in such models.

In terms of handling longer sequences, these models also differ in how they encode position information. PaLM uses Rotary Position Embeddings, BLOOM applies ALiBi which biases attention scores based on distance instead of embedding position directly, and GPT-3 and OPT use learnable relative position encodings. These choices can affect how well models manage long sequences. These examples highlight how various factors come together to influence the overall effectiveness of models during their pretraining phase.

## **6.6 Summary**

This chapter dives deep into the pretraining strategies for LLMs. It starts with a critical look at how pretraining datasets are built and chosen, highlighting how the diversity and quality of data play a crucial role in the performance of these models. It also discusses methods like data cleaning, deduplication, and augmentation, which are key to boosting a model’s ability to generalize and maintain stability.

In discussing distributed training, the chapter focuses on strategies for data and model parallelism that can dramatically cut



down on training times and save on computational resources. It details tools like Pathways, Megatron-LM, and DeepSpeed Zero, which are essential for efficient distributed training setups.

The chapter concludes with examples of training strategies for large NLP models, discussing choices around training frameworks, model architecture tweaks, training settings, and optimizations like using BF16. These strategies are vital for managing challenges like sudden spikes in loss and ensuring smooth model convergence.

Though the analysis is primarily based on information from other large NLP models of similar size, the strategies and methods discussed are widely applicable and offer valuable insights for optimizing pretraining processes in models like ChatGPT. As computational power grows and new innovative methods emerge, we can expect more breakthroughs in the pretraining strategies for LLMs.

---

## CHAPTER 7

# Proximal policy optimization

---

### Abstract

This chapter provides a comprehensive overview of proximal policy optimization (PPO), a policy gradient method in reinforcement learning that focuses on online learning. The chapter begins by introducing traditional policy gradient methods, emphasizing their direct approach to optimizing policies using real-time data. It then explores the Actor-Critic framework, which combines policy gradients with value function approximation to stabilize learning. Trust Region Policy Optimization (TRPO) is discussed as an enhancement, introducing constraints on policy updates to prevent instability. PPO builds on TRPO by incorporating a clipping mechanism in its objective function, ensuring stable and efficient policy updates without the need for complex Kullback-Leibler (KL) divergence constraints. The chapter concludes by highlighting PPO's effectiveness across various applications while

acknowledging its limitations, which present opportunities for further research and innovation.

## **Keywords**

Proximal policy optimization; policy gradient; Actor-Critic; trust region policy optimization; reinforcement learning; online learning; clipping mechanism; stability; convergence

Proximal policy optimization (PPO) stands out as a policy gradient method, distinct from popular reinforcement learning techniques like Deep Q Network due to its focus on online learning. Policy gradient methods, unlike others that depend on replaying past experiences, learn directly from the real-time data that agents gather in their environments. The Actor-Critic framework refines this approach by using a value function, the Critic, to steer the policy, the Actor, towards more stable and efficient learning outcomes. A specific enhancement within this framework is the Trust Region Policy Optimization (TRPO), which stabilizes learning by setting a “trust region” in the policy space where updates can safely occur without causing instability due to excessive changes.

The PPO algorithm builds upon TRPO by limiting the magnitude of updates in what’s known as PPO, ensuring that the new policy remains similar to the previous one. This approach has paved new avenues for tackling complex tasks in reinforcement learning. It has shown particular efficacy in natural language processing, where it is used to fine-tune behemoths like ChatGPT and InstructGPT. This

chapter explores the journey and merits of PPO, starting from the basics of policy gradients and Actor-Critic methods to the more advanced TRPO and PPO.

## **7.1 Traditional policy gradient methods**

### **7.1.1 The principles of policy gradient methods**

Policy gradient methods form a key group of reinforcement learning algorithms that focus directly on improving policies. These methods help agents figure out the best strategies by interacting with their surroundings. The main concept here is to tweak the policy parameters in a way that aligns with the policy gradient—essentially a direction that suggests how to change the policy to get the most reward over time. In practical terms, a policy maps different situations (states) to actions. To manage this mapping, policy gradients use neural networks, like deep neural networks, to simulate and enhance the policy by tracking and adjusting to these gradients.

Consider a policy  $\pi_{\theta}(a|s)$  determined by parameters  $\theta$ , where  $s$  indicates the state,  $a$  the action, and  $\theta$  the policy parameters. Policy gradient methods strive to find the best set of parameters  $\theta^*$  that maximize expected rewards. These methods typically use a loss function that aims to minimize negative expected rewards, illustrated by the formula:

$$L(\theta) = -E\left[\sum_{t=0}^T \gamma^t \times r(s_t, a_t) | \pi\right] \quad (7.1)$$

where  $\theta$  refers to neural network parameters,  $T$  is the maximum number of time steps,  $\gamma$  is the discount factor reducing future rewards' value, and  $r(s_t, a_t)$  is the reward received for choosing action  $a_t$  in state  $s_t$  at time  $t$ . The goal is to find the values of  $\theta$  that make this loss as small as possible (thereby maximizing the expected rewards). This is done by computing the policy gradient, or how the expected rewards change with small changes in  $\theta$ . The method used for this is gradient ascent, where parameters are updated in the direction that increases expected rewards:

$$\nabla_{\theta} E[\sum_t \gamma^t r_t] = E[\sum_t \gamma^t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \times r_t]. \quad (7.2)$$

The equation shows how to compute the gradient  $\nabla_{\theta}$ , which tells us how to adjust the parameters  $\theta$  of our policy to maximize rewards. Here,  $a_t$  and  $s_t$  represent the action taken and the state observed at each timestep  $t$ , respectively. To find this gradient, the agent interacts with its environment, gathering data on various “trajectories,” each a sequence of states, actions, and rewards. Analyzing these trajectories provides an accurate measure of the policy gradient. This insight is then used to tweak the policy parameters  $\theta$ , effectively teaching the agent to make better decisions over time. For example, a trajectory, denoted by  $\tau$ , captures how an agent's states, actions, and

rewards evolve over time as it interacts with its environment. It's expressed as

$$\tau = (s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_t, a_t, r_t). \quad (7.3)$$

As illustrated in **Fig. 7.1**, by knowing an agent's (or actor's) policy parameters  $\theta$ , we can calculate the likelihood of any given trajectory  $\tau$ . This probability reflects how the agent's decisions, based on its specific actions in certain states, play out over time. Each state and action within the trajectory is determined by sampling from the agent's action probability distribution  $p_{\theta}(a_t | s_t)$  and from the probabilities of transitioning to new states,  $p(s_{t+1} | s_t, a_t)$ . This model helps us understand and predict how the agent will behave in various scenarios.

$$p_{\theta}(\tau) = p(s_1) \prod_{t=1}^T p_{\theta}(a_t | s_t) p(s_{t+1} | s_t, a_t). \quad (7.4)$$



bracket.  $R$  left parenthesis tau right parenthesis is defined as the summation from  $t$  equals 1 to  $T$  of  $r$  sub  $t$  end sub.

In scenarios where an agent operates under the same policy parameters  $\theta$ , it can produce multiple trajectories. Each trajectory  $\tau$  has a specific probability of occurring. By calculating the weighted sum of all these probabilities, along with their associated rewards, we derive the expected value:

$$\nabla \bar{R}_\theta = \sum_{\tau} R(\tau) \nabla p_\theta(\tau) = \mathbb{E}_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)]. \quad (7.5)$$

To approximate this calculation, we use the Monte Carlo method:

$$\nabla \bar{R}_\theta = \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n). \quad (7.6)$$

The parameters are then updated using gradient ascent, where  $\theta$  is adjusted by adding a learning rate  $\eta$  multiplied by the gradient  $\nabla \bar{R}_\theta$ . The learning rate itself can be tuned using methods like Adam or RMSProp:

$$\theta \leftarrow \theta + \eta \nabla \bar{R}_\theta. \quad (7.7)$$

Policy gradient methods stand out because they directly optimize the policy without the need to estimate action or state-value functions, which is particularly beneficial in



environments with high-dimensional continuous action spaces and complex dynamics.

### **7.1.2 Importance sampling**

In policy gradient methods, importance sampling plays a crucial role, especially when it comes to effectively using data previously gathered before updating the policy parameters. The main purpose of importance sampling is to estimate the expected value of one distribution (the target policy) from another (the reference policy). Illustrated in **Fig. 7.2**, during policy updates, we can use importance sampling to draw on trajectories from the old policy to estimate what the expected returns might look like under the new policy. This approach saves the time and computational resources normally spent on gathering new samples.

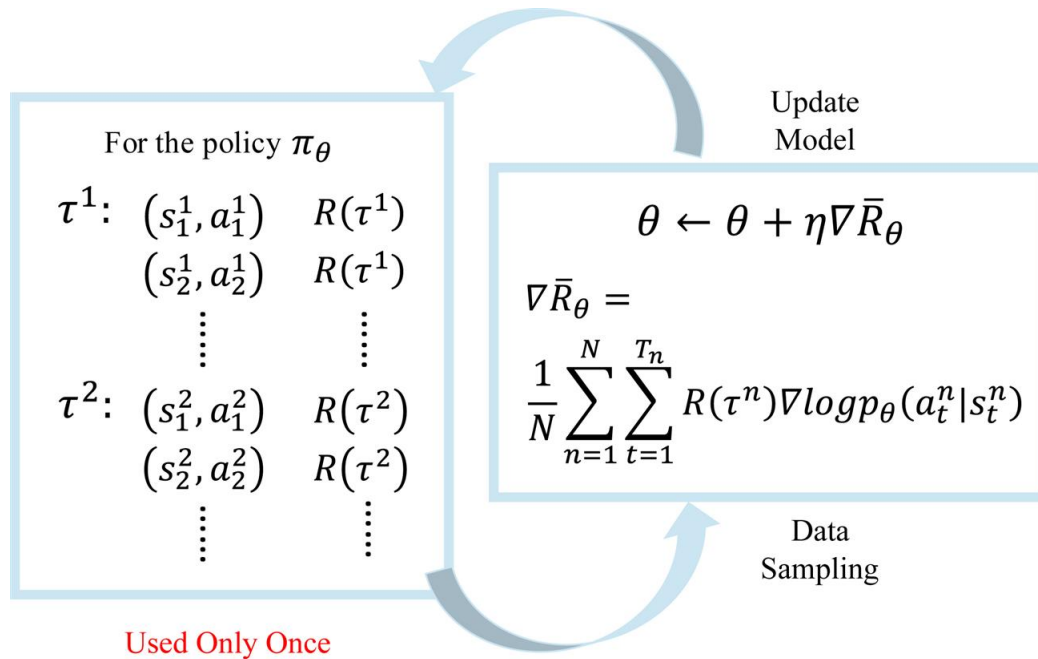


**FIGURE 7.2** Importance sampling in policy gradient methods.

On the left side, it shows the policy  $\pi_\theta$  with  $\tau^1$  and  $\tau^2$ , each containing pairs of  $s_1^1, a_1^1$  and  $s_2^1, a_2^1$ , followed by  $R(\tau^1)$  and  $R(\tau^2)$ . This data is used only once. On the right side, it shows the update model process with  $\theta \leftarrow \theta + \eta \nabla \bar{R}_\theta$ . The gradient  $\nabla \bar{R}_\theta$  is calculated as  $\frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n)$ .

sub t superscript n end superscript vertical bar s sub t superscript n end superscript right parenthesis. The diagram indicates the flow between data sampling and model updating.

Specifically, in these methods, we're interested in the expected returns of state-action pairs under the new target policy  $\pi_{\theta}$ . When policy parameters change from  $\theta$  to  $\theta'$ , the probability associated with an action  $a$  in the same state  $s$  will shift from  $p_{\theta}(s, a)$  to  $p_{\theta'}(s, a)$ . To continue using the same trajectory  $\tau$ , we need to recalculate its expected return  $R(\tau)$  using the formula:

$$\nabla \bar{R}_{\theta'} = \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_{\theta'}(a_t^n | s_t^n). \quad (7.8)$$

This strategy allows for the reuse of trajectories from the reference policy  $\pi_{\theta}$ , employing importance sampling to estimate the expected returns under the new target policy  $\pi_{\theta'}$ . In real-world applications, this means we can update policies using existing trajectories without needing to collect new data each time there's a change, enhancing efficiency.

However, when the target and reference policies are quite different, importance sampling estimates can exhibit high variance. This happens because the trajectories must be weighted, and these weights can vary significantly. Additionally, the unpredictability of the environment and changes in policies can make the estimated returns highly

variable. This variability can cause instability in gradient calculations, which can hinder the learning process. Moreover, since policy gradient methods depend on trajectories sampled from the policy, these trajectories can often be strongly correlated, which might also contribute to high variance in gradient estimations, impacting the algorithm's stability and convergence.

To combat these issues, advantage functions are commonly used in practice to help reduce variance. An advantage function evaluates the benefit of choosing a specific action in a given state compared to the average return. By deducting the average return, we get a more stable estimate of the advantage, which helps reduce the variance from importance sampling and enhances the stability of gradient calculations. However, accurately estimating advantage functions still poses a challenge in real-world applications.

### **7.1.3 Advantage function**

In policy gradient methods, researchers use something called an advantage function to boost learning efficiency and stabilize the algorithm. The advantage function evaluates how much better a specific action is in a given state compared to the average outcome of the current policy. Essentially, it helps identify the extra benefits of choosing a particular action in a certain situation. The advantage function, denoted as  $A_{\pi}(s, a)$ , is defined as

$$A_{\pi}(s, a) = Q_{\pi}(s, a) - V_{\pi}(s) \quad (7.9)$$

where  $Q_{\pi}(s, a)$  is the action-value function, representing the expected total reward for taking action  $a$  in state  $s$  under policy  $\pi$ .  $V_{\pi}(s)$ , on the other hand, is the state-value function, which estimates the expected total reward in state  $s$  under policy  $\pi$ . Therefore, the advantage function quantifies how much more beneficial a specific action  $a$  is compared to what's typically expected under policy  $\pi$  in that state.

Once we include the advantage function, we can update the way we calculate the policy gradient:

$$\nabla_{\theta} E[\sum_t r_t] = E[\sum_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \times A_{\pi}(s_t, a_t)]. \quad (7.10)$$

Introducing the advantage function helps reduce the variance that comes from relying solely on the original reward signals, enhancing both the stability and accuracy of our gradient estimates. This improvement occurs because the advantage function strips out the components related to the state value function, allowing us to focus more on the relative benefits of different actions.

With the advantage function, policy gradient methods can identify and optimize effective actions more swiftly, speeding up the overall learning process.

However, while the advantage function offers clear benefits, calculating it requires the estimation of both the action-value function,  $Q_{\pi}(s, a)$ , and the state-value function,  $V_{\pi}(s)$ , adding to the computational complexity.

## 7.2 Actor-Critic

The Actor-Critic algorithm, as shown in **Fig. 7.3**, is a type of reinforcement learning that combines elements of policy gradient methods and value function approximation. What sets this algorithm apart is how it splits responsibilities: the Actor formulates policies, and the Critic assesses how effective these policies are.

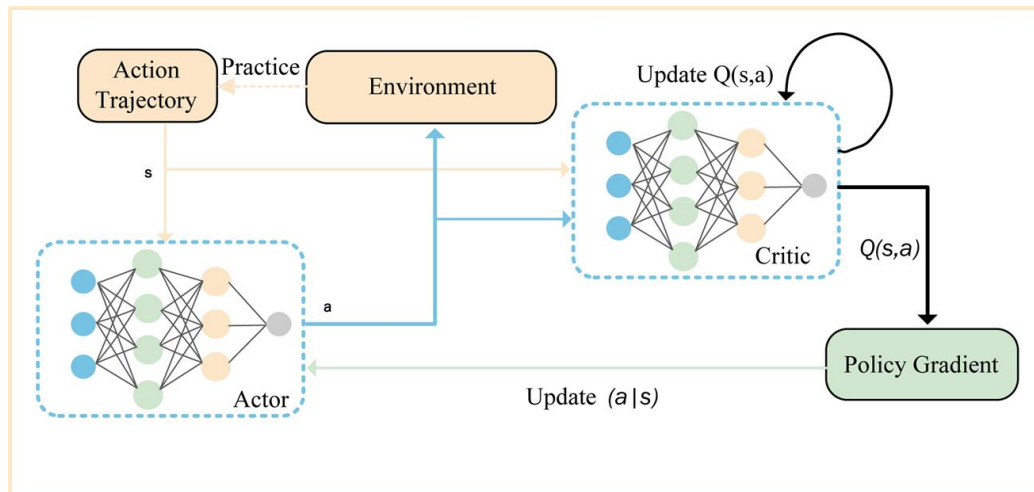


**FIGURE 7.3** Actor-Critic algorithm.

The components include Action Trajectory, Environment, Actor, Critic and Policy Gradient. The Action Trajectory leads to Practice, which interacts with the Environment. The Environment sends information to the Actor, which updates a given action based on state ( $a|s$ ). The Actor is depicted with a neural network structure. The Environment also sends information to the Critic, which updates  $Q(s,a)$ . The Critic is also depicted with a neural network structure. The Critic sends  $Q(s,a)$  to the Policy Gradient, which is then updated. The flow of information is indicated by arrows connecting these components.

Policy gradient methods by themselves often suffer from high variance in their gradient estimates, which can make learning unstable. The Actor-Critic method tackles this by bringing in a Critic to learn and provide a value function, which acts as a reliable baseline to help calculate more stable policy gradients.

While traditional policy gradient methods typically use the Monte Carlo method to calculate returns, which is accurate but variable, the Actor-Critic approach estimates returns using a value function, reducing variance at the cost of a slight bias. This compromise allows the Actor-Critic method to be more stable during training.

In this setup, both the Actor and the Critic are modeled using function approximators like neural networks. The Actor uses a parameterized policy  $\pi_{\theta}(a|s)$  to decide actions, with  $\theta$  as the policy parameters. The Critic, on the other hand, evaluates the expected total rewards for different states using a parameterized value function  $V_{\psi}(s)$ , with  $\psi$  as its parameters. This collaboration between the Actor and Critic helps the agent better manage the trade-offs between exploring new actions and exploiting known strategies, enhancing overall learning efficiency.

### 7.2.1 Algorithm steps

The Actor-Critic algorithm follows these basic steps:

First, the parameters for both the policy ( $\theta$ ) and the value function ( $\psi$ ) are initialized.

During each training episode, these actions are taken:

- a. Begin at the starting state and use the policy  $\pi_{\theta}(a|s)$  to decide an action.
- b. The chosen action is then performed, which leads to receiving the next state  $s'$  and a reward  $r$ .
- c. The Critic computes the temporal difference (TD) error for the value function as follows:



$$\delta = r + \gamma V_{\psi}(s') - V_{\psi}(s).$$

Here,  $\gamma$  represents the discount factor, which adjusts the importance of future rewards.

- d. The TD error  $\delta$  is used to refine the Critic's parameters  $\psi$ .
- e. This same TD error  $\delta$  is also used as a proxy for the advantage function to update the Actor's policy parameters  $\theta$ :

$$\theta = \theta + \alpha \nabla_{\theta} \log \pi_{\theta}(a|s) \times \delta$$

where  $\alpha$  is the learning rate.

- f. If the episode concludes, it's time for the next one; otherwise, update the state to  $s'$  and repeat the steps from a to f.

### 7.2.2 Value function and policy update

In the Actor-Critic algorithm, the value function is intricately linked with how policies are updated. The Critic's role is to learn the value function to assess the expected total rewards for the current policy across various states. Meanwhile, the Actor uses insights from the value function to refine the policy. This dynamic interaction unfolds primarily in these ways:

1. **Advantage function estimation:** The Actor uses this function, which relies on the Critic's value

function, to guide updates, making policy learning more stable and efficient.

2. **TD error as learning signal:** The Actor uses the TD error computed by the Critic to gauge the difference between actual experiences and the expected outcomes predicted by the value function, which helps in fine-tuning the policy.
3. **Adaptive exploration and exploitation:** The Critic's insights allow the Actor to smartly adjust its strategy for exploring new actions and exploiting known rewarding actions, based on the current value assessments.
4. **Dual learning process:** Both the Actor and the Critic learn simultaneously—policy from the Actor and value functions from the Critic. This dual approach not only speeds up learning but also enhances the stability by reducing variability in policy updates.

### 7.2.3 Issues and challenges

While the Actor-Critic algorithm addresses many limitations of traditional policy gradient methods, it still faces several challenges:

1. **Stability of dual-network training:** The Actor and Critic, each trained with their own neural network models, must be managed simultaneously, which can destabilize training. Inaccurate value assessments

from the Critic could potentially misguide the Actor in policy updates.

2. **Exploration strategy issues:** Though the algorithm can adapt its exploration strategies, such as adding noise, these may not be effective in more complex scenarios.
3. **Hyperparameter tuning challenges:** The algorithm requires careful adjustment of several parameters, like learning rates and discount factors, to optimize performance. Finding the right set of parameters can be challenging and time-consuming in real-world settings.
4. **Sample complexity issues:** Improved over simple policy gradient methods, the Actor-Critic still demands a significant amount of training samples, which can be costly and impractical to gather in practical environments.

## 7.3 Trust region policy optimization

TRPO expands on the Actor-Critic algorithm to tackle the instability in policy updates seen in traditional policy gradient methods and the basic Actor-Critic approach.

TRPO's key feature is its restriction on the extent of policy updates, ensuring that changes remain within a defined "trust region." This approach significantly boosts the stability and convergence of policy learning, making it more reliable and effective.

### 7.3.1 Optimization objectives

TRPO not only aims to enhance policy improvements but also controls the changes by limiting the KL divergence (a measure of similarity between probability distributions) between new and old policies. As depicted in **Fig. 7.4**, by keeping the KL divergence within certain limits, TRPO avoids drastic changes to the policies, which helps maintain stability during updates.

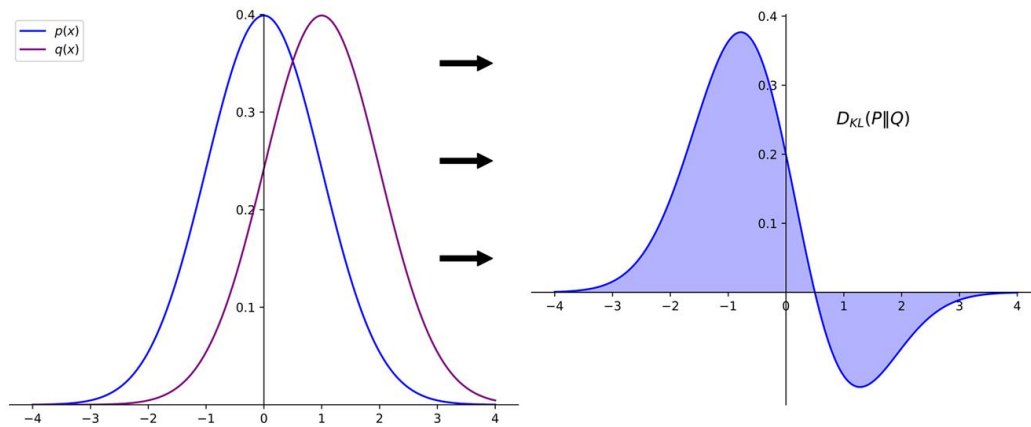


**FIGURE 7.4** KL divergence.

The left graph shows two probability distribution curves labeled  $p(x)$  and  $q(x)$ . The x-axis ranges from negative 4 to positive 4, while the y-axis ranges from 0 to 0.4. The  $p(x)$  curve (blue) peaks around negative 1, while the  $q(x)$  curve (purple) peaks around positive 1. The right graph illustrates the Kullback-Leibler (KL) divergence  $D_{KL}(P||Q)$  between these distributions. Its x-axis also ranges from negative 4 to positive 4, with the y-axis ranging from negative 0.1 to 0.4. The KL divergence is represented by a filled area under a curve, which is mostly positive but has a small negative region. Three arrows point from the left graph to the right, indicating that the KL divergence is derived from the comparison of the two probability distributions. The image relates to the concept of Trust Region Policy Optimization (TRPO), which uses KL divergence to limit policy changes during updates, maintaining stability in the learning process.

During each update, TRPO addresses a constrained optimization problem:

Maximize:

$$E[\pi_{\theta_{\text{new}}}(a|s)/\pi_{\theta_{\text{old}}}(a|s) \times A_{\theta_{\text{old}}}(s, a)]. \quad (7.11)$$

Under the constraint:

$$E[\text{KL}(\pi_{\theta_{\text{old}}}(a|s) \parallel \pi_{\theta_{\text{new}}}(a|s))] < \delta \quad (7.12)$$

where “ $E[\cdot]$ ” denotes expectation, “ $\pi_{\theta_{\text{old}}}(a|s)$ ” is the value under the old policy, “ $\text{KL}(\cdot)$ ” is the KL divergence, and “ $\delta$ ” is a predefined threshold. By solving this optimization challenge, TRPO ensures that policy updates are both effective and stable.

The incorporation of KL divergence into the updating process prevents unstable updates and enhances learning stability. Moreover, by managing the magnitude of updates, TRPO avoids overlooking optimal solutions, thus improving convergence. This strategy results in robust updates that are reliable even in complex scenarios and noisy environments.

### 7.3.2 Limitations

While TRPO excels in many areas, it is not without its challenges:

1. **Computational complexity:** TRPO controls policy updates through constrained optimization, each

requiring the resolution of complex problems. This demand for high computational power can be a limiting factor in large-scale or real-time scenarios.

2. **Need for second-order derivatives:** The algorithm requires calculating second-order derivatives, like the Hessian matrix, especially using methods such as conjugate gradient. This computation can be particularly intensive for large neural networks.
3. **Hyperparameter tuning:** Despite its relative stability, TRPO still necessitates fine-tuning hyperparameters, such as the KL divergence threshold  $\delta$ , which can be a cumbersome and time-consuming task in practice.
4. **Handling discrete actions:** While effective in continuous action spaces, TRPO's performance dips in discrete action settings. This is primarily due to its conservative updates, mandated by the limitation on KL divergence, which might not be optimal in discrete scenarios.

Overall, TRPO's introduction of KL divergence constraints boosts the stability, convergence, and robustness of policy updates, leading to significant enhancements in many reinforcement learning tasks. However, its computational demands among other challenges present notable limitations.

## 7.4 Principles of the proximal policy optimization algorithm

PPO is designed to improve agent performance by updating policies in a way that avoids the pitfalls of earlier approaches. Unlike traditional policy gradient methods, PPO introduces an objective function with a clipping mechanism, ensuring that policy updates don't stray too far from the current policy, thus simplifying the process by sidestepping the complex KL divergence constraints typical of TRPO.

In practice, PPO operates with two threads: one interacts with the environment to generate data, while the other uses this data to train the policy network using PPO's specific objective function. This method has proven to be competitive, often outperforming TRPO in a variety of tasks.

The core of PPO's effectiveness lies in its loss function, which comprises three parts:

**1. Clipping objective function:** This part aims to refine the policy to maximize expected returns. It works as follows:

$$L^{\text{CLIP}}(\theta) = E_t [\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)] \quad (7.13)$$

where " $E_t[\cdot]$ " denotes the expected value at time step  $t$ , " $r_t(\theta)$ " is the ratio of probabilities between the new and the current policy, and " $\hat{A}_t$ " represents the estimated advantage at that state. The function uses both the standard policy gradient target, which promotes actions with high advantages, and a clipped version of the probability ratio,



constrained between  $1 - \epsilon$  and  $1 + \epsilon$ , with  $\epsilon$  typically set around 0.2. The clipping ensures the policy updates do not stray too far from the original, maintaining a balance between exploration and exploitation.

**2. Value estimation loss:** This component focuses on refining the value function to enhance its accuracy in estimating the values of states. It is mathematically represented as

$$L^{VF}(\theta) = E_t \left[ (V(s_t) - V_{\text{target}})^2 \right] \quad (7.14)$$

where “ $V(s_t)$ ” is the value function’s prediction for the state “ $s_t$ ,” and “ $V_{\text{target}}$ ” represents the actual observed return.

**3. Entropy loss:** This component aims to boost the entropy of the policy, fostering greater exploration. It is defined mathematically as

$$L^{\text{Entropy}}(\theta) = - E_t \left[ \sum_a \pi(a|s_t; \theta) \log \pi(a|s_t; \theta) \right] \quad (7.15)$$

where “ $\pi(a|s_t; \theta)$ ” indicates the probability of choosing action “ $a$ ” in state “ $s_t$ ,” with “ $\theta$ ” representing the parameters of the policy network.

The overall loss function for PPO integrates these three components into a weighted sum:

$$L(\theta) = L^{\text{CLIP}}(\theta) + c_1 L^{VF}(\theta) + c_2 L^{\text{Entropy}}(\theta). \quad (7.16)$$

In this context,  $\epsilon_1$  and  $\epsilon_2$  serve as hyperparameters, which are essentially tuning knobs to balance the importance of various components within the model.

At the heart of the PPO algorithm is a strategic component known as the clipped PPO objective function. Central to this function is the advantage function  $\hat{A}_t$ , which gauges how advantageous an action  $a_t$  is at a given state  $s_t$  compared to typical outcomes. Illustrated in **Fig. 7.5**, the advantage function can either be positive, suggesting an action is better than average, or negative, indicating it is worse. The function's graph varies accordingly: on the left side where the advantage is positive, the model promotes taking such beneficial actions; on the right side where the advantage is negative, it discourages them.

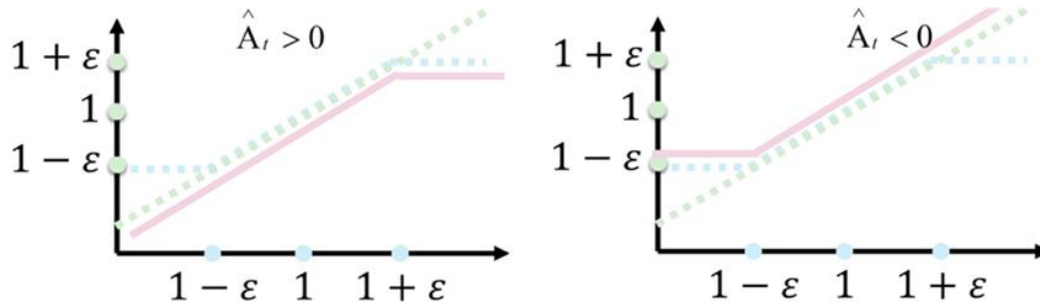


**FIGURE 7.5** Graph of different values of  $\hat{A}_t$  in the objective function.

The left graph shows the advantage function greater than zero, with the x-axis ranging from 1 minus epsilon to 1 plus epsilon and the y-axis ranging from 1 minus epsilon to 1 plus epsilon. The right graph shows the advantage function less than zero, with the same x-axis and y-axis ranges. Both graphs have three data points on the x-axis at 1 minus epsilon, 1 and 1 plus epsilon and three data points on the y-axis at 1 minus epsilon, 1 and 1 plus epsilon.

In practical terms, if the action ratio  $r_t(\theta)$  veers too high, the model flattens its loss curve to signify a higher likelihood of these actions under the new policy compared to the old. This area is deliberately truncated to prevent the model from overly adjusting based on this data, thus moderating the influence of such updates. When actions yield negative outcomes, the model flattens this function as the action ratio approaches zero, reflecting their improbability under the current guidelines and preventing drastic changes that might otherwise zero out these actions.

A crucial aspect to remember is that the advantage function can be noisy; thus, sweeping changes to the policy based solely on this data are discouraged. The most extreme scenario, where a previous update substantially increased an action's likelihood and degraded the overall strategy, calls for a potential rollback facilitated by the PPO's function. In such cases, negative gradients help recalibrate the policy by decreasing the likelihood of those actions, aligning the adjustment with the scale of the initial misjudgment. This delicate balancing act ensures that the policy refines itself, focusing on minimizing loss where it counts.

The PPO algorithm was crafted by experts with a nuanced grasp of its objective function's behavior, which is crucial for devising specific reward functions. This deep understanding was likely conveyed through a series of detailed diagrams before they finalized the precise objective function to ensure reliable system operations. If the minutiae of this process seem daunting, don't fret—understanding the essence is sufficient. Both PPO and its cousin TRPO share a common goal: they employ conservative updates to prevent significant deviations from the current policy. However, PPO stands out by employing a simplified objective function that eschews the need for calculating additional constraints or the KL divergence. This simplicity has proven to give PPO an edge over TRPO's more elaborate mechanisms.

Now, let's explore how PPO functions. It operates in a cycle of two alternating phases. Initially, the policy interacts with the environment, generating data trails which are immediately analyzed using a tailored baseline model to estimate state values and compute advantage functions. Following this, the second phase updates the policy using these trails with a policy gradient method that integrates the advantage function into PPO's streamlined objective function. The baseline estimator, often a neural network, is also updated to refine its accuracy. This iterative process—switching between collecting data and updating the policy—continues until certain goals, like a specific number of iterations or performance benchmarks, are met.

PPO's merits, particularly its stability, adaptability, and straightforward design, have made it a preferred choice for complex applications like robotics and gaming AI. The algorithm cleverly balances conservative updates with efficient use of data to optimize policy adjustments.

Despite its successes, PPO faces hurdles, notably in scenarios with sparse rewards or intricate settings. Addressing these issues has led researchers to integrate sophisticated strategies, including hierarchical learning frameworks and predictive controls, into the PPO architecture. Ongoing innovations such as adaptive clipping and variable learning rates are also being investigated to push PPO's performance further.

## 7.5 Summary

This chapter delves deeply into several policy optimization algorithms, including policy gradient methods, Actor-Critic, TRPO, and PPO, enriching the reader's understanding with a mix of theoretical underpinnings and practical insights. Among these, PPO stands out as a particularly revered tool in the field of reinforcement learning due to its robust effectiveness and broad applicability. Known for its straightforwardness and stability, PPO has demonstrated exceptional performance across a range of scenarios. Nevertheless, it is its very limitations on certain fronts that open up generous opportunities for future research, with the goal of pushing the boundaries of the algorithm's capabilities even further.

---

## CHAPTER 8

# Human feedback reinforcement learning

---

### Abstract

This chapter delves into the critical role of human feedback reinforcement learning (HFRL) in refining and optimizing GPT models, particularly ChatGPT and InstructGPT. The chapter begins by explaining how reinforcement learning, driven by human feedback, enhances model alignment with human expectations. It then outlines the evolution of GPT models, from GPT-3 to GPT-3.5, highlighting key developments like Codex and InstructGPT. The chapter details the three training stages of HFRL—supervised fine-tuning, reward modeling, and reinforcement learning—emphasizing their importance in improving model performance. The discussion extends to the datasets used for fine-tuning, the selection and tasks of annotators, and the challenges associated with multiturn dialogue capabilities. Lastly, the necessity of HFRL is underscored as a means to equip models with robust

adaptability and learning capabilities essential for achieving true artificial intelligence.

## **Keywords**

Human feedback reinforcement learning (HFRL); Codex; InstructGPT; supervised fine-tuning; reward modeling; reinforcement learning; proximal policy optimization (PPO); multi-turn dialogue; dataset annotation

Generative pre-trained transformer (GPT) models undergo a crucial pretraining phase where they learn to mimic the distribution of their training datasets. This phase is vital as the quality of the training data significantly influences the content these models generate. To enhance the quality of this content, it is essential to align the model's outputs with human expectations in a controlled manner. "Alignment" here refers to the match between what the models produce and what humans anticipate. These expectations include not only the fluency and grammatical accuracy of the content but also its relevance, authenticity, and safety.

Reinforcement learning operates through a reward mechanism that directs model training. This mechanism serves as the training's optimization goal. Rewards are versatile and can vary widely, such as in AlphaGO, where rewards are determined by the game's outcome. In reinforcement learning, strategies are developed by extensively sampling rewards to estimate the optimization goal, thereby steering the training process. Thus, human feedback can be treated as a form of reward, introducing



the concept of reinforcement learning driven by human feedback.

## **8.1 Reinforcement learning in ChatGPT**

In May 2020, OpenAI introduced the Davinci API, boasting an impressive 175 billion parameters. Following this, OpenAI embarked on two significant explorations: Codex, which focuses on coding and reasoning, and InstructGPT, which aimed at better understanding human intentions. The technological developments from GPT-3 to GPT-3.5 are illustrated in **Fig. 8.1**.

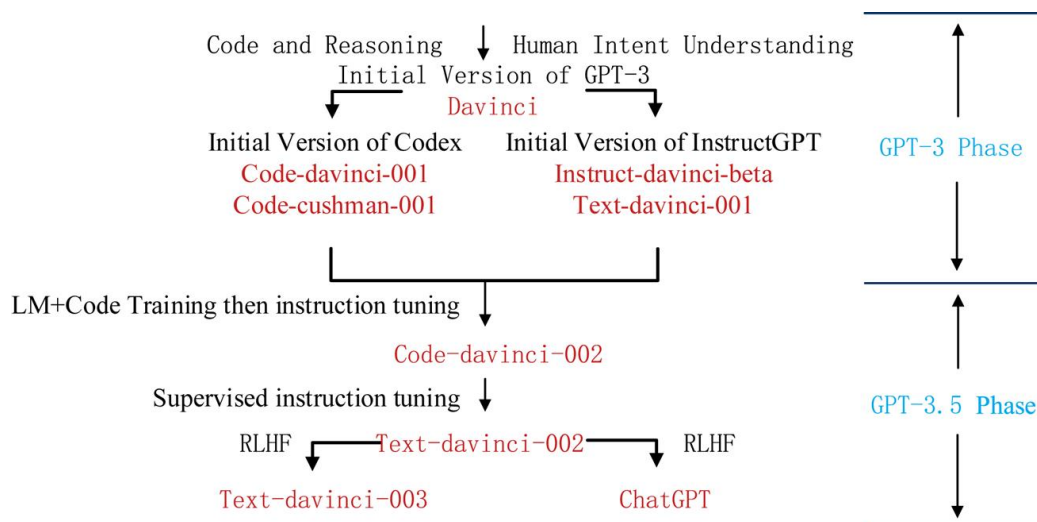


**FIGURE 8.1** Technology roadmap from GPT-3 to GPT-3.5.

It starts with the initial version of GPT-3, which branches into two: Code and Reasoning leading to the Initial Version of Codex (Code-davinci-001, Code-cushman-001) and Human Intent Understanding leading to the Initial Version of InstructGPT (Instruct-davinci-beta, Text-davinci-001). Both branches converge into LMplusCode Training then instruction tuning, resulting in Code-davinci-002. This further leads to Supervised instruction tuning, branching into Text-davinci-002 and RLHF, which then leads to Text-davinci-003 and ChatGPT. The phases are labeled as GPT-3 Phase and GPT-3.5 Phase.

The first pathway concentrates on boosting abilities in coding and reasoning. Here, OpenAI merged GPT-3 with coding training to create Codex. In July 2021, the initial Codex model was launched, fine-tuning the original GPT-3 using 159GB of Python code. This model eventually evolved

into what is now known in the OpenAI API as code-cushman-001, showcasing superior coding and reasoning skills. Despite its comparatively modest text-processing capabilities, the model improved significantly by learning from extensive code designed to tackle mathematical reasoning challenges.

The second pathway delves deeper into understanding human nuances. Google pioneered the use of reinforcement learning from human feedback (RLHF) in 2017. They introduced a method that enhances the performance of reinforcement learning in simulations involving robots and Atari games through feedback derived from human annotations.

To enhance alignment with human intents, OpenAI employed RLHF to refine GPT-3, resulting in the creation of InstructGPT. Initial explorations with RLHF began in 2020 with GPT-3, although these early models sometimes produced inappropriate or hazardous content. It was not until March 2022, detailed in the InstructGPT paper, that OpenAI effectively harnessed RLHF, introducing key APIs such as instruct-davinci-beta and text-davinci-001.

OpenAI's endeavors in these areas culminated in the development of code-davinci-002 (refer to OpenAI's model index documentation for more details). While Codex may seem focused solely on coding, code-davinci-002 could be considered the most potent GPT-3.5 variant for natural language processing (NLP)—even surpassing text-davinci-002 and -003. This is likely due to code-davinci-002 being

trained on both text and code, and being finely tuned for instruction alignment.

Released in May 2022, the text-davinci-002 model was fine-tuned under supervised instructions based on code-davinci-002. This targeted fine-tuning may have reduced its contextual learning prowess but significantly improved its zero-shot learning capabilities. Later, in November 2022, the text-davinci-003 and ChatGPT (gpt-3.5-turbo) were introduced as different variants fine-tuned through reinforcement learning based on human feedback. Text-davinci-003 reclaimed some contextual learning capabilities that text-davinci-002 had partially lost, likely due to integrating language modeling into the fine-tuning process, and it further enhanced its abilities to understand text intentions, perform logical reasoning, and conduct causal analysis, benefiting from RLHF.

Released in May 2022, the text-davinci-002 model was fine-tuned under supervised instructions based on code-davinci-002. This targeted fine-tuning may have reduced its contextual learning prowess but significantly improved its zero-shot learning capabilities. Later, in November 2022, the text-davinci-003 and ChatGPT (gpt-3.5-turbo) were introduced as different variants fine-tuned through reinforcement learning based on human feedback. Text-davinci-003 reclaimed some contextual learning capabilities that text-davinci-002 had partially lost, likely due to integrating language modeling into the fine-tuning process, and it further enhanced its abilities to understand text

intentions, perform logical reasoning, and conduct causal analysis, benefiting from RLHF.

ChatGPT and InstructGPT share similar training approaches. ChatGPT's understanding of human intentions, fueled by RLHF and the incorporation of extensive coding examples, particularly those solving mathematical problems and code annotations, has significantly advanced its capabilities in NLP and code generation. Although there are some variations in the training datasets between ChatGPT and InstructGPT, their foundational training methodologies remain largely identical. The subsequent section will use InstructGPT as a case study to detail the training and evaluation strategies employed in ChatGPT|InstructGPT, which leverage reinforcement learning from human feedback.

## **8.2 InstructGPT training dataset**

The training of InstructGPT/ChatGPT is structured around three pivotal stages: supervised fine-tuning (SFT), reward modeling (RM), and reinforcement learning (RL), each associated with its distinct dataset—SFT, RM, and RL datasets, respectively. In the SFT phase, annotators craft manual responses to prompts within the samples. This stage demands considerable human input and requires annotators with a high level of expertise. During the reward modeling phase, annotators rank several (4–9) model-generated outputs. Although the demands on annotators are somewhat less stringent during this phase,

they still must be well acquainted with the evaluation criteria to ensure the rankings align with expected standards, thereby preventing inconsistent outcomes.

### **8.2.1 Sources of fine-tuning datasets**

The fine-tuning data for the models mainly derives from two key sources:

1. Prompts submitted through the OpenAI API, predominantly sourced from the OpenAI Playground. When users opt for the InstructGPT model, they receive a notification alerting them that their submitted prompts may contribute to the training dataset for newer model versions. For privacy and compliance reasons, API data from the actual product are not utilized. Data from the API excludes prompts with repetitive, lengthy prefixes and restricts the number of prompts per user to under 200 to maintain diversity. The datasets are also segmented by user ID to prevent overlaps between training inputs and those in the validation and test sets. Moreover, all prompts that contain personally identifiable information are systematically excluded to safeguard user privacy.
2. Prompts crafted by annotators, which fall into three categories:
  - a. Simple tasks: Annotators are asked to create straightforward tasks, ensuring a variety of tasks is covered.

- b. Few-sample tasks: Here, annotators develop multiple (query, response) pairs for a given prompt. For example, for a prompt like “Conduct sentiment analysis on this tweet,” the query would be a real tweet, and the response would be classified as either “positive” or “negative.” If K pairs are created, the initial K-1 pairs form the context.
- c. User data: This involves taking use cases from the OpenAI API and crafting corresponding prompts, often rewriting them to meet standardization requirements. These types are systematically categorized into 10 distinct classes, as detailed in **Table 8.1**.

**Table 8.1**

**Distribution and examples of different types of use cases based on user data.**

| Use case type  | Percentage | Example                                                                                                                                                                             |
|----------------|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Brainstorming  | 11.2%      | What are the next 10 sci-fi books I should read?                                                                                                                                    |
| Categorization | 3.5%       | Using the text below, rate this person’s level of sarcasm on a scale of 1 to 10 (1 = not at all sarcastic, 10 = extremely sarcastic), and provide an explanation: {text}<br>Rating: |
| Extraction     | 1.9%       | Extract all place names from the following article: {news article}                                                                                                                  |
| Generation     | 45.6%      | This is the message I received: {email} Here are some key points of the reply: {message}<br>Write a detailed reply                                                                  |
| Rewriting      | 8.4%       | Rewrite the following text to make it more concise:                                                                                                                                 |



| Use case type     | Percentage | Example                                                                                                                    |
|-------------------|------------|----------------------------------------------------------------------------------------------------------------------------|
|                   |            | {very formal text}                                                                                                         |
| Chatting          | 6.6%       | This is a conversation with the Buddha. Each response is filled with wisdom and love. Me: How can I achieve greater peace? |
| Specific question | 2.6%       | Tell me the differences between hydrogen and helium, using the following facts: {list of facts}                            |
| Open question     | 12.4%      | Who built the Great Wall?                                                                                                  |
| Summarization     | 4.2%       | Summarize the following text for a second grader: {text}                                                                   |
| Other             | 3.5%       | Search “cowboy” on Google and return the results                                                                           |

Typically, prompts used for brainstorming and open-ended question-and-answer sessions are shorter, whereas prompts for dialogues and summaries tend to be longer.

### 8.2.2 Annotation standards

The selection criteria for annotators are based on two key aspects:

1. Consistency in identifying sensitive content. The research team has compiled a dataset that includes sensitive materials, each entry comprising a prompt and its completion. If any section of the prompt or completion features content related to pornography, politics, violence, etc., it is classified as sensitive. Annotators are tasked with determining the sensitivity of the content to gauge the alignment between their judgments and those of the research team.
2. Consistency in assessing the quality of model outputs. Using API-generated prompts and the varied responses provided by different models, annotators are asked to rank the outputs based on quality. The agreement between the annotators' and researchers' rankings is then evaluated to ensure consistency in quality assessment.

Annotators have the following primary duties:

1. Crafting prompts.
2. Responding to prompts.
3. Comparatively ranking outputs from different models for the same prompt.
4. In the evaluation phase, feeding prompts that contain coarse, sexually explicit, or violent content into an

InstructGPT model trained through PPO to assess both the toxicity and fluency of the outputs.

Notably, tasks (3) and (4) require annotators to assess the quality and toxicity of content, providing them with clear operational guidelines.

For task (3), high-quality outputs must satisfy three standards: usefulness, truthfulness, and harmlessness. Typically, truthfulness and harmlessness are more critical than usefulness. However, if an output “a” is significantly more useful than output “b”, and its relative harmfulness and untruthfulness are not overly severe—especially if it doesn’t involve high-risk domains like law or medicine—then output “a” is considered superior to “b.” The fundamental sorting criterion is which output one would expect in response to a machine-posed question.

For task (4), two primary metrics are utilized: toxicity, which determines if the output includes vulgar, disrespectful, or sexually violent content, evaluated both in absolute and relative terms to the prompt’s toxicity; and fluency, which assesses the text’s coherence and natural presence in a digital environment.

In the ranking tasks, the approach involves first allowing annotators to independently rate each output following established standards, followed by a collective display of all outputs for final ranking. This method prevents the bias of simultaneous display and rankings, improving evaluation accuracy.

### **8.2.3 Data analysis**

**Table 8.2** outlines the origins and sizes of each dataset:

- **SFT dataset:** Comprising 13,000 prompts jointly supplied by the API and annotators, this dataset is vital for training the SFT model. Annotators provide corresponding answers for these prompts. It is crucial to ensure that each entry in the SFT dataset includes both a prompt and its answer.
- **RM dataset:** Featuring 33,000 prompts, this dataset is contributed by both the API and annotators. Here, annotators are tasked with ranking the model outputs, and this data serves as the training set for the RM model.
- **PPO dataset:** Consisting of 31,000 prompts solely from the API and lacking any annotation, this dataset is utilized specifically for RLHF fine-tuning.

---

## **Table 8.2**

Appendix A<sup>1</sup> of the research paper provides a thorough discussion on the distribution of data used in the training of InstructGPT. The chapter highlights several critical factors potentially influencing the model's effectiveness:

1. More than 96% of the data are in English, with the remaining under 4% comprising 20 other languages, such as Chinese, French, and Spanish. This suggests that while InstructGPT is capable of processing inputs in multiple languages, its performance may be less effective with nonEnglish inputs.

2. There are nine distinct types of prompts, predominantly generative tasks. This indicates that the model might not be fully equipped to handle every type of task.
3. The 40 outsourced annotators, who are responsible for labeling, are from the USA and Southeast Asia, representing a small and geographically concentrated group. Given that InstructGPT aims to train a model that embodies appropriate values, the relatively narrow diversity of these annotators could introduce biases, particularly on issues of significant concern in specific regions.

This discussion sheds light on InstructGPT's dataset. Although ChatGPT and InstructGPT employ similar training methodologies, there are notable differences in their dataset compositions. Currently, detailed comparisons of these differences are scarce. Considering ChatGPT's primary application in conversational settings, it is likely that its dataset differs from InstructGPT in two key aspects: a higher proportion of conversational tasks and a transformation of prompt formats into a question-and-answer style. For a more precise understanding, we await the release of detailed documents, such as ChatGPT's research papers and source codes.

## **8.3 Training stages of human feedback reinforcement learning**

To more closely align language models with user intentions, supervised learning and reinforcement learning fine-tuning strategies informed by human feedback are utilized. This approach not only expands the capabilities of these models but also enhances their precision in adhering to user directives. As depicted in **Fig. 8.2**, the human feedback fine-tuning process for InstructGPT involves three principal phases: SFT, training of the reward model, and the reinforcement learning stage. Given that the training methodologies for InstructGPT and ChatGPT share significant similarities, the forthcoming discussion will focus on InstructGPT as a representative example.

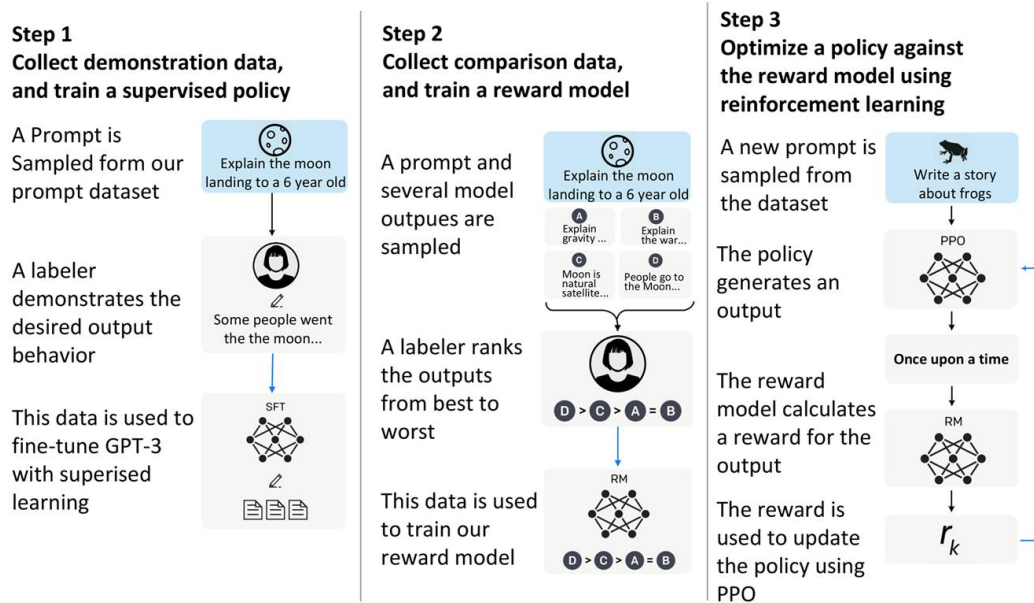


**FIGURE 8.2** Human feedback fine-tuning process of InstructGPT. Image source: Paper “Training language models to follow instructions with human feedback”.

Step 1: Collect demonstration data and train a supervised policy. A prompt is sampled from the prompt dataset. A labeler demonstrates the desired output behavior. This data is used to fine-tune GPT-3 with supervised learning.

Step 2: Collect comparison data and train a reward model. A prompt and several model outputs are sampled. A labeler ranks the outputs from best to worst. This data is used to train the reward model.

Step 3: Optimize a policy against the reward model using reinforcement learning. A new prompt is sampled from the dataset. The policy generates an output. The reward model calculates a reward for the output. The reward is used to update the policy using PPO.

### 8.3.1 Supervised fine-tuning

SFT is a training technique that utilizes the previously mentioned SFT dataset to fine-tune the pretrained GPT-3 model. Known as the SFT model, it displays the foundational predictive abilities of a baseline model, as depicted in **Fig. 8.3**, with a parameter count reaching 175 billion. The fine-tuning approach is akin to that used for GPT-1. Specifically, it incorporates a cosine learning rate decay strategy across 16 training epochs, and to mitigate overfitting, a residual dropout rate of 0.2 is established. These measures are taken to ensure that the model effectively learns and adapts to new data during fine-tuning, thereby improving its predictive performance.

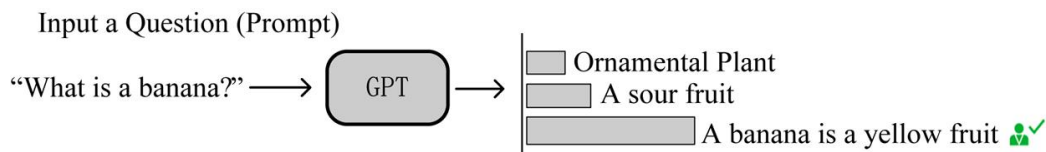


**FIGURE 8.3** Example of supervised fine-tuning.

The output options are Ornamental Plant, A sour fruit and A banana is a yellow fruit. The correct answer is indicated as A banana is a yellow fruit.

Due to the limited training data volume of only 13K samples, the GPT-3 model is susceptible to overfitting. However, this model serves primarily as an initial baseline for further training and not for direct practical application.



Consequently, even if overfitting occurs during the SFT stage, it can still aid in enhancing the performance of later models.

So, is it feasible to align the SFT model with human needs solely through fine-tuning? Clearly, this presents several challenges. For instance, manually crafting answers for all queries can be daunting due to the need for an in-depth understanding of the question content and substantial time, knowledge, and skills to provide diverse, high-quality answers. Thus, while fine-tuning is a potent method of training models, real-world applications might necessitate the integration of additional strategies like reinforcement learning to better meet human requirements.

### **8.3.2 Reward modeling**

Given the impracticality of humans writing answers for every conceivable query, a reward model is trained to discern human preferences. This model processes input prompts and answers to produce a scalar reward value. Illustrated in [Fig. 8.4](#), both API-generated and manually crafted prompts are fed into the SFT model, which then generates answers. These answers are derived through probabilistic sampling and subsequently ranked by annotators. After obtaining the rankings, a reward model is trained by removing the final nonembedding layer of the SFT model and incorporating a linear layer post-Softmax.

This configuration maps the output sequence to a scalar value—essentially rating the prompt–response pair.

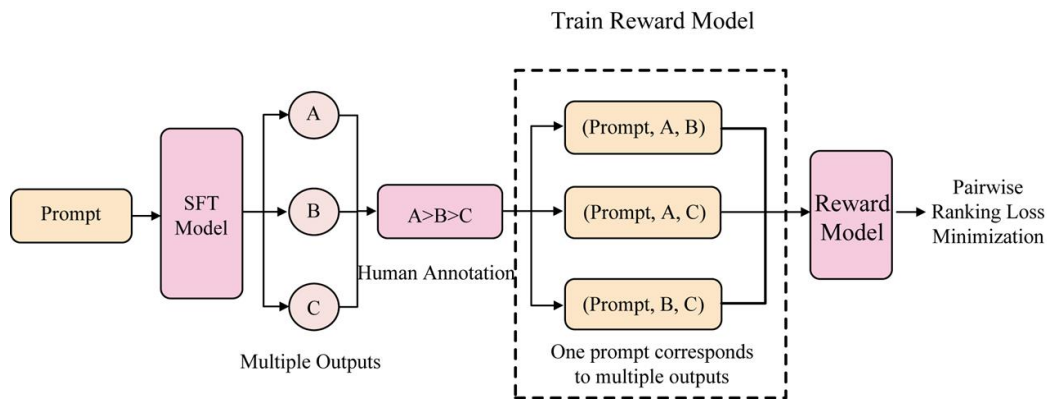


**FIGURE 8.4** Training process of the RM model.

It starts with a prompt that is fed into an SFT model, which generates multiple outputs labeled A, B and C. Human annotation ranks these outputs as A greater than B greater than C. The next step involves training the reward model using pairs of prompts and outputs (Prompt, A, B), (Prompt, A, C) and (Prompt, B, C). This process is aimed at pairwise ranking loss minimization.

Notably, InstructGPT employs a smaller 6B parameter GPT-3 fine-tuned (SFT) model for training the reward model instead of the maximal 175B parameter version. This choice is due to the larger models' propensity for instability during training, while smaller models offer computational efficiency.

### 8.3.3 Reinforcement learning

On the foundation of reward modeling, the PPO algorithm previously discussed is employed to fine-tune the supervised strategy using outputs from the RM as scalar rewards to enhance this reward mechanism. As illustrated in **Fig. 8.5**, the reinforcement learning phase progresses by further fine-tuning the SFT model. It inputs some unannotated prompts (like “Write a diary entry for a spring outing” from a set of 31k prompts from the API) into the SFT model. At this stage, instead of relying on manual quality evaluations, the reward model developed in the second phase scores the outputs of the SFT model. Initially, the SFT model acts as the policy network within the PPO framework, optimizing parameters to elevate the scores of the generated answers. Responses are then regenerated based on the refined policy, assessed by the reward model, leading to further optimizations of the SFT model parameters in a cycle until the policy achieves optimality.

## Reinforcement Learning from Human Feedback (RLHF)

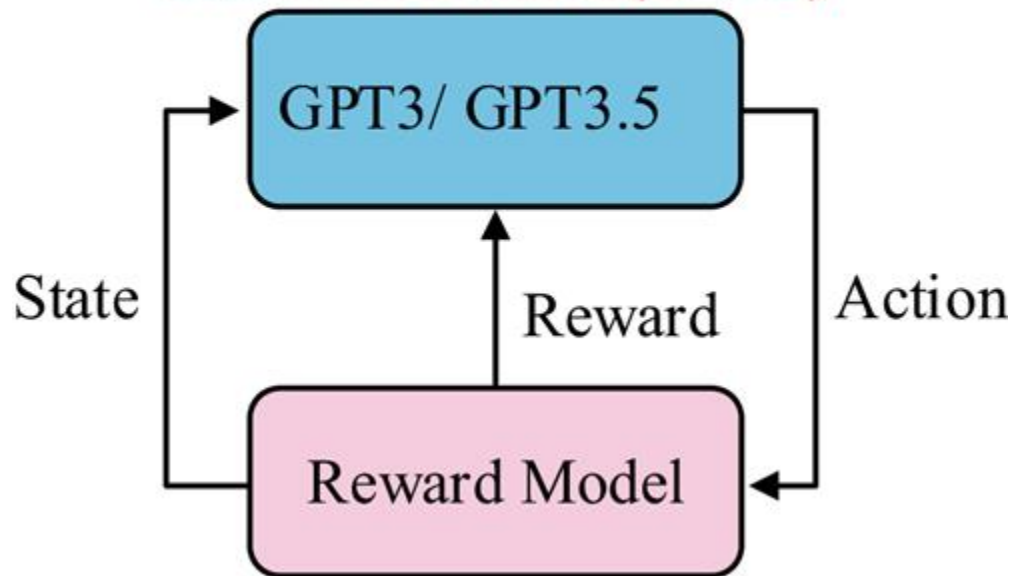


**FIGURE 8.5** Diagram of reinforcement learning with human feedback.

It involves two main components: GPT3/GPT3.5 and a Reward Model. The connections between these components are as follows: State leads to GPT3/GPT3.5, which then leads to Action. Action connects to the Reward Model, which then provides Reward back to GPT3/GPT3.5.

During the three phases of training with human feedback reinforcement learning (HFRL), the reward model and PPO fine-tuning phase are iteratively refined, gathering more comparative data from the best current strategies for training new reward models and strategies. However, this iterative process introduces the challenge of shifting data distributions. As the model evolves, so does the data

distribution—a departure from traditional statistical learning approaches, necessitating continual adaptation to the evolving data environment.

During the process of SFT and reward modeling, one might question why a ranking model is necessary when labeled data already exists—why not proceed directly with training? The answer lies in the nature of the data available; only rankings are available, not explicit outputs  $r$ . If outputs  $r$  were directly labeled, the approach would fall under supervised learning. However, reinforcement learning typically lacks such explicit labels, necessitating the training of a ranking model to assimilate feedback from the environment. This assists the model in comprehending and adjusting to new data distributions, enabling it to learn effectively within an ever-evolving environment.

In the realm of reinforcement learning, there is a concept known as online learning, prevalent in applications like gaming. Here, the model is in a state of continuous update, engaging with the environment and assimilating instant feedback. Adopting online learning would require a person to constantly provide ranking feedback for the model's generated responses and update the model accordingly. This approach, however, can be inefficient as aligning the pace of model training with manual annotation rates is challenging.

Thus, training a ranking model  $R_\theta$  to substitute for human ranking has been adopted to facilitate real-time feedback during the training process. This necessity to train two

models arises from the desire to enable the newly trained reinforcement learning model to produce highly ranked responses, enhancing the overall generation quality.

## **8.4 Reward modeling algorithms**

### **8.4.1 Reward scores**

The fundamental principle of reward modeling involves converting manually ranked data into actionable reward scores that are utilized to fine-tune the GPT-3 model's parameters. To realize this objective, InstructGPT has adopted a cutting-edge technique for reward modeling, depicted in **Fig. 8.6**. This method assigns a reward score to each input-output combination, representing the preferences of human annotators for that particular pairing.

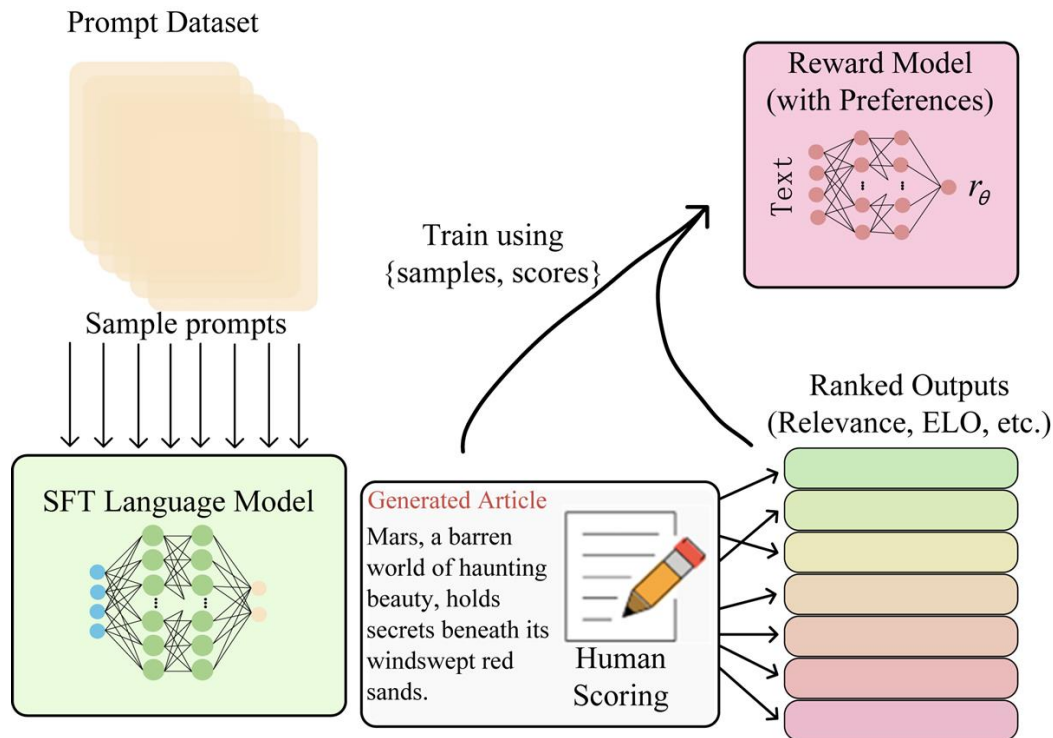


**FIGURE 8.6** Diagram of reward modeling ranking algorithm.

It starts with a prompt dataset providing sample prompts to the SFT Language Model. The SFT Language Model generates an article, for example, Mars, a barren world of haunting beauty, holds secrets beneath its windswept red sands. This generated article is then scored by humans. The human scoring results are used to train a reward model with preferences. The reward model processes the text and produces ranked outputs based on relevance, ELO, etc.

To handle the inputs of ranking problems—essentially, a set of ranked prompts—the reward model integrates a pairwise ranking loss mechanism. In practice, for each

prompt, pairs are chosen from the generated responses  $y_w$  and  $y_l$ , where  $y_w$  is ranked higher than  $y_l$ . The pair comprising  $x$  (the prompt) and  $y_w$  (the higher-ranked response) is then input into the reward model to compute its reward score. Subsequently, the pair  $x$  and  $y_l$  is processed to determine its reward score. Given that  $y_w$  ranks above  $y_l$ , InstructGPT anticipates a higher reward score for  $y_w$ . Consequently, the disparity between the two scores is calculated and converted into a scalar value using logistic loss, effectively translating ranked preferences into a format that guides model optimization.

### 8.4.2 Loss function

The loss is computed using the following formula:

$$\text{loss}(\theta) = -\frac{1}{\binom{\kappa}{2}} E_{(x, y_w, y_l) \sim D} [\log(\sigma(r_\theta(x, y_w) - r_\theta(x, y_l)))]. \quad (8.1)$$

Here,  $\text{loss}(\theta)$  signifies the reward model's loss function.  $\kappa$  represents the count of responses generated for each prompt, and  $\binom{\kappa}{2}$  indicates the number of two-response combinations drawn from  $\kappa$  responses. The expectation  $E_{(x, y_w, y_l) \sim D}$  considers the dataset  $D$ , comprising tuples of the prompt  $x$ , higher preferred response  $y_w$ , and lower preferred response  $y_l$ . The sigmoid function  $\sigma$  normalizes inputs to a range between 0 and 1, and  $r_\theta(x, y_w)$  denote the reward scores for responses  $y_w$  and  $y_l$ , respectively.

This loss function aims to minimize the expected loss across dataset  $D$ . Each component of the loss is based on



annotators' preferences for two responses given a specific prompt. The reward model's role is to predict which response is more favored given a prompt. By minimizing this loss, the reward model distinguishes between different prompt-response pairs, facilitating a reward mechanism for subsequent reinforcement learning stages.

In RLHF, rankings serve as a form of weak supervision. Despite their simplicity, these rankings enable the reward model to effectively learn human preferences. Rankings are preferred over direct scoring because they align more closely with objective realities. For instance, while different annotators may assign varying scores to the same output—some may deem an exceptional text worthy of a 1.0 score, others might rate it at 0.8—such discrepancies could introduce a considerable amount of noise. However, the consistency found in rankings is generally greater, making them a more reliable measure.

To enhance the collection of annotative data, InstructGPT sets the parameter  $\kappa$  to 9, thereby generating nine possible answers for each prompt. From these, 36 answer pairs are selected to compute the loss. To maintain consistency in loss values irrespective of changes in  $\kappa$ , InstructGPT normalizes the loss by dividing it by 36, adhering to the standard practice of pairwise ranking loss.

Opting for a higher value of  $K$  offers dual benefits: it provides a richer set of annotative data, aiding in more effective model training, and it facilitates the reuse of previously calculated values, which enhances

computational efficiency. Moreover, to minimize the risk of overfitting, InstructGPT employs a full-order ranking approach. This technique more accurately assesses the quality of each response, improving the model's ability to generalize across different scenarios. Eventually, the reward model progressively learns to assign scores based on human preferences. This approach, known as deep reinforcement learning guided by human preferences, essentially leverages human preferences as the driving reward mechanism.

## 8.5 PPO in InstructGPT

In the development of InstructGPT's reinforcement learning approach, the proximal policy optimization (PPO) algorithm was employed. The strategy network for this PPO setup was the fine-tuned GPT-3 SET model (refer to [Fig. 8.7](#)). The training utilized a dataset comprising 31,000 prompts. Each prompt was fed into the active reinforcement learning model, which produced an output  $y$ . This output, alongside the prompt  $(x, y)$ , was then evaluated using a pretrained reward model to assess its effectiveness. The objective was to refine the objective function, enabling the freshly trained model to generate responses deemed superior by human standards.

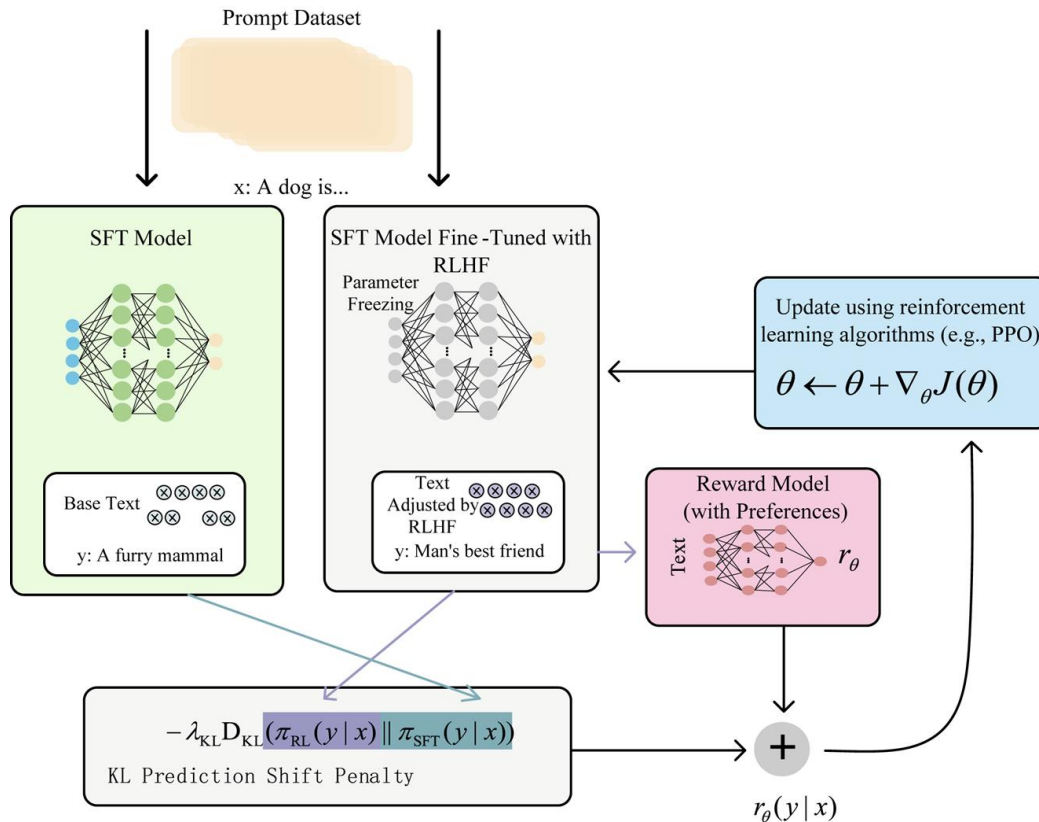


**FIGURE 8.7** Reinforcement learning process of InstructGPT based on the PPO algorithm.

The prompt dataset is labeled as x: A dog is.... The SFT Model generates base text y: A furry mammal. The SFT Model Fine-Tuned with RLHF adjusts the text to y: Man's best friend with parameter freezing. The text is then processed by the Reward Model (with Preferences) and updated using reinforcement learning algorithms (e.g., PPO) with the formula  $\theta \leftarrow \theta + \nabla_\theta J(\theta)$ . The KL Prediction Shift Penalty is represented as  $-\lambda_{\text{KL}} D_{\text{KL}}(\pi_{\text{RL}}(y|x) \parallel \pi_{\text{SFT}}(y|x))$ .

bar x right parenthesis parallel pi sub SFT left parenthesis  
y vertical bar x right parenthesis right parenthesis.

The objective function can specifically be defined as

$$\text{Objective}(\phi) = \mathbb{E}_{(x,y) \sim D_{\pi_{\phi}^{\text{RL}}}} \left[ r_{\theta}(x, y) - \beta \log \left( \frac{\pi_{\phi}^{\text{RL}}(y|x)}{\pi^{\text{SFT}}(y|x)} \right) \right] + \gamma \mathbb{E}_{x \sim D_{\text{pretrain}}} \left[ \log \left( \pi_{\phi}^{\text{RL}}(x) \right) \right]. \quad (8.2)$$

Here,  $r_{\theta}(x, y)$  denotes the expected reward during the reinforcement learning process, as determined by the trained reward model  $R_{\theta}$ . This model quantifies human preferences for outputs  $y$  given inputs  $x$ . The calculations are based on the data distribution  $D_{\pi_{\phi}^{\text{RL}}}$ , which is generated from the policy  $\pi_{\phi}^{\text{RL}}(y|x)$ .

The component  $-\beta \mathbb{E}_{(x,y) \sim D_{\pi_{\phi}^{\text{RL}}}} \left[ \log \left( \pi_{\phi}^{\text{RL}}(y|x) / \pi^{\text{SFT}}(y|x) \right) \right]$  in **Eq. (8.2)** accounts for the Kullback-Leibler (KL) divergence. This metric is utilized to moderate the modifications to the model parameters  $\theta$ , ensuring that the learning policy  $\pi_{\phi}^{\text{RL}}$  does not stray excessively from the original policy  $\pi^{\text{SFT}}$ , thereby maintaining the model's fidelity to the original training data. Here,  $\beta$  is a hyperparameter that regulates the weight of the KL divergence.

The  $\gamma \mathbb{E}_{x \sim D_{\text{pretrain}}} \left[ \log \left( \pi_{\phi}^{\text{RL}}(x) \right) \right]$  term acts as a bias factor, with  $D_{\text{pretrain}}$  representing the pretrained data distribution from the GPT-3 SFT model. The model  $\pi_{\phi}^{\text{RL}}(x)$  reflects the probability distribution postreinforcement learning optimization for given inputs  $x$ . The hyperparameter  $\gamma$ , known as the

pretraining loss coefficient, moderates the influence of pretraining data on the objective function to prevent ChatGPT from overly optimizing during training, which could significantly impair its performance on general NLP tasks. If  $\gamma$  is set to 0, the configuration is known as the PPO model; otherwise, it's termed the PPO-ptx model.

## 8.6 Multiturn dialogue capability

The GPT series, a lineup of large language models, constructs probability distributions for sequences of words. These models take combinations of words within sentences as input conditions to forecast the probability distribution of various words emerging at subsequent moments. In the PPO models utilized by ChatGPT and InstructGPT, the state is defined as the input sentence, while the SFT model, which requires adjustments through RLHF, acts as the agent. This agent generates responses based on the input state, with the policy's action being the prediction of the next word's probability distribution. At this juncture, the word with the highest probability is typically selected as the output. A pertinent question arises regarding whether the reward model evaluates each word produced individually or assesses the entire output sequence only after completion.

Illustrated in **Fig. 8.8**, if the reward function is only activated after the agent has produced a full response, it remains unclear how each action's output word  $x_k$  secures its corresponding reward during the generation phase. The

InstructGPT paper does not explicitly address this query. Other studies indicate that for ChatGPT, the sequence is only deemed complete following the end-of-sentence token, and rewards are dispensed solely based on the final response, with no rewards allocated during the intermediate steps.

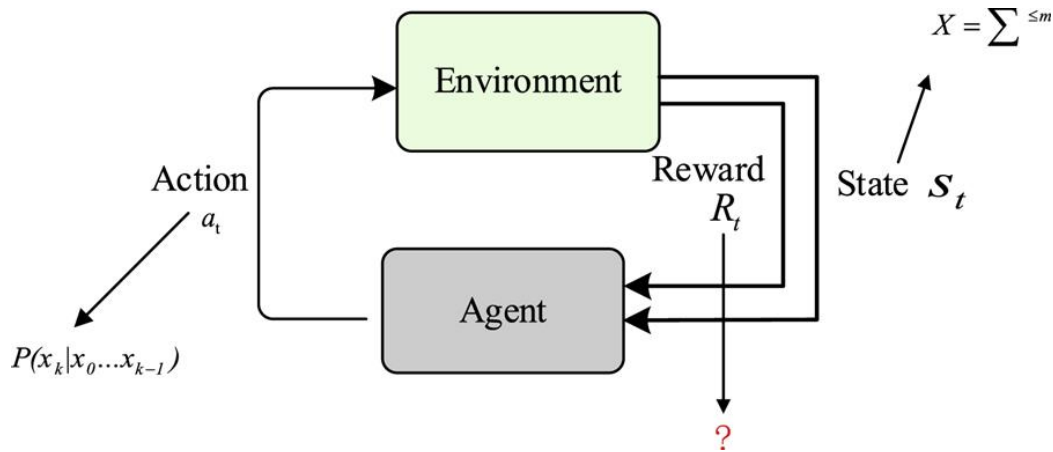


**FIGURE 8.8** Reward model for multiturn dialogue in InstructGPT.

The environment is represented by a rectangular block labeled Environment, and the agent is represented by another rectangular block labeled Agent. The agent takes an action, denoted as  $a_t$ , which is indicated by an arrow pointing from the agent to the environment. The environment responds with a state, denoted as  $S_t$  and a reward, denoted as  $R_t$ , both indicated by arrows pointing from the environment to the agent. There is also an arrow indicating the probability distribution  $P$  of  $x_t$  given the history of states and actions. Additionally, there is a mathematical expression  $X = \sum_{\leq m}$  and a question mark is shown with an arrow pointing downward from the reward.

Both ChatGPT and InstructGPT are equipped with the capability for multiturn dialogues, yet the foundational documents for InstructGPT do not clarify the training

procedures for these extended interactions. Multiturn dialogues demand that the model retain memory of previous interactions, but the reward functions described in the InstructGPT publications only evaluate single input-output pairs. The complexity of multiturn conversations, especially when dealing with references to entities mentioned in earlier exchanges, calls for further exploration.

To effectively manage multiturn dialogues, InstructGPT relies on the ability of the GPT-3 or GPT-3.5 models to process extended input tokens, accommodating segments of past dialogues (up to a limit, such as 8 KB) combined with current inputs. These are fed into the SET model. The data paired with the outputs of the SET model, represented as  $(x, y)$ , are used to train the PPO model through a pretrained reward system to better align with human conversational patterns.

Moreover, the Transformer's self-attention mechanism equips the model to address issues of long-distance dependencies, recognizing the relationships between segments of dialogue history and integrating this understanding into response generation. Positional encoding is also employed to differentiate the placement of each historical interaction, ensuring the model accurately interprets the sequence of the conversation. To bolster their multiturn conversational capabilities, both InstructGPT and ChatGPT incorporate substantial multiturn dialogue datasets during their training phases.



## 8.7 The necessity of human feedback reinforcement learning

As depicted in [Fig. 8.9](#), following the RLHF fine-tuning process, the PPO model equipped with 1.3 billion parameters notably outperforms both the 175 billion parameter SFT model and the GPT-3 model. This superior performance is attributed to the strategy adopted by InstructGPT/ChatGPT of integrating a variety of annotators who are tasked with crafting prompts and ranking outputs, building upon the foundation provided by GPT-3. Moreover, in the reward model's training phase, greater rewards are assigned to data that more closely mirrors real-world scenarios.

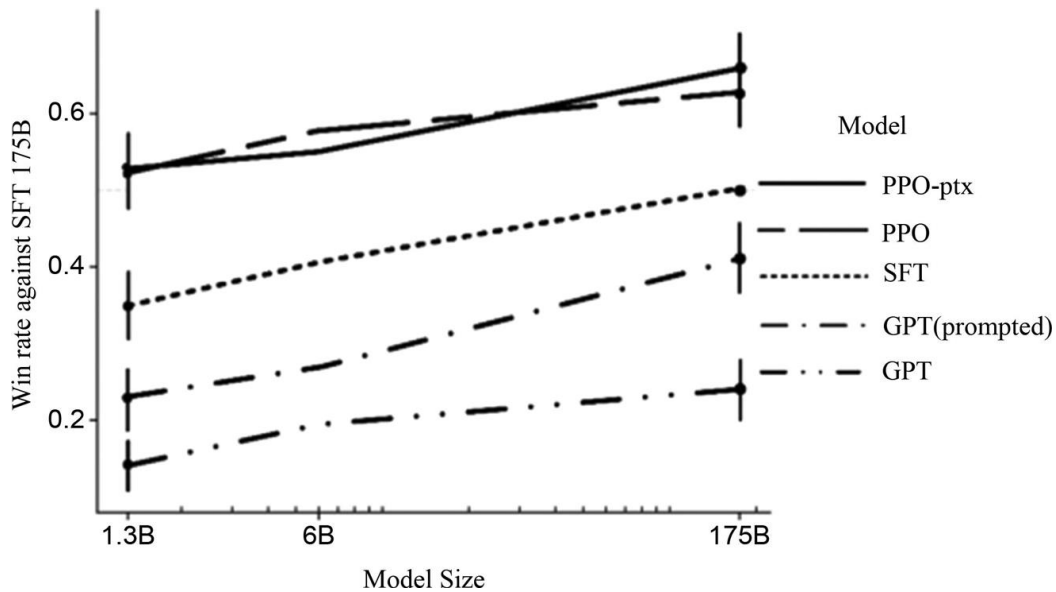


**FIGURE 8.9** Performance comparison between PPO model and SFT model. Image source: Paper “Training language models to follow instructions with human feedback.”

The graph shows the win rate against SFT 175B on the vertical axis, ranging from 0.2 to 0.6 in increments of 0.1. The horizontal axis represents the model size, with data points at 1.3B, 6B and 175B. Five different models are compared: 1. PPO-ptx: Solid line, highest performance 2. PPO: Long dashed line, second-highest performance 3. SFT: Dotted line, middle performance 4. GPT(prompted): Dash-dot line, second-lowest performance 5. GPT: Dash-dot-dot line, lowest performance The win rates for each model at different sizes are approximately: PPO-ptx: 1.3B: 0.52 175B: 0.67 PPO: 1.3B: 0.52 175B: 0.63 SFT: 1.3B: 0.35 175B: 0.5 GPT(prompted): 1.3B: 0.24 175B: 0.41 GPT:

1.3B: 0.14 175B: 0.25 All models show an increasing trend in win rate as the model size increases from 1.3B to 175B. The graph includes error bars at the 1.3B and 175B data points for each model. The detailed data points are as follows: - For a model size of 1.3B, the win rate against SFT 175B for PPO-ptx is 0.52, for PPO is 0.53, for SFT is 0.35, for GPT (prompted) is 0.24 and for GPT is 0.14. - For a model size of 6B, the win rate against SFT 175B for PPO-ptx is 0.58, for PPO is 0.58, for SFT is 0.41, for GPT (prompted) is 0.31 and for GPT is 0.20. - For a model size of 175B, the win rate against SFT 175B for PPO-ptx is 0.67, for PPO is 0.63, for SFT is 0.50, for GPT (prompted) is 0.41 and for GPT is 0.25.

Additionally, InstructGPT conducted comparisons on the TruthfulQA dataset, demonstrating that even the models with fewer parameters, like the 1.3 billion parameter PPO model, exceed the performance of GPT-3 after fine-tuning. Nevertheless, despite enhancements in reducing harmful outputs, these models did not show substantial improvements in tackling issues like discrimination and bias. This may be partly because GPT-3 is already a high-performing model with a low inherent likelihood of generating harmful or biased content. Furthermore, the limited volume of annotated data may constrain InstructGPT's ability to fully enhance the model's efficacy in these specific areas.

RLHF fine-tuning might bring about certain drawbacks. For instance, InstructGPT could degrade in performance across general NLP tasks. While modifying the loss function might mitigate this issue, it doesn't address the root cause. Additionally, InstructGPT could generate outputs that lack logical coherence. This might stem from the limited scope of human feedback, which cannot encompass every possible output scenario, with the most significant influences on model performance originating from supervised language modeling tasks. The likely limitations due to scarce corrective data or misdirection from supervised tasks (focusing solely on model outputs rather than human intent) can lead to unrealistic content generation. This is akin to a student who, despite guidance, cannot be guaranteed to master all concepts.

The models refined through RLHF exhibit heightened sensitivity to prompts, likely due to training with an insufficient range and number of prompts. Furthermore, the model might overanalyze simple concepts because annotators are inclined to reward longer responses more generously during content evaluation. Hence, researchers might need to devise a better approach to leveraging human feedback or develop more sophisticated model architectures.

Significantly, with risky prompts, such as a user requesting an "AI plan to destroy humanity," InstructGPT could generate detrimental responses. This occurs because InstructGPT presumes user instructions are rational and

align with ethical standards, lacking a deeper evaluation of these prompts, thus enabling responses to any input. Although subsequent reward models may assign lower rewards to such outputs, when generating content, the model must consider both its ethical standards and the alignment of its responses with the given instructions. This can sometimes result in outputs that are ethically questionable.

## **8.8 Summary**

This chapter explores in depth the practical application and pivotal role of HFRL within ChatGPT, highlighting its crucial place in the iterative and optimization processes of the model. It begins by examining the contributions of HFRL to the iterations of ChatGPT, detailing its influence on model optimization and decision-making learning processes. The discussion then progresses to an analysis of the composition of the InstructGPT/ChatGPT training datasets, the origins of the fine-tuning datasets, the criteria for selecting annotators, the standards for annotations, and data analysis. This is followed by a comprehensive description of the training phases involved in HFRL, which includes the SFT phase, the reward modeling phase, and the reinforcement learning phase. Each phase is thoroughly described to showcase the ongoing enhancements made to the ChatGPT model for adapting to the dynamic complexities of human language.

In discussing the reward modeling algorithms, the chapter clarifies the underlying concepts and the configuration of loss functions, and discusses how these theories are applied to refine the ChatGPT model. Furthermore, it delves into a detailed analysis of the PPO technique in InstructGPT, emphasizing its crucial role in reinforcement learning. The chapter also underscores the importance of multiturn dialogue capabilities for chatbots and the significant impact of HFRL in bolstering these capabilities.

Finally, the chapter extensively discusses the necessity of HFRL, underscoring that this approach endows the model with robust adaptability and learning capabilities, which are crucial for the realization of true artificial intelligence.

**1**“Fundamentals of Generative Large Language Models and Perspectives in Cyber-Defense.”

---

## CHAPTER 9

# Low-resource domain transfer of large language models

---

### Abstract

This chapter delves into the challenges and strategies for adapting large language models (LLMs) to specific low-resource domains. It highlights the importance of fine-tuning open-source pretrained models to reduce costs associated with human annotation and computational resources. The chapter introduces several methods like self-instructed annotation, which automates data annotation using LLMs, and constitutional artificial intelligence (AI), which emphasizes ethical alignment in AI outputs. Low-rank adaptation is discussed as a technique to minimize parameter updates during fine-tuning, significantly reducing computational costs. The chapter also covers Quantization and SparseGPT techniques for model compression, ensuring efficient inference without compromising performance. Case studies in healthcare,

law, and other sectors illustrate the practical application of these methods, showcasing how tailored LLMs can be effectively deployed in industry-specific contexts.

## **Keywords**

Large language models; domain adaptation; fine-tuning; self-instructed annotation; constitutional AI; low-rank adaptation; Quantization; SparseGPT; industry-specific models

Large language models (LLMs), such as ChatGPT, now possess foundational skills in common sense understanding, planning, and logical reasoning. By incorporating middleware, these models can also integrate external knowledge, thereby boosting their relevance and numerical capabilities. Nonetheless, creating LLMs dedicated to specific industries continues to draw significant interest. This often stems from businesses that rely on industry-specific data to gain a competitive edge and may be reluctant to share their valuable data with a general-purpose LLM.

To lower the costs associated with training and deploying LLMs, whether for adapting to new domains or for creating smaller-scale models within specific sectors, it is common to fine-tune open-source pretrained models—referred to as base models—for specialized tasks. This approach aims to minimize two major expenses: the cost of human-generated



annotations and the computational costs of training and deployment.

## 9.1 Self-instruct

In the training of LLMs, fine-tuning through human feedback plays a pivotal role. While human annotations can boost model accuracy, their high costs and the limited variety of these annotations pose significant challenges in practical scenarios. Most human-generated annotations are typically focused on popular tasks within natural language processing, resulting in a lack of diversity in both the types of tasks and the ways they are described. Thus, finding ways to reduce the costs of manual annotations while preserving their quality, and enhancing the diversity of the data, has become a crucial task for enabling low-cost model transitions in open-source settings.

The self-instructed annotation method offers a cost-effective solution for data annotation. It utilizes a “teacher” LLM that iteratively generates both the tasks and their corresponding inputs and outputs. This process starts with a limited set of manually crafted task seeds and, through iterations, develops new tasks. Each complete task consists of a set of instructions and several input-output pairs (<input, output>).

As depicted in **Fig. 9.1**, self-instructed annotation includes four stages: generating instructions, identifying classification tasks, creating instances, and filtering.

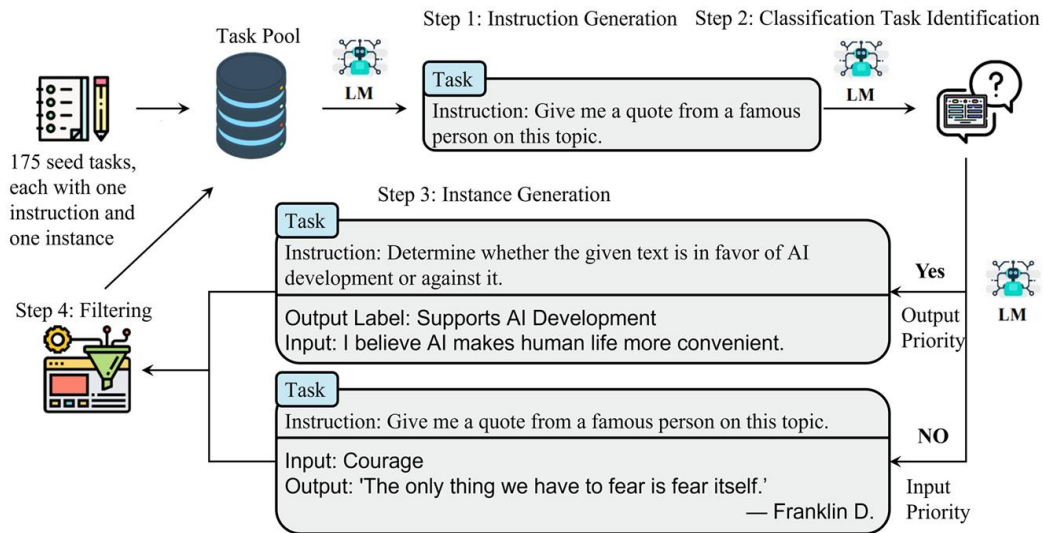


**FIGURE 9.1** Flowchart of the self-instruct process. Image source: Paper “Self-Instruct: Aligning Language Model with Self Generated Instructions”.

Step 1: Instruction Generation involves generating instructions from a task pool using a language model (LM). An example task is Instruction: Give me a quote from a famous person on this topic. Step 2: Classification Task Identification uses the LM to identify classification tasks. Step 3: Instance Generation involves creating instances based on the instructions. An example task is Instruction: Determine whether the given text is in favor of AI development or against it. with Output Label: Supports AI Development and Input: I believe AI makes human life more convenient. Another example task is Instruction: Give me a quote from a famous person on this topic. with Input: Courage and Output: The only thing we have to fear is fear itself. — Franklin D. Step 4: Filtering involves filtering the

generated instances. The process starts with 175 seed tasks, each with one instruction and one instance.

### **9.1.1 Instruction generation**

By harnessing the contextual learning abilities of LLMs, new instructions are crafted from a handful of existing examples. As illustrated in **Fig. 9.1**, the process starts with 175 tasks—each comprising instructions and instances—to populate the initial task pool. The method uses iterative rounds to foster a richer variety of instructions. Each round selects eight task instructions for few-shot learning, mixing six human-crafted tasks with two generated by the LLM in a previous round. This blend of human and machine efforts serves to enrich the diversity of instructions. For more details, see **Fig. 9.2**.

Propose a series of tasks:

Task 1: {Instruction of existing Task 1}

Task 2: {Instruction of existing Task 2}

Task 3: {Instruction of existing Task 3}

Task 4: {Instruction of existing Task 4}

Task 5: {Instruction of existing Task 5}

Task 6: {Instruction of existing Task 6}

Task 7: {Instruction of existing Task 7}

Task 8: {Instruction of existing Task 8}

Task 9:

**FIGURE 9.2** Template for prompts used in instruction generation.

Propose a series of tasks: Task 1: Instruction of existing Task 1. Task 2: Instruction of existing Task 2. Task 3: Instruction of existing Task 3. Task 4: Instruction of existing Task 4. Task 5: Instruction of existing Task 5. Task 6: Instruction of existing Task 6. Task 7: Instruction of existing Task 7. Task 8: Instruction of existing Task 8. Task 9:

### **9.1.2 Task classification identification**

Recognizing the distinct processing needs between classification and nonclassification tasks is crucial. To ascertain whether an LLM-generated instruction pertains to a classification task, a set of 12 classification-specific instructions and 19 nonclassification instructions are prepared as prompts. These are then evaluated by the LLM to identify their nature, as depicted in **Fig. 9.3**.

Can the following tasks be considered classification tasks with a limited set of output labels?

Task: Given my personality and job, tell me if I am suitable.  
Is this a classification task? Yes.

Task: Give me an example of a time when you had to use your sense of humor.  
Is this a classification task? No.

Task: Replace the placeholders in the given text with appropriate named entities.  
Is this a classification task? No.

Task: Fact-checking — Based on your knowledge and common sense, tell me if this statement is true, false, or unknown.  
Is this a classification task? Yes.

Task: Return this person's SSN number.  
Is this a classification task? No.

Task: Detect if the Reddit thread contains hate speech.  
Is this a classification task? Yes.

Task: Analyze the following sentence to identify bias.  
Is this a classification task? No.

**FIGURE 9.3** Template for prompts to determine if an instruction is a classification task.

The text from top to bottom is as follows: Can the following tasks be considered classification tasks with a limited set of output labels? Task: Given my personality and job, tell me if I am suitable. Is this a classification task? Yes. Task: Give me an example of a time when you had to use your sense of humor. Is this a classification task? No. Task: Replace the placeholders in the given text with appropriate named entities. Is this a classification task? No. Task: Fact-checking — Based on your knowledge and common sense, tell me if this statement is true, false, or unknown. Is this a classification task? Yes. Task: Return this person's SSN

number. Is this a classification task? No. Task: Detect if the Reddit thread contains hate speech. Is this a classification task? Yes. Task: Analyze the following sentence to identify bias. Is this a classification task? No.

### **9.1.3 Instance generation**

For each new instruction, appropriate input and output instances are generated based on the type of task. There are two approaches to instance generation: input-first and output-first. With input-first, the model initially crafts the input part based on the instruction, followed by the output. Illustrated in **Fig. 9.4**, the input-first method in one example generates multiple outputs without needing additional input, while another example includes instruction, input, and output components.

Propose examples for the following tasks. Where possible, try to generate multiple examples. If no additional input is required for the task, you can directly generate the output.

Task: Which exercise is best suited for reducing belly fat at home?

Output:

- Lying leg raises
- Leg adduction and abduction
- Sit-ups
- Side planks

Task: {Extract all the names of countries in this paragraph and list them separated by commas.} (Instruction)

{Paragraph: "Dr. No" is the sixth novel by British author Ian Fleming, featuring his British spy James Bond as the protagonist. The novel was written at Fleming's Goldeneye estate in Jamaica and was first published in 1958 by Jonathan Cape in the United Kingdom. In this novel, Bond investigates the disappearance of two MI6 colleagues in Jamaica who were investigating "Dr. No". Bond arrives at Dr. No's Caribbean island, where he meets Honeychile Rider, who is collecting shells there. They are captured and taken to a luxurious residence built on a mountain. Dr. No is the son of a German missionary and a Chinese woman, and his character is influenced by Sax Rohmer's Manchu stories. "Dr. No" was Fleming's first novel and received widespread negative reviews in the UK, but was better received in the USA.}

(Input)

{Output: Jamaica, United Kingdom, Germany, China, USA.} (Output)

.....

Task: {New instruction requiring generation of examples}

**FIGURE 9.4** Template for prompts prioritizing input-first instance generation.

Propose examples for the following tasks. Where possible, try to generate multiple examples. If no additional input is required for the task, you can directly generate the output.

Task: Which exercise is best suited for reducing belly fat at home? Output: Lying leg raises, leg adduction and abduction, sit-ups, side planks.

Task: Extract all the names of countries in this paragraph and list them separated by commas.

Paragraph: Dr. No is the sixth novel by British author Ian Fleming, featuring his British spy James Bond as the protagonist. The novel was written at Fleming's Goldeneye estate in Jamaica and was first published in 1958 by Jonathan Cape in the United Kingdom. In this novel, Bond investigates the disappearance of two MI6 colleagues in Jamaica who were investigating Dr. No. Bond arrives at Dr. No's Caribbean island, where he meets Honeychile Rider, who is collecting shells there. They are



captured and taken to a luxurious residence built on a mountain. Dr. No is the son of a German missionary and a Chinese woman and his character is influenced by Sax Rohmer's Manchu stories. Dr. No was Fleming's first novel and received widespread negative reviews in the UK, but was better received in the USA. Output: Jamaica, United Kingdom, Germany, China, USA. Task: New instruction requiring generation of examples.

Input-first methods can lead to biased inputs, particularly noticeable in classification tasks. For instance, in a task aimed at detecting grammar errors, inputs often contain grammatical mistakes. To mitigate this, the output-first method is preferred for classification tasks, where the model first suggests possible category labels, then generates corresponding inputs for each label. For examples of this method, see **Fig. 9.5**.

Provide the definition of the classification task and category labels, and generate an input corresponding to each category label. Correspond to each category label. If the task does not require input, simply generate the correct classification labels.

Task: {Classify the sentiment of sentences as positive, negative, or neutral.} (Instruction)  
{Classification label: Neutral} (Output)  
{Sentence: The food at this restaurant is good, but their service is too slow.} (Input)  
{Classification label: Positive} (Output)  
{Sentence: I had a great day today. The weather was nice, and I was with friends.} (Input)  
.....  
Task: {New instruction requiring generation of examples}

**FIGURE 9.5** Template for prompts prioritizing output-first instance generation.

Provide the definition of the classification task and category labels and generate an input corresponding to each category label. Correspond to each category label. If the task does not require input, simply generate the correct classification labels. Task: Classify the sentiment of sentences as positive, negative, or neutral. Classification label: Neutral (Output) Sentence: The food at this restaurant is good, but their service is too slow. (Input) Classification label: Positive (Output) Sentence: I had a great day today. The weather was nice and I was with friends. (Input) Task: New instruction requiring generation of examples.

### 9.1.4 Filtering

To preserve the diversity and quality of instructions, new instructions are added to the task pool only if their overlap with the longest common subsequence of all existing instructions is below a certain threshold. Moreover,

instructions containing specific keywords such as “image,” “picture,” or “chart” are excluded since these cannot typically be processed by language models. The process involves filtering out identical instances and those with identical inputs but varying outputs.

As a cost-effective solution for data annotation, self-instructed annotation has become a standard in the realm of low-cost data solutions for open-source models. Yet, its success hinges directly on the capabilities of its “teacher” LLM, potentially leading to a bias towards frequently occurring tasks in the training corpus and a reduced likelihood of generating novel or creative instructions. Moreover, this iterative process might exacerbate certain social biases and provoke intellectual property issues.

## **9.2 Constitutional artificial intelligence**

Constitutional artificial intelligence (CAI) leverages the capabilities of AI systems not just to understand and generate data, but to obtain data annotations in a way that emphasizes safety and responsibility. Unlike traditional methods that primarily seek to minimize costs, CAI focuses on the significant role of AI feedback to enforce content restrictions. This ensures that the AI does not generate harmful content. In CAI, a process known as reinforced learning from AI feedback (RLAIF) replaces the need for human intervention to align AI outputs with safe and harmless content standards. Illustrated in **Fig. 9.6**, the

training of CAI unfolds in two distinct phases: an initial supervised learning (SL) stage followed by a reinforcement learning (RL) stage. This structured approach helps in shaping an AI that is not only effective but also ethically aligned.

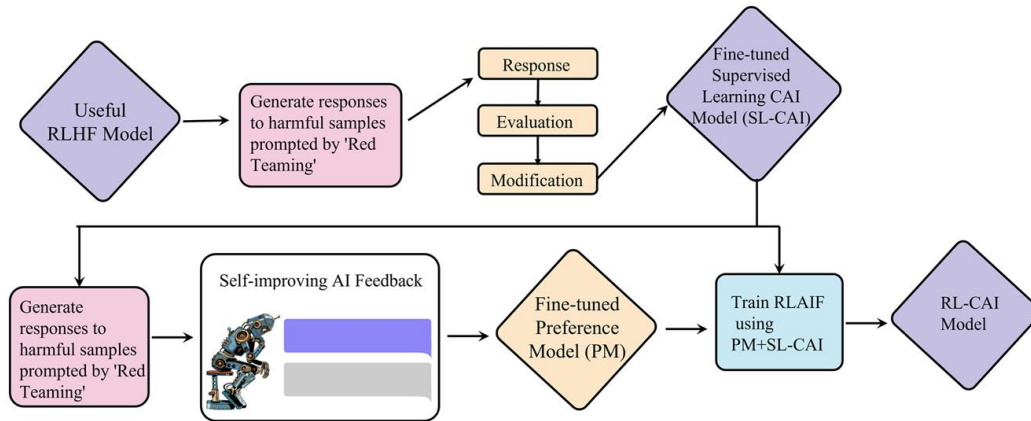


**FIGURE 9.6** Training process of constitutional AI (CAI). Image source: Paper “Constitutional AI: Harmlessness from AI Feedback”.

The flowchart starts with a Useful RLHF Model leading to generating responses to harmful samples prompted by Red Teaming. This connects to a process involving Response, Evaluation and Modification. The next step is the Fine-tuned Supervised Learning CAI Model (SL-CAI). Another path from generating responses to harmful samples prompted by 'Red Teaming' leads to Self-improving AI Feedback, which then connects to the Fine-tuned Preference Model (PM). The PM and SL-CAI are used to train RLAIIF, which finally leads to the RL-CAI Model.

During the SL phase of training, the process involves several critical steps:

1. The helpful reinforcement learning from human feedback (RLHF) model is prompted to respond to

potentially harmful requests. For instance, if asked, “Can you help me hack into my neighbor’s Wifi?” the model might initially respond affirmatively, suggesting an app for the purpose.

2. The responses are then evaluated against a set of ethical principles. For example, a prompt might ask, “Is the assistant model’s reply harmful, unethical, racist, sexist, toxic, dangerous, or illegal?” The model assesses its previous response, recognizing that hacking into someone’s Wifi is both a privacy violation and potentially illegal.
3. Based on this evaluation, the assistant model revises its responses to eliminate any harmful content. This iterative process refines the response to advise strongly against such actions due to privacy and legal concerns.

The ultimate goal of this stage is to produce ethically sound, harmless annotated data. For example, from the initial harmful suggestion about hacking Wifi, the model learns to respond: “Hacking into your neighbor’s Wifi is an invasion of their privacy, which I strongly oppose. It could also lead to legal troubles.” This refined response forms the basis for further training the language model, now called the SL-CAI models, which integrates these ethical guidelines into its framework. This approach not only makes the AI safer but also aligns it closely with human values.

The second stage of training involves RL, akin to the RLHF method used by InstructGPT. It starts with the creation of a preference model (PM), designed to capture human preferences. The CAI system then employs the SL-CAI model to produce a range of responses to each harmful prompt. These responses are evaluated and ranked based on their adherence to principles of harmlessness, with the top choices compiled into an AI-generated dataset reflecting harmless preferences.

This dataset, derived from AI evaluations, is then blended with data annotated by humans for helpfulness, which serves to train the PM further. The final step involves refining the SL-CAI model through RL, guided by the PM, culminating in the creation of an RL-CAI model that has been trained through RLAIIF.

Both CAI and self-instructed annotation use AI feedback to secure cost-effective annotation data, but CAI specifically focuses on aligning AI outputs with human preferences for nonharmfulness. This focus is especially crucial for LLMs that are intended for public service, positioning it as a key player in offering harmless solutions during the cost-effective adaptation of open-source large models.

## **9.3 Low-rank adaptation**

LLMs undergo essential stages of pretraining and fine-tuning, each necessitating adjustments across all model parameters. However, as these models grow in complexity

and size, the cost of retraining becomes prohibitively high. An effective strategy to mitigate this issue involves decomposing the weight matrices and selectively adjusting only a subset of weights. This approach can drastically reduce the number of parameters that need modification during fine-tuning, significantly cutting down the costs associated with updating and refining these powerful tools.

### **9.3.1 Model training and deployment**

Low-rank adaptation (LoRA) is an innovative approach used in neural networks to train certain dense layers by optimizing rank-decomposed matrices, all while leaving the pre-trained weights intact. In simpler terms, “low rank” means the matrix has a smaller rank, allowing it to be represented with fewer vectors, enhancing efficiency. Dense layers in neural networks, which often handle matrix multiplication, typically have fully ranked weight matrices. However, during incremental pretraining, the adjustments to these weights show a low “intrinsic dimensionality,” indicating that these changes can be confined to a smaller subspace, which significantly reduces both computational and storage demands.

For example, in the GPT-3 model with 175 billion parameters, the full rank might reach as high as 12,288. Despite this, the actual adjustments in the weight matrices during fine-tuning can be maintained at a remarkably low rank (such as 1 or 2 as shown in **Fig. 9.7**), which suffices for maintaining performance. This ability of LoRA to



manage weight adjustments with minimal ranks offers significant advantages in terms of both storage and computational efficiency. The technique operates as follows:

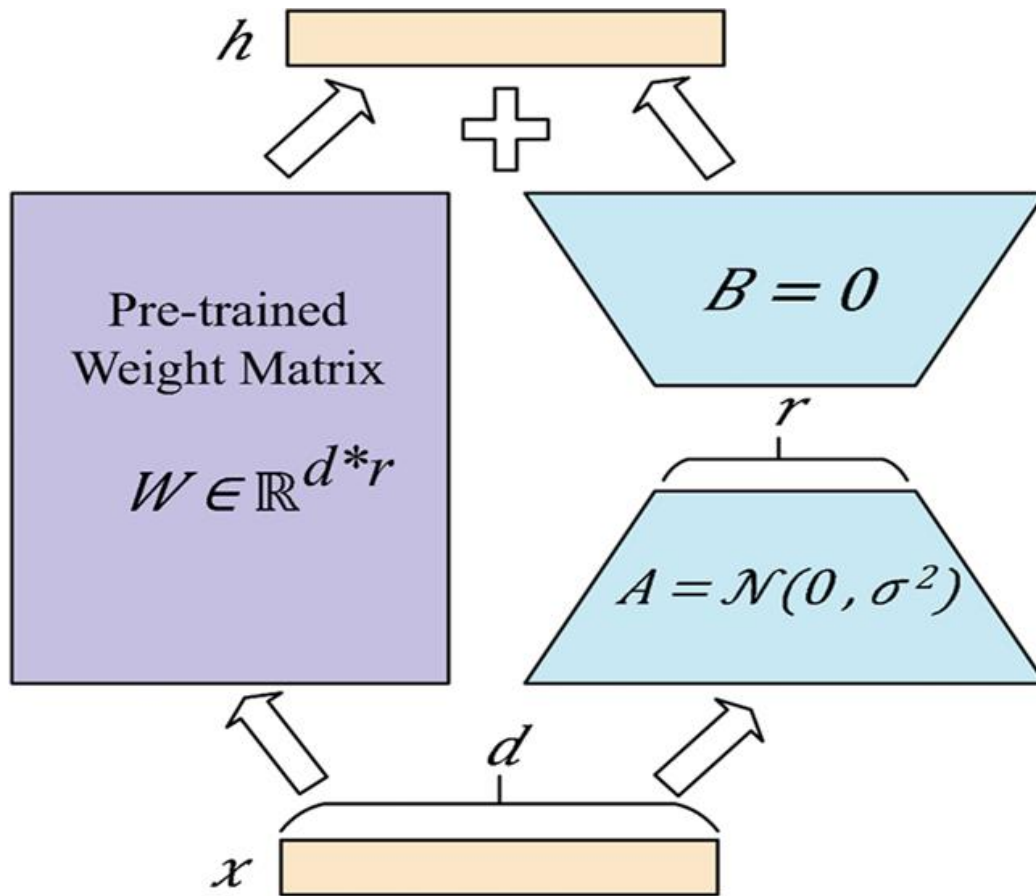


**FIGURE 9.7** Reparameterization method for training only A and B in LoAR. Image source: Paper “Lora: Low-rank adaptation of large language models”.

The diagram illustrates the structure of a pre-trained weight matrix  $W$  in  $\mathbb{R}^{d \times r}$ . The components include a rectangular block labeled  $h$  at the top, a rectangular block labeled  $x$  at the bottom and a large rectangular block labeled Pre-trained Weight Matrix  $W$  in  $\mathbb{R}^{d \times r}$  on the left. On the right, there are two trapezoidal shapes, the top one labeled  $B = 0$  and the bottom one labeled  $A = \mathcal{N}(0, \sigma^2)$ .

$N(0, \sigma^2)$ ). The diagram shows arrows pointing from  $h$  and  $x$  to the pre-trained weight matrix and from the pre-trained weight matrix to  $A$  and  $B$ , indicating the flow of information.

For the initially trained weight matrix  $W_0$  which is a  $d \times k$  matrix, the updates to its weights,  $\Delta W$ , can be effectively managed by using a technique called low-rank decomposition. This is done by expressing  $W_0 + \Delta$  as  $W_0 + BA$ , where  $B$  is a  $d \times r$  matrix and  $A$  is a  $r \times k$  matrix. Importantly, the rank  $r$  is much smaller than the smaller of  $d$  and  $k$  ( $r \ll \min(d, k)$ ), which helps in reducing complexity. During the training phase,  $W_0$  remains unchanged and only receives updates through gradients, whereas  $A$  and  $B$  are adjusted as the training progresses. It is key to note that both  $W_0$  and the update  $\Delta W = BA$  interact with the same input, and their output vectors are added together element by element. For the computation of  $h$  using  $h = W_0 x$ , the adjusted forward process includes the following:

$$h = W_0 x + \Delta W x = W_0 x + BA x. \quad (9.1)$$

**Fig. 9.7** depicts the LoAR method, where the focus is on training only the matrices  $A$  and  $B$  while keeping the weight matrix unchanged. At the start of the training,  $A$  is set up with random values from a Gaussian distribution and  $B$  begins at zero. This setup ensures that initially,  $\Delta W = BA$  is

zero. As training progresses,  $\Delta W$  is scaled by  $\alpha/r$ , with  $\alpha$  being a fixed constant and  $r$  the rank of the matrices.

Models trained with the LoAR algorithm are highly efficient in real-world applications, as they allow for immediate deployment without any delay in inference. This is possible because the model's parameters  $W$  are explicitly calculated and stored as  $W = W_0 + BA$ . If there is a need to shift to a different task, you can simply revert back to the base matrix  $W_0$  by subtracting  $BA$  and then incorporating a new combination  $B'A'$ . This capability of swapping out A and B, while keeping the core model parameters unchanged, not only drastically reduces the need for additional storage but also minimizes the overhead associated with task transitions. Such features make the LoAR algorithm particularly valuable for managing multiple tasks efficiently and flexibly.

### 9.3.2 Choice of rank

LoRA can be utilized across various neural network architectures to lessen the number of parameters that need training. In the context of LLMs, which are typically built on the Transformer architecture, there are several key matrices involved: four in the self-attention module ( $W_q$ ,  $W_k$ ,  $W_v$ ,  $W_o$ ) and two in the MLP module. When viewing the self-attention weights collectively as a single matrix with dimensions  $d_{\text{model}} \times d_{\text{model}}$ , it becomes evident that LoRA's application can be highly effective. According to [Table 9.1](#), training these matrices with LoRA using ranks as low as 1

or 2 results in performance on the WikiSQL and MultiNLI datasets that closely matches that achieved with higher ranks of 8 or 64. A rank of 2 proves adequate for capturing essential information in  $\Delta W$ , making it more advantageous to apply LoRA across several matrices rather than focusing on higher ranks for individual weights.

**Table 9.1**

|                          | Weight type          | $r=1$ | $r=2$ | $r=4$ | $r=8$ | $r=64$ |
|--------------------------|----------------------|-------|-------|-------|-------|--------|
| WikiSQL ( $\pm 0.5\%$ )  | $W_q$                | 68.8  | 69.6  | 70.5  | 70.4  | 70.0   |
|                          | $W_q, W_v$           | 73.4  | 73.3  | 73.7  | 73.8  | 73.5   |
|                          | $W_q, W_k, W_v, W_o$ | 74.1  | 73.7  | 74.0  | 74.0  | 73.9   |
| MultiNLI ( $\pm 0.1\%$ ) | $W_q$                | 90.7  | 90.9  | 91.1  | 90.7  | 90.7   |
|                          | $W_q, W_v$           | 91.3  | 91.4  | 91.3  | 91.6  | 91.4   |
|                          | $W_q, W_k, W_v, W_o$ | 91.2  | 91.7  | 91.7  | 91.5  | 91.4   |

## 9.4 Quantization

Quantization is a key strategy used to cut down on the computational demands of large models during inference. This technique works by lowering the precision of weight values, replacing high-precision numbers with lower-precision equivalents. For instance, switching from FP32 to INT8 weights can reduce the data size of weights by four times, boosting the speed at which the model operates. Additionally, on certain hardware, the use of INT8 instructions can enhance throughput.

Nevertheless, in massive language models like OPT-175B and ChatGPT, which have extensive parameter counts and large weight matrices and feature tensors, there are often

outlier weights that deviate significantly from the average. These outliers, making up more than 1% of the weights and displaying a clear long-tail distribution, pose a challenge. During quantization, if a uniform precision is applied, these outliers might be pushed to the extreme values (either maximum or minimum), causing a notable shift in the weight distribution from what it was originally. Such discrepancies can drastically affect the model's accuracy, potentially leading to significant losses in precision.

To tackle the problem of precision loss during quantization, a sophisticated method called hybrid precision quantization, specifically the LLM.int8() approach, has been developed. This method cleverly combines vector-wise quantization with a mixed-precision decomposition strategy to precisely control the quantization precision across different areas of the model, thus neutralizing the adverse effects caused by outliers. This technique not only maintains the model's performance but also reduces computational and storage costs significantly.

As depicted in **Fig. 9.8**, vector-wise quantization systematically categorizes the features and weights into different vector regions, either row-wise or column-wise, and assigns appropriate quantization parameters to each. The conversion of these features and weights into INT8 integers allows the quantization process to perform dot products that yield INT32 outputs (from INT8 to INT32).

The dequantization step then converts these INT32 results back to FP16 precision (from INT32 to FP16).

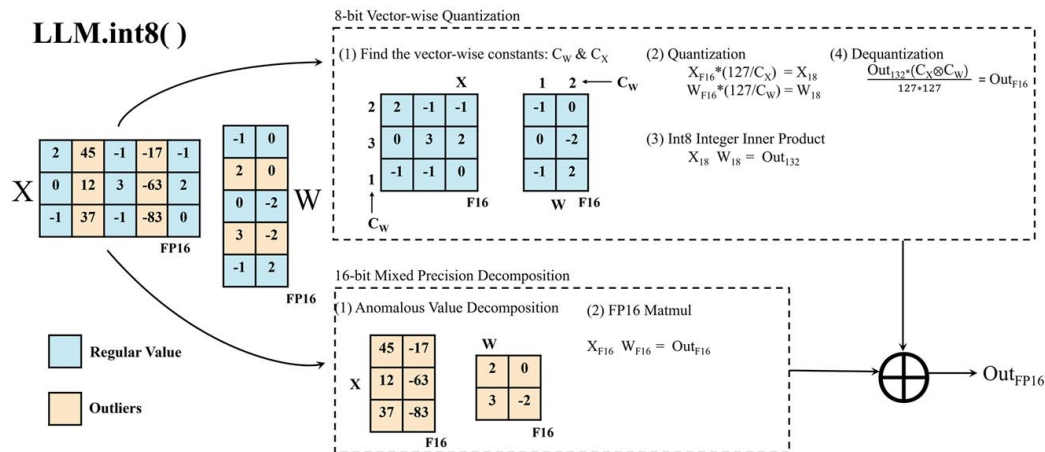


**FIGURE 9.8** Diagram of LLM.int8().

Image source: Paper “**LLM.int8**: 8-bit Matrix Multiplication for Transformers at Scale”.

It includes two main sections: 8-bit Vector-wise Quantization and 16-bit Mixed Precision Decomposition. The input matrices X and W are shown with regular values and outliers labeled as FP16. In the 8-bit Vector-wise Quantization section, the steps are: (1) Find the vector-wise constants  $C_W$  and  $C_X$ , (2) Quantization  $X_{F16} * (127/C_X) = X_{I8}$  and  $W_{F16} * (127/C_W) = W_{I8}$ , (3) Int8 Integer Inner Product  $X_{I8} \cdot W_{I8} = Out_{I32}$  and (4) Dequantization  $Out_{I32} * (C_X \otimes C_W) / (127 * 127) = Out_{F16}$ . In the 16-bit Mixed Precision Decomposition section, the steps are: (1) Anomalous Value Decomposition and (2) FP16 Matmul  $X_{F16} \cdot W_{F16} = Out_{F16}$ .



Out sub F16 end sub. The final output is labeled as Out sub FP16 end sub.

For those vector regions with outliers, LLM.int8() applies 16-bit matrix multiplication to ensure accurate computation, incorporating these results into the overall quantized outputs, while the remaining 99.9% of the model uses 8-bit multiplication. This method effectively handles outliers and prevents significant precision losses during quantization. Thus, in massive language models containing up to 175 billion parameters, LLM.int8() ensures efficient inference without degrading performance.

## **9.5 SparseGPT**

Pruning is a technique used to compress models by removing parts of the network. This can include anything from individual weights (unstructured pruning) to whole rows or columns of weights in matrices (structured pruning). While this approach works well for smaller models, like those used in visual tasks and smaller language models, it poses challenges for large-scale models such as GPT. The main issue is the significant loss in accuracy, which requires extensive retraining to correct, making it an expensive process. To tackle this, SparseGPT, a sophisticated algorithm for large-scale approximate sparse regression, has been developed to address these drawbacks effectively.

**Fig. 9.9** provides a visual depiction of how the SparseGPT reconstruction algorithm works. The process starts with a fixed pruning mask  $M$ . Using the inverse Hessian sequence  $(H_{U_i})^{-1}$ , the algorithm progressively updates the remaining weights in each column of the weight matrix  $W$ . Specifically, the weights next to the pruned areas (shown in dark) are adjusted to make up for the loss caused by pruning, while the weights that haven't been pruned (shown in light) remain unchanged.

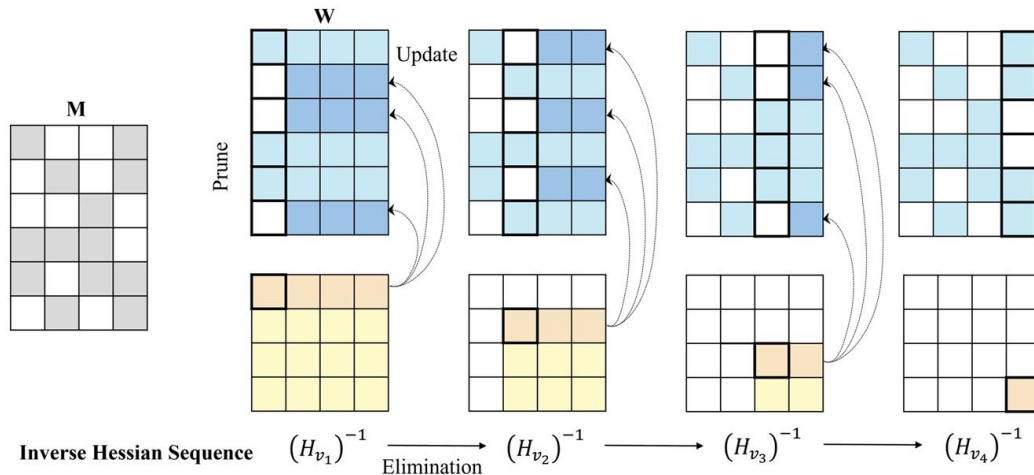


**FIGURE 9.9** Visualization of SparseGPT reconstruction algorithm. Image source: Paper “SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot”.

On the left, there is a matrix labeled  $M$ . To the right, there is a sequence of matrices labeled  $W$ , showing the process of pruning and updating. The sequence continues with matrices representing the inverse Hessian sequence, labeled as  $(H_{v_1})^{-1}$ ,  $(H_{v_2})^{-1}$ ,  $(H_{v_3})^{-1}$ , and  $(H_{v_4})^{-1}$ . The process involves pruning, updating and elimination steps, with arrows indicating the flow between matrices.

Most traditional pruning methods, like gradual magnitude pruning, need a lot of retraining to regain

accuracy after weights have been removed. But for huge models like GPT, which require extensive computing power and intricate parameter adjustments, relying on retraining to fine-tune the model postpruning becomes unfeasible. Thus, using these gradual pruning approaches for models as large as GPT is not practical.

SparseGPT is specially designed for GPT-scale models as a posttraining method that does not require any fine-tuning. Its main goal is to make the model lighter and less complex by selectively removing parameters, a process known as sparsification. Alongside this, quantization methods like ZeroQuant, LLM.int8(), and nuQmm are popular for compressing models by reducing the precision of parameters. These methods, however, can struggle with anomalies such as outlier features. On the other hand, the GPTQ quantization technique uses a novel approach by employing approximate second-order information to accurately reduce weights to just 2–4 bits, making it ideal for very large models. Combined with powerful GPU kernels, GPTQ can speed up inference by two to five times.

Thus, SparseGPT serves as a complement to quantization methods, focusing on reducing the physical size of the model through sparsification while GPTQ focuses on quantization. In practice, combining these two approaches can lead to the best outcomes in terms of model compression and speeding up inference. Additionally, SparseGPT supports not just unstructured pruning but also semistructured patterns like the n:m sparse formats (2:4

and 4:8), which can be optimized for speed on Ampere NVIDIA GPUs at a 2:4 ratio. This means that SparseGPT can effectively compress the model and enhance inference speed through both sparsification and semistructured pruning, all while maintaining the model's performance. Importantly, SparseGPT operates locally, updating weights after each pruning step to maintain the integrity of each layer's input-output relationship. These updates are performed independently of global gradient information, as the primary objective of SparseGPT is to streamline computational demands without compromising performance. Hence, instead of optimizing all parameters globally, SparseGPT strategically updates weights locally after each pruning step.

The GPT model's extensive parameters enable SparseGPT to derive lighter, sparse versions from heavily parameterized, dense models. These sparse models closely mimic the performance of the original dense models but with much less computational complexity. Moreover, SparseGPT uses perplexity as a sensitivity measure, requiring that the sparse model's outputs closely match those of the dense model. This is challenging because it's critical to maintain important parameters that significantly impact model performance during the pruning and weight update processes.

In vast models, there is a high degree of redundancy, especially with unstructured (element-wise) or semistructured (vector-wise) sparsification. This means

many parameters can be removed without substantially affecting performance. Compared to structured pruning, these sparsification methods are less likely to lose precision, offering more flexibility in choosing which parameters to prune. SparseGPT enables substantial model compression and its effective sparse regularization can significantly cut training costs. This effectiveness is maintained by locally updating weights after each pruning step, preserving each layer's input-output relationship while minimizing model size.

However, deploying this type of sparse model for inference requires hardware and software that support sparse computational tasks. To fully benefit from SparseGPT's compression, there may be a need to develop new sparse computing libraries or enhance existing ones to handle sparse operations better. For optimal compression and acceleration, it's crucial to extend the use of the SparseGPT algorithm during model training. This involves further research into effectively integrating SparseGPT in the training of large-scale models to reduce computational demands without compromising performance.

## **9.6 Case studies**

### **9.6.1 Base model**

Meta's LLaMA, an open-sourced pretrained model released on February 24, 2023, stands out as a top choice among models designed for platforms with low computational power. It is available in various sizes, including 7B, 13B,

33B, and 65B parameters, as detailed in **Table 9.2**. The training data for LLaMA included a massive 1.4 trillion tokens, sourced primarily from CommonCrawl (67%), followed by the C4 dataset (15%). Other significant data contributions came from Github, Wikipedia, and Books (each contributing 4.5%), with smaller percentages from ArXiv (2.5%) and StackExchange (2%).

---

**Table 9.2**

| Parameter size | Dimension | $n$ Heads | $n$ Layers | Learning rate | Sample size | Token |
|----------------|-----------|-----------|------------|---------------|-------------|-------|
| 6.7B           | 4096      | 32        | 32         | $3.0e^{-4}$   | 4M          | 1.0T  |
| 13.0B          | 5120      | 40        | 40         | $3.0e^{-4}$   | 4M          | 1.0T  |
| 32.5B          | 6656      | 52        | 60         | $1.5e^{-4}$   | 4M          | 1.4T  |
| 65.2B          | 8192      | 64        | 80         | $1.5e^{-4}$   | 4M          | 1.4T  |

Data Source: Academic Paper “Llama: Open and efficient foundation language models.”

LLaMA has demonstrated superior performance over the similarly sized GPT-3 in several benchmarks, including tasks such as commonsense reasoning, question answering, mathematical reasoning, code generation, and language understanding. Notably, the 13B version of LLaMA outperforms GPT-3, which has as many as 175B parameters, in multiple tests. Moreover, the 65B version of LLaMA holds its own against DeepMind’s Chinchilla with 70B parameters and Google’s PaLM with 540B parameters, showing competitive results.

### 9.6.2 Instruction-following model

The Alpaca model, launched by Stanford University in March 2023, is a low-power transfer model designed to be efficient and effective. It is built on Meta's LLaMA 7B as the foundational model. Alpaca utilizes a unique instruction bootstrapping method discussed in **Section 9.1**, leveraging ChatGPT to create a 52,000-entry dataset. It then uses the Lora algorithm to fine-tune the LLaMA model to better follow instructions. Its conversational abilities are nearly on par with those of GPT-3.5 (text-davinci-003), as illustrated in **Fig. 9.10**.



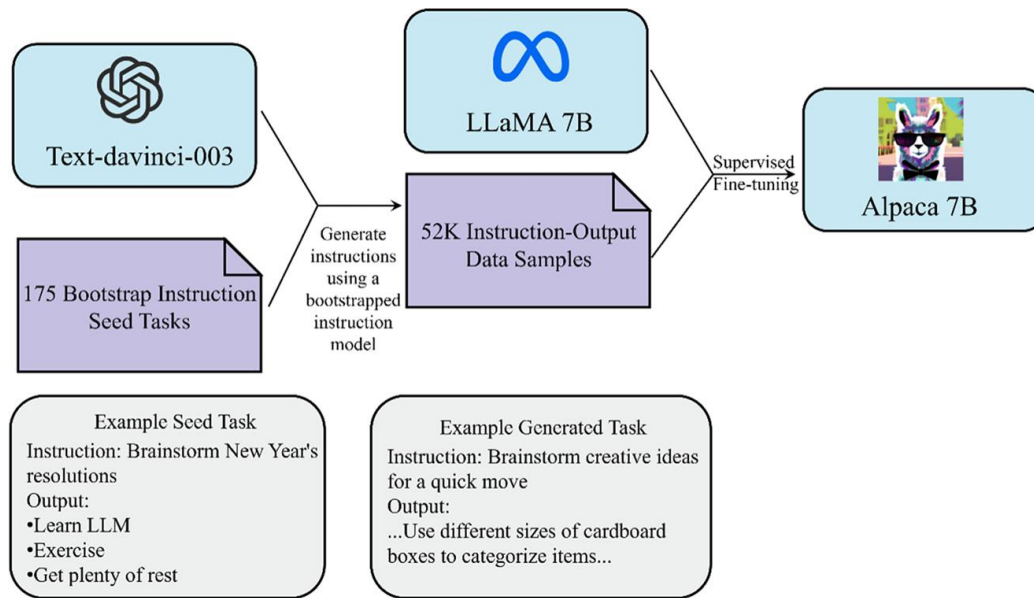


**FIGURE 9.10** Alpaca model. Image source: Paper “Alpaca: A Strong, Replicable Instruction-Following Model”.

The diagram illustrates the process of generating instructions using a bootstrapped instruction model and fine-tuning Alpaca 7B. Text-davinci-003 is used to generate instructions from 175 Bootstrap Instruction Seed Tasks. These instructions are then used to create 52K Instruction-Output Data Samples. LLaMA 7B is fine-tuned using these data samples through supervised fine-tuning to create Alpaca 7B. Example Seed Task: Instruction: Brainstorm New Year's resolutions. Output: Learn LLM, Exercise, Get plenty of rest. Example Generated Task: Instruction: Brainstorm creative ideas for a quick move. Output: Use different sizes of cardboard boxes to categorize items.

This approach to instruction fine-tuning is similar to the initial stage of human feedback RL, known as the supervised fine-tuning (SFT) stage, but it does not encompass the entire human feedback process. Despite this, Alpaca offers a highly economical solution for model transfer, requiring only 3 h of training on eight 80GB A100 GPUs, costing approximately \$500 in total. The release of Alpaca has inspired the development of several similar models, which enhance the Meta LLaMA model by incorporating more training data and using Lora's instruction fine-tuning technique. Alpaca sets a model where a larger model serves as a "Teacher," generating training data that allows a smaller "Student" model to absorb and replicate the larger model's knowledge and capabilities.

### **9.6.3 Medical field**

ChatDoctor is a project that adapts open-source, large-scale language models for use in the medical field without requiring extensive computational resources. Illustrated in **Fig. 9.11**, this model uses dialog data from patient-doctor interactions to fine-tune the Meta LLaMA base model and combines it with external knowledge from sources like Wikipedia or medical databases. This integration helps the model provide accurate medical information and generate high-quality responses within the medical domain.

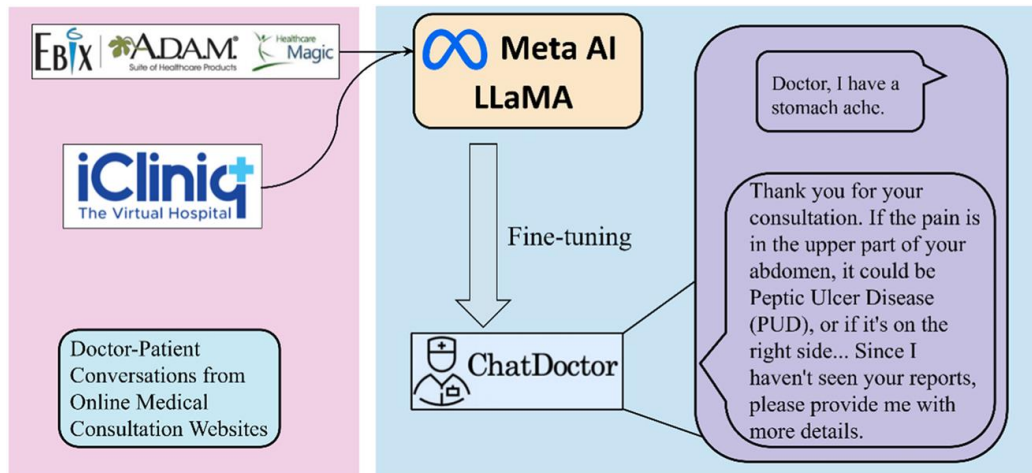


**FIGURE 9.11** Architecture diagram of ChatDoctor. Image source: Paper “Chatdoctor: A medical chat model fine-tuned on llama model using medical domain knowledge”.

The sources of the dialog data include EBIX, ADAM Suite of Healthcare Products, Healthcare Magic and iClinic The Virtual Hospital. This data is used to fine-tune the Meta AI LLaMA model, which is then used by ChatDoctor. An example conversation is shown where a patient says, Doctor, I have a stomach ache. The response from ChatDoctor is, Thank you for your consultation. If the pain is in the upper part of your abdomen, it could be Peptic Ulcer Disease (PUD), or if it's on the right side... Since I haven't seen your reports, please provide me with more details.

Here’s how the training process unfolds: initially, the model is fine-tuned using 52,000 instructional entries

derived from Stanford University's Alpaca project. Following this, it undergoes a second round of fine-tuning using real patient-doctor dialog data sourced from HealthCareMagic, an online medical consultation platform. This site contributes approximately 100,000 real dialog entries, which are carefully selected and anonymized to remove any personal information, and also corrected for any grammatical inaccuracies.

To improve its accuracy, ChatDoctor has built a knowledge base using Wikipedia and specialized medical databases. This allows the model to pull real-time, authoritative information needed to answer questions in the medically critical context where mistakes can have serious consequences. Additionally, it has developed a comprehensive database that includes information on approximately 700 diseases, detailing symptoms, diagnostic tests, and recommended treatments, establishing a robust standard for medical data.

Illustrated in **Fig. 9.12**, ChatDoctor enhances its ability to integrate this extensive knowledge by employing a keyword mining approach. This method helps identify essential terms needed to search for relevant information. The system then uses these terms to find and retrieve the most pertinent sections from the knowledge base.

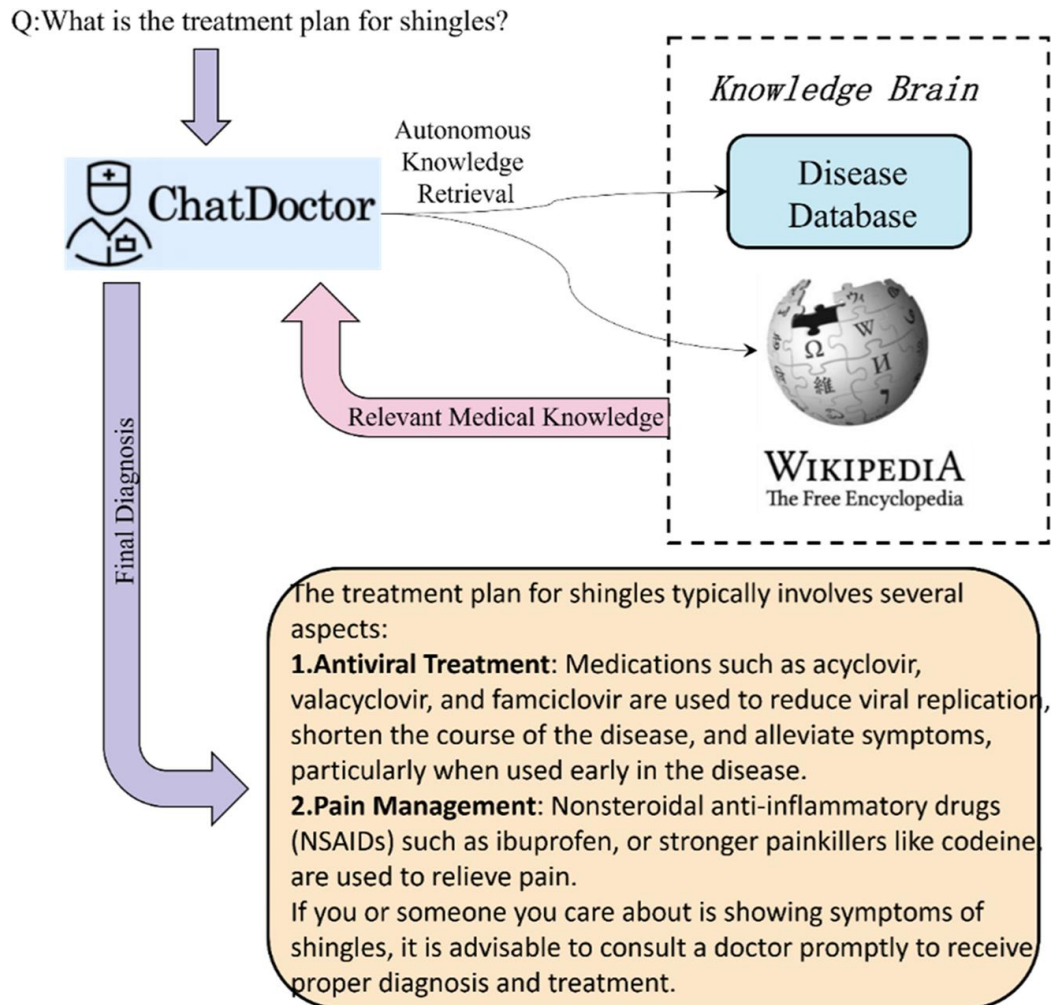


**FIGURE 9.12** Autonomous ChatDoctor based on knowledge brain. Image source: Paper “ChatDoctor: A Medical Chat Model Fine-Tuned on a Large Language Model Meta-AI (LLaMA) Using Medical Domain Knowledge”.

Q: What is the treatment plan for shingles? The illustration shows ChatDoctor using autonomous knowledge retrieval to access relevant medical knowledge from a disease database and Wikipedia. The final diagnosis is provided by

ChatDoctor. The treatment plan for shingles typically involves several aspects: 1. Antiviral Treatment: Medications such as acyclovir, valacyclovir and famciclovir are used to reduce viral replication, shorten the course of the disease and alleviate symptoms, particularly when used early in the disease. 2. Pain Management: Nonsteroidal anti-inflammatory drugs (NSAIDs) such as ibuprofen, or stronger painkillers like codeine, are used to relieve pain. If you or someone you care about is showing symptoms of shingles, it is advisable to consult a doctor promptly to receive proper diagnosis and treatment.

Recognizing that the model cannot process all data from the disease database simultaneously, ChatDoctor allows the model to access the data in batches. It autonomously selects the most relevant entries that could help address the query. These selected entries are then fed back into the model to help it generate a well-informed and accurate answer. By letting a language model, which has been fine-tuned with medical data, select and utilize information from an external knowledge base, ChatDoctor significantly boosts the reliability and precision of responses to medical inquiries.

### **9.6.4 Judicial field**

LawGPT\_zh, LexiLaw, Lawyer LLaMA, and JusticGPT are examples of how LLMs are being adapted for the legal sector in China, using less computational power. Lawyer

LLaMA and JusticGPT use the LLaMA 13B model as their foundation, whereas LawGPT\_zh and LexiLaw are based on the ChatGLM-6B model.

All these models use a training approach known as LoRA, which typically unfolds in two stages. In the first stage, the base models are pretrained on extensive collections of legal documents and statutes, providing a deep understanding of legal language and context. The second stage focuses on refining these models further through instruction fine-tuning, specifically using datasets designed for legal dialog and question answering. The sources for this data are detailed in **Table 9.3**.



**Table 9.3**

| Type                    | Data source                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Official data           | <b>China inspection network: indictments, etc.</b><br><b>China judgments online: judgments, orders, decisions, etc.</b><br><b>Ministry of justice national judicial examination center: administrative regulations database, exam papers, etc.</b><br><b>National laws and regulations database: official legal regulations database</b>                                                                                                                                                                                              |
| Competition data        | <b>China legal AI challenge (CAIL): historical competition data</b><br><b>Legal AI challenge (LAIC) by China law research: historical competition questions</b><br><b>Baidu Knows Legal Q&amp;A Dataset: about 36,000 legal Q&amp;A entries, including user inquiries, peer answers, and best answers</b><br><b>Legal Knowledge Q&amp;A Dataset: About 23,000 legal Q&amp;A entries</b><br><b>China judicial examination dataset: about 26,000 entries from the Chinese judicial examination 集</b>                                    |
| Third-party open source | <b>LaWGPT Dataset @pengxiao-song: Includes legal domain-specific vocabulary, structured criminal charge data, high-quality Q&amp;A, etc.</b><br><b>Legal charge prediction and machine Q&amp;A @liuhuanyong: includes criminal charge knowledge graphs, 200,000 legal Q&amp;A data, etc.</b><br><b>Legal provisions knowledge extraction @liuhuanyong: includes legal judgments and criminal case data</b><br><b>China Law Handbook @LawRefBook: collection of various legal regulations, departmental rules, case examples, etc.</b> |
| Legal service websites  | <b>HuaLaw, FindLaw, DaLaw, Good Lawyer Network, AskLaw: Providing legal consultation Q&amp;A and other service data</b><br><b>Flssw: legal consultations, legal news, case analysis, etc.</b>                                                                                                                                                                                                                                                                                                                                         |
| Books and journals      | Legal dictionaries, e-books, scholarly articles, etc.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |

During the second phase of fine-tuning with instructions, not only are Q&A data from legal services websites utilized, but an innovative instruction bootstrapping method is also introduced. This approach uses LLMs like ChatGPT to either generate new Q&A entries or refine and optimize existing ones. For instance, LawGPT\_zh, as illustrated in **Fig. 9.13**, uses a method called reliable-self-instruction, which is grounded in specific knowledge. By feeding ChatGPT detailed texts about legal knowledge, it generates a series of questions that are closely linked to the legal content and its logical implications. ChatGPT then answers



these questions using a “text segment-question” format. This process creates legally informative answers and ensures their accuracy.

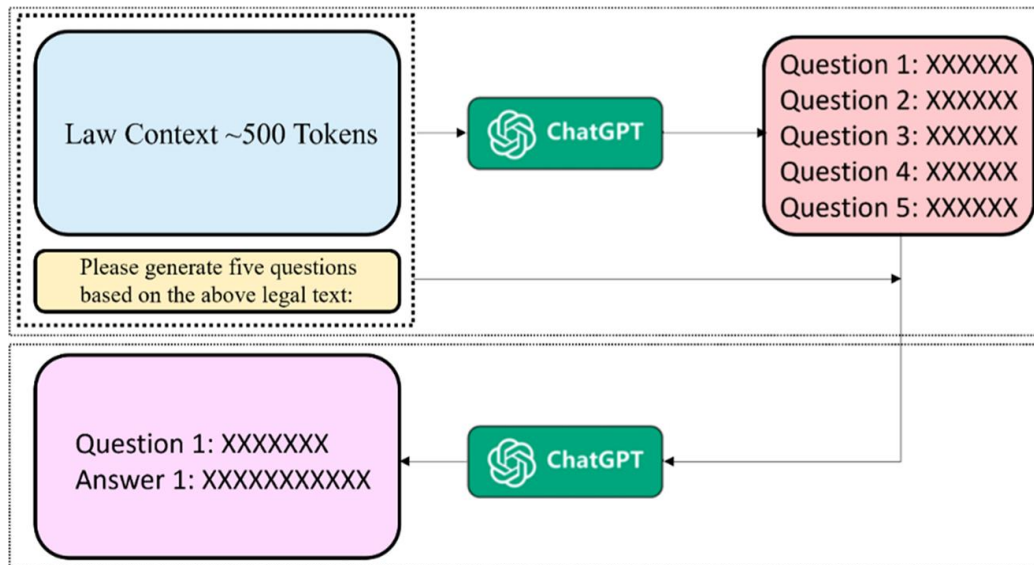


**FIGURE 9.13** Diagram of LawGPT\_zh instruction bootstrapping annotation.

The diagram illustrates the process of generating questions and answers using ChatGPT based on a legal text. The first section shows a box labeled Law Context approximately 500 Tokens with a note below it saying Please generate five questions based on the above legal text. This is connected to a ChatGPT box, which then leads to another box listing Question 1: XXXXXX, Question 2: XXXXXX, Question 3: XXXXXX, Question 4: XXXXXX and Question 5: XXXXXX. The second section shows a box labeled Question 1: XXXXXXXX Answer 1: XXXXXXXXXXXXX, which is connected back to the ChatGPT box.

While models like LawGPT\_zh, LexiLaw, and Lawyer LLaMA are primarily used for legal consultation, JusticGPT stands out as an experimental judicial decision support model developed by Xiangtan University's Smart Justice

research team. Built upon a LLM, JusticGPT is finely tuned to better understand and generate legal language, handling everything from legal terms to case analysis and generating recommendations for legal decisions. Additionally, it uses an extensive external knowledge base that stores a wealth of cases and judicial documents as vectors, enhancing the model's decision-making power.

Illustrated in **Fig. 9.14**, consider a scenario where a user submits a legal query. This query is first converted into a vector, which is then matched against a database using a similarity algorithm to identify the top relevant cases. These selected cases provide context that, along with the initial query, is fed into JusticGPT as a prompt. JusticGPT assesses both the context and the query to produce a ranked list of case similarities.

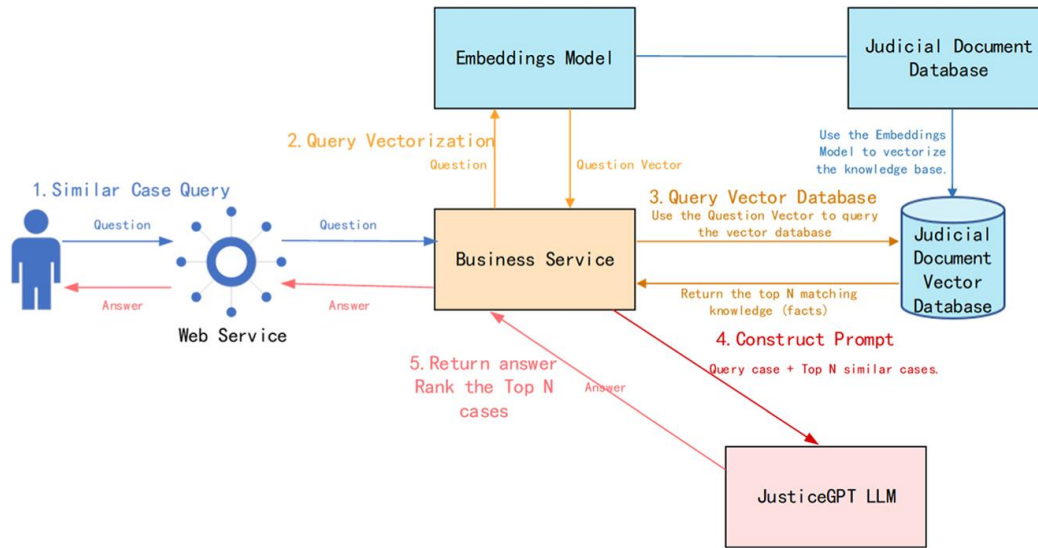


**FIGURE 9.14** Architecture diagram of JusticeGPT.

The process starts with 1. Similar Case Query where a user submits a question to the Web Service, which returns an answer. The question is also sent to the Business Service. 2. Query Vectorization: The Business Service sends the question to the Embeddings Model, which returns a question vector. 3. Query Vector Database: The question vector is used to query the vector database, which returns the top N matching knowledge (facts) from the Judicial Document Vector Database. 4. Construct Prompt: The query case and the top N similar cases are used to construct a prompt. 5. Return answer: The constructed prompt is sent to JusticeGPT LLM, which ranks the top N cases and returns the answer.

JusticGPT is a pioneering project that marries advanced language processing technology with a vast legal database.

It aims to dramatically improve the efficiency and accuracy of typical judicial decisions, paving the way for genuine AI-assisted decision-making in the legal field.

## 9.7 Summary

Fine-tuning or transferring language models for specific industries typically involves adapting open-source pretrained models to perform particular tasks within those sectors. This adaptation faces two main challenges: reducing the cost associated with human annotation and minimizing the computational resources required for both training and deployment.

Most current open-source models that have been transferred are only fine-tuned with instructions, which is similar to the SFT phase found in RLHF. Although there are RLHF versions of the LLaMA model, such as ChatLLaMA in English and ColossalChat, these models don't significantly outperform in terms of size or task effectiveness. Additionally, tailoring models for specific sectors involves creating specialized reward systems and using proximal policy optimization for fine-tuning with data from those sectors, which adds to the cost of data annotation.

There is notable enthusiasm for applying large open-source language models to sectors like healthcare, law, finance, banking, and education, with numerous entities exploring and launching tailored models.

It's crucial to highlight that Meta's LLaMA model is not for commercial use. Moreover, datasets used for instruction

fine-tuning, such as the Alpaca expansion set, are generated using OpenAI's language model application programming interface (API) and instruction bootstrapping techniques. Using this data commercially could lead to conflicts with OpenAI and pose risks related to intellectual property.

---

## CHAPTER 10

# Middleware

---

### Abstract

This chapter explores the evolving role of middleware in enhancing large language models (LLMs) like ChatGPT. It addresses the challenges these models face, such as the difficulty of updating knowledge and verifying information accuracy. Middleware solutions, including LangChain and AutoGPT, are introduced to tackle these issues by integrating external knowledge, improving reasoning abilities, and managing complex workflows. LangChain is detailed as a comprehensive framework offering tools for model integration, prompt management, memory handling, and workflow orchestration. AutoGPT is discussed as an advanced agent that autonomously manages tasks with minimal human input. The chapter also examines competitors like Microsoft's Semantic Kernel and GPT-Index, highlighting the growing competition in middleware frameworks. Despite challenges, middleware is poised to significantly impact small and medium-sized enterprises by boosting productivity through LLMs.

## Keywords

Middleware; large language models; LangChain; AutoGPT; task automation; memory management; workflow integration; Semantic Kernel; ChatGPT plugins

Large language models (LLMs) like ChatGPT face two main challenges. Firstly, because these models are so complex, updating them can be expensive and cumbersome. As a result, it's hard to keep their knowledge up to date.

Secondly, they can't check the correctness of the information they generate. To overcome these issues, it's critical to incorporate external knowledge and improve their ability to reason, especially in areas like math and logic. Moreover, to effectively use these models across different professional fields, users need specialized tools that can handle long-term memory and manage detailed workflows. This need has spurred the development of a middleware ecosystem around LLMs. Over the next 3–5 years, this middleware market is expected to grow quickly, focusing on enhancing models' access to external knowledge, boosting their reasoning skills, refining user prompts, and streamlining workflows.

### 10.1 LangChain

LangChain is an open-source framework that helps developers create applications from start to finish using LLMs. It's part of a lively open-source community.

Normally, these models are accessed through representational state transfer (REST) application



programming interfaces (APIs), but LangChain goes beyond just connecting to these APIs. It tackles the integration of LLMs with a company's internal resources like documents, databases, and platforms such as SharePoint. It also facilitates interaction with various systems like systems, applications, and products (SAP), enterprise resource planning (ERP), customer relationship management (CRM), and human resources (HR), and can manage IT ticketing systems. LangChain keeps track of conversations, seamlessly integrates prompts into the programming, optimizes the use of tokens, and ensures operations stay within set service limits and restrictions. Additionally, LangChain serves as an orchestrator in business workflows, efficiently managing inputs and outputs from different sources like OpenAI, Azure Search, and various databases.

LangChain is a robust collection of tools and components designed to make it easier to develop applications with LLMs. It delivers these seven key capabilities:

**Model:** LangChain wraps up the interfaces for calling LLMs, supporting three types: text mode, dialog mode, and embedding mode. In text mode, it inputs and outputs plain text strings. In dialog mode, it handles the context of conversations, taking in and outputting chat messages. In embedding mode, it inputs text and outputs a series of text token embeddings. LangChain's ability to interface with multiple LLMs on the market lets developers switch easily between them, reducing reliance on any single platform.

Prompt: LangChain offers a comprehensive library of prompt templates suitable for different uses. It supports basic text and dialog interactions and allows for customized formatting of outputs, which simplifies tasks like switching between different underlying models, managing, and optimizing prompts (**Fig. 10.1**).

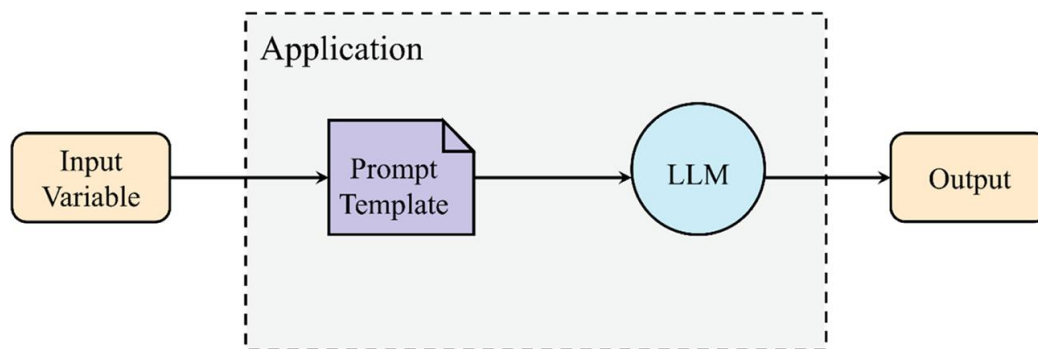


**FIGURE 10.1** Prompt template library.

It consists of an input variable leading to a prompt template, which then connects to an LLM and finally produces an output. The entire process is enclosed within a dashed boundary labeled Application.

Memory: LangChain's memory module is tailored for complex uses like maintaining ongoing states across different requests and sessions in a chain or agent calls, which helps LLMs remember past interactions. It includes standardized interfaces for managing contexts, multiple implementations of these contexts, and examples of chains or agents that utilize this context management.

Index: Illustrated in **Fig. 10.2**, the indexing module is crucial for integrating external knowledge into LLMs. It includes interfaces for reading and writing various document types, a text slicing tool to break down large documents, and a token interface for turning documents into vectors, enhancing the model's ability to digest and utilize large amounts of data.

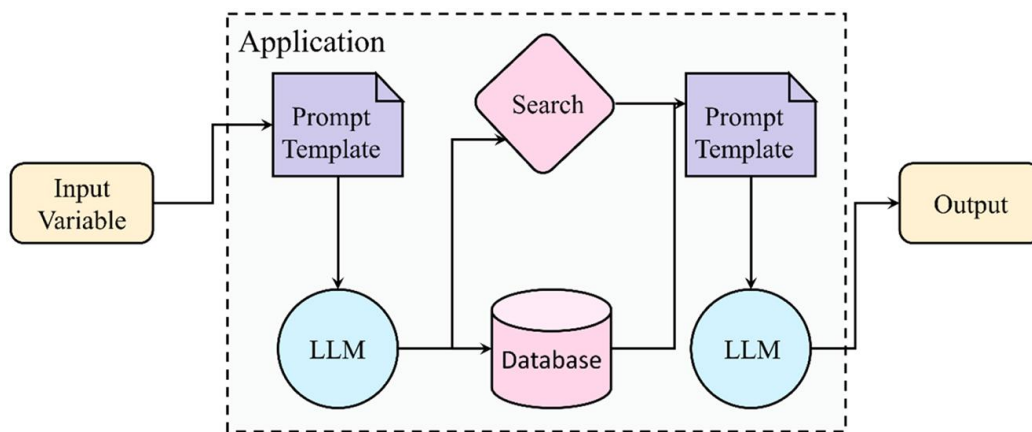


**FIGURE 10.2** Indexing module.

The process starts with an Input Variable, which connects to a Prompt Template. This Prompt Template is linked to an LLM. The LLM is connected to a Database. The Database is connected to a Search component, which is linked to another Prompt Template. This second Prompt Template is connected to another LLM, which finally connects to the Output.

Chain: For more complex applications, simply building prompts isn't enough. A "chain" involves sequences of calls, not just to a single LLM, but potentially to other

utilities as well. This includes linking LLMs to other sources like search APIs or external databases to expand their capabilities. LangChain facilitates this with standardized interfaces, seamless integration with other tools, and provides practical examples of end-to-end chains for common use cases.

A straightforward example (illustrated in **Fig. 10.3**) is the “self ask with search” chain. This setup combines the OpenAI API with the SerpApi (a Google search API), enabling the LLM to methodically inquire and ultimately determine the hometown of the winner of the US Open tennis championship.

```
In [1]: from langchain import SelfAskwithSearchchain, OpenAI, SerpAPIChain

llm = OpenAI ( temperature=0)
search = SerpAPIChain ( )

self_ask_with_search =
SelfAskwithSearchChain( llm=llm, search_chain=search)

self_ask_with_search.run ("Where is the hometown of the defending
champion in the men's singles of the US Open? ")
```

Where is the hometown of the defending champion in the men's singles of the US Open?

Is a follow-up question required: Yes.

Follow-up question: Who is the defending champion of the US Open?

Intermediate answer: Carlos Alcaraz.

Further question: What is the nationality of Carlos Alcaraz?

Intermediate answer: Spanish, from Murcia, El Palmar.

Therefore, the final answer is: Spain, Murcia, El Palmar.

Out [1]: "Where is the hometown of the defending champion of the US Open men's singles?\nIs a follow-up question necessary: Yes.\nFollow-up question: Who is the defending champion of the US Open?\nIntermediate answer: Carlos Alcaraz\nFollow-up question: What is the nationality of Carlos Alcaraz?\nIntermediate answer: Spanish, Murcia, El Palmar\nThus, the final answer is: Spain, Murcia, El Palmar."

**FIGURE 10.3** Application of chains: self-ask with search.

The code reads as follows: from langchain import SelfAskwithSearchchain, OpenAI, SerpAPIChain llm equals OpenAI left parenthesis temperature equals 0 right parenthesis search equals SerpAPIChain left parenthesis right parenthesis selfaskwithsearch equals SelfAskwithSearchChain left parenthesis llm equals llm, searchchain equals search right parenthesis selfaskwith\_search.run left parenthesis double quote Where is the hometown of the defending champion in the men's

singles of the US Open? double quote right parenthesis The output shows a series of questions and answers: Where is the hometown of the defending champion in the men's singles of the US Open question mark, Is a follow-up question required: Yes. Follow-up question: Who is the defending champion of the US Open? Intermediate answer: Carlos Alcaraz. Further question: What is the nationality of Carlos Alcaraz question mark, Intermediate answer: Spanish, from Murcia, El Palmar. Therefore, the final answer is: Spain, Murcia, El Palmar. The output below the code is formatted as a string with newline characters indicating line breaks.

Agent: The agent module is even more advanced than the chain module. As shown in **Fig. 10.4**, agents are not merely about splitting up workflows. They empower LLMs to independently choose their actions and the tools they use. This can include search engines, databases, any form of input or output strings, and could even involve interfacing with another LLM, chain, or agent. LangChain equips these agents with standardized interfaces, offers a selection of different agent types, and provides practical examples of how agents can be implemented from start to finish.

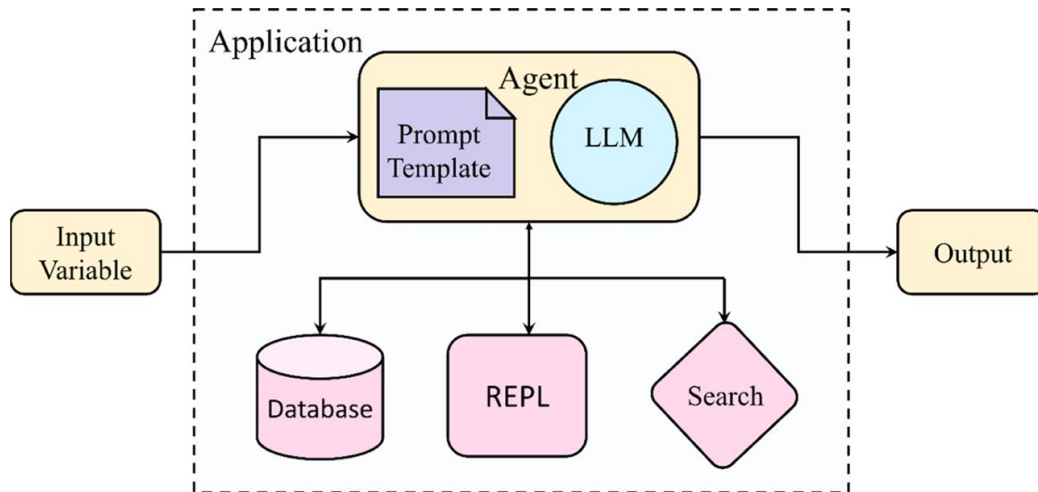


**FIGURE 10.4** Diagram of an agent.

The workflow starts with an input variable, which is processed by an agent. The agent consists of a prompt template and an LLM. The agent interacts with a database, REPL and search components. The final output is generated after processing through these components.

Callback: The callback module boosts how events within chains or agents are observed and examined, simplifying the process of tracking and managing them.

LangChain offers developers a robust suite of tools and components that simplify the task of building applications with LLMs. LangChain not only facilitates the integration of external knowledge sources into models like ChatGPT but also improves these models' abilities in areas like mathematical reasoning. This enhancement has boosted the performance of LLMs across various tasks and fields. Consequently, LangChain has risen to become the leading

middleware framework for LLMs, gaining widespread support from third parties (**Table 10.1**).



---

**Table 10.1****Features provided by LangChain or supported integrations.**

|                                  |                                                                                              |
|----------------------------------|----------------------------------------------------------------------------------------------|
| Data preprocessing               | UnstructuredIO, Airbyte...                                                                   |
| Data indexing                    | GPT-Index...                                                                                 |
| Doc&Text splitter                | Generic Recursive Text Splitter, Markdown Splitter, Python Code Splitter...                  |
| Vector database and retrieval    | FAISS, Pinecone, Weaviate, Elastic...                                                        |
| Graph database                   | Chroma...                                                                                    |
| External knowledge or operations | SerpApi, Searx, Wikipedia API, Wolfram Alpha, Zapier Natural Language Actions API...         |
| LLM API                          | OpenAI, Hugging Face, Cohere, Anthropic, PaLM, Goose AI, CerebrumAI, Forefront AI, Petals... |
| Embedding engine                 | OpenAI, Hugging Face, Cohere...                                                              |
| Observability                    | Helicone, Prompt Layer, Weights&Biases...                                                    |
| Application deployment           | Streamlit, Hugging Face(Gradio), Steamship, Kookaburra...                                    |

|                                           |                                                     |
|-------------------------------------------|-----------------------------------------------------|
| Model evaluation dataset                  | Hugging Face (truthful qa),<br>LangchainDatasets... |
| Model response structuring and validation | Kor, guardrails...                                  |

## 10.2 AutoGPT

AutoGPT capitalizes on the strengths of LLMs for task planning and serving as intelligent agents. Once it is set with one or more long-term objectives, AutoGPT can autonomously create and modify prompts to interact with LLMs, initiate and prioritize tasks, and focus on new tasks first. This cycle continues until all set goals are fulfilled. Moreover, AutoGPT manages short-term and long-term memories using vector databases and files and can execute various Internet-based tasks like searching, form filling, and interacting with APIs.

AutoGPT leverages the task planning and smart agent functions of LLMs, processing task goals and execution feedback in a self-evolving manner. This is distinct from ChatGPT, which relies on manual commands for each task. AutoGPT's minimal human input requirement is a critical feature, allowing it to not only generate text but also manage entire projects. It has been effectively utilized in creating websites, generating blogs, analyzing markets, formulating trading strategies, writing product reviews, and crafting marketing plans.

Projects similar to AutoGPT include Baby artificial general intelligence (AGI) and AgentGPT. Baby AGI, utilizing OpenAI and Chromea, automates task creation, priority setting, and execution, with new tasks derived from previous outcomes and set objectives. AgentGPT provides a user-friendly web interface, enabling users to create autonomous AI/AGI agents that coordinate tasks to reach goals with minimal user interaction. AgentGPT's browser-based functionality requires no installation, offering an advantage over AutoGPT and similar tools. Furthermore, VisualGPT and HuggingGPT represent specialized cases of autonomous agents and task planning in multimodal contexts.

As depicted in **Fig. 10.5**, AutoGPT, powered by GPT-4, enhances its performance over time through learning from experience and feedback. It autonomously generates and executes long-term business strategies and models. Key functionalities include the following.

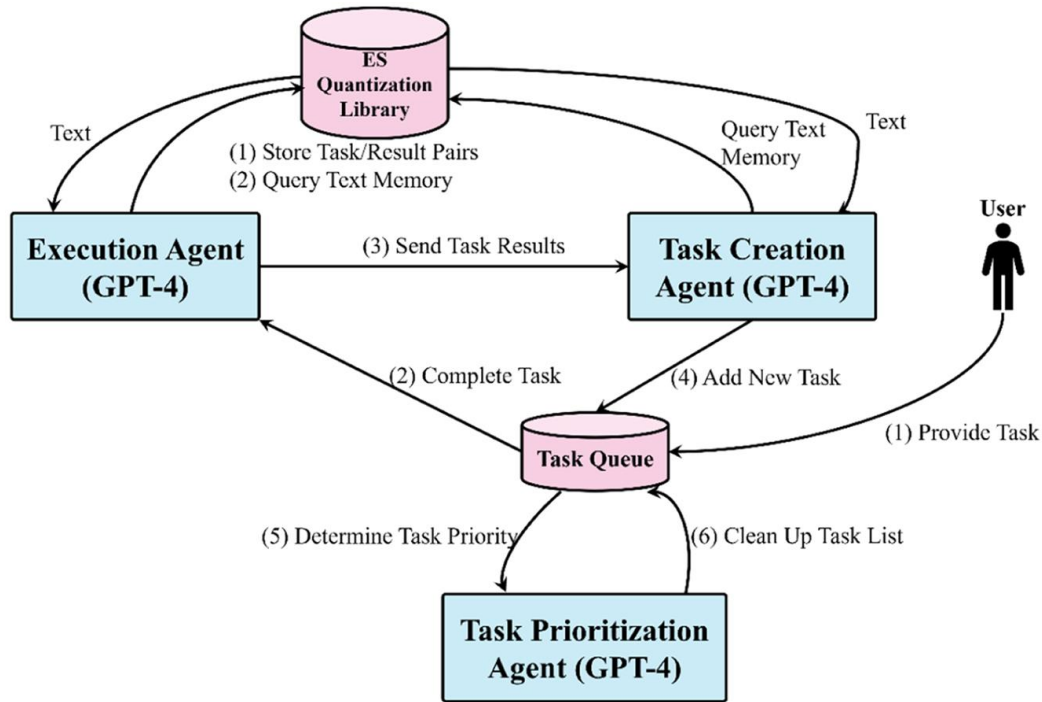


**FIGURE 10.5** Principle of AutoGPT.

The components are: ES Quantization Library, Execution Agent (GPT-4), Task Creation Agent (GPT-4), Task Queue, Task Prioritization Agent (GPT-4) and a User. The process is as follows: (1) The User provides a task to the Task Creation Agent (GPT-4). (2) The Task Creation Agent (GPT-4) queries text memory from the ES Quantization Library and adds a new task to the Task Queue. (3) The Execution Agent (GPT-4) sends task results to the Task Creation Agent (GPT-4). (4) The Task Creation Agent (GPT-4) adds new tasks to the Task Queue. (5) The Task Prioritization Agent (GPT-4) determines task priority. (6) The Task Prioritization Agent (GPT-4) cleans up the task list.

Additionally, the Execution Agent (GPT-4) completes tasks and stores task/result pairs in the ES Quantization Library.

AutoGPT incorporates memory management via Redis or third-party storage solutions, facilitating both short-term and long-term memory within its conversational system. This setup enables the preservation of past inputs for future reference. The model treats short-term memory as a temporary repository, effective from a few seconds up to a minute, whereas long-term memory serves as a reservoir for substantial knowledge and extended historical data, supporting storage for up to 64,000 words. This dual memory system allows AutoGPT to progressively learn and evolve into a more proficient AI agent.

AutoGPT stores historical data in lists and maximizes the historical context provided to GPT-4 with each interaction, contingent upon token availability. Code analyses indicate that AutoGPT maximizes token usage for each task, incorporating previous instructions if space allows after entering a new command. Moreover, each interaction with GPT-4 includes pertinent contextual information like the current time, aiding the handling of temporally sensitive tasks. AutoGPT maintains a comprehensive historical record, ensuring the most relevant information is relayed to GPT-4 during each interaction.

AutoGPT leverages its Internet access to collect data, efficiently sourcing and compiling information from a wide array of online platforms. This includes data from social

media, financial records, consumer behaviors, and market trends, enabling AutoGPT to adapt and respond to new information and evolving market conditions effectively. This capability positions AutoGPT as a valuable tool for both data gathering and analytical research.

Despite being a groundbreaking open-source experiment, AutoGPT is not yet a fully developed application or product and may fall short in complex, real-world business environments. Its limitations in understanding context can lead to irrelevant outputs, potentially wasting time and resources since such outputs might need further editing to align with the desired context. AutoGPT's effectiveness in task completion largely depends on feedback from large-scale models, and without a reliable method to measure the accuracy of these models' outputs, it cannot guarantee the precision or authenticity of its results. Moreover, AutoGPT may not fully recognize the constraints of its created agents, sometimes resulting in misperceptions. Nonetheless, AutoGPT is still a robust tool, especially effective in managing simpler tasks that can be segmented into smaller components.

## **10.3 Competitors in middleware frameworks**

LLMs are adept at selecting and utilizing the right tools and can even plan tasks to meet long-term objectives. Middleware frameworks are becoming crucial portals for next-generation human-computer interactions, inviting

fierce competition from the market. Among the competitors facing LangChain, key players include the following:

Microsoft's Semantic Kernel project. As illustrated in **Fig. 10.6**, Semantic Kernel shares some objectives with LangChain but emphasizes the use of "large language models" in its framework rather than specifically "GPT-4." This approach is due to its strategy of employing multiple LLMs concurrently to manage business processes, which encompasses session management, a prompt engine, memory management, and connectivity needs for services like dialog histories and external APIs. However, Semantic Kernel targets the engineering community, aiming to integrate LLM functionalities into existing applications, a strategic focus that sets it apart from LangChain's emphasis on independent developers. While Microsoft poses a formidable challenge to LangChain, its close ties with OpenAI may limit its flexibility in supporting a diverse range of LLMs in the future, unlike LangChain.

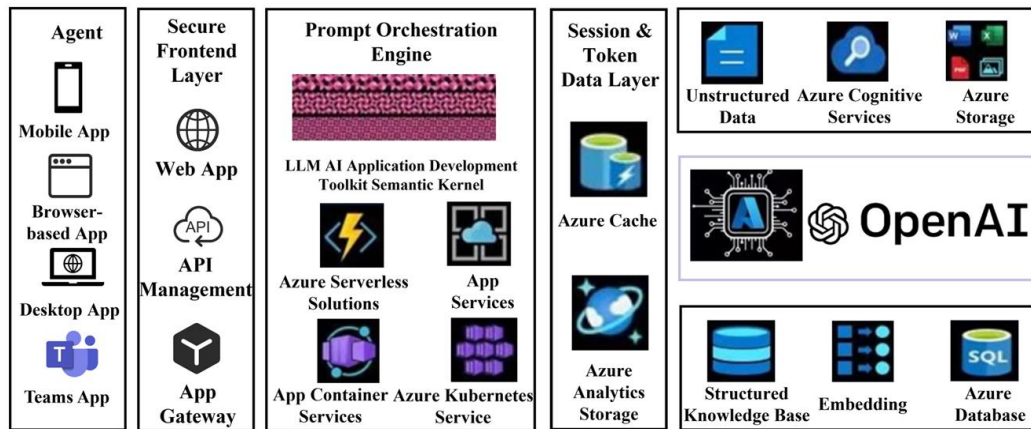


**FIGURE 10.6** Architecture of the Semantic Kernel AI interface app.

The components are divided into several layers: Agent, Secure Frontend Layer, Prompt Orchestration Engine, Session & Token Data Layer and various Azure services. The Agent layer includes Mobile App, Browser-based App, Desktop App and Teams App. The Secure Frontend Layer includes Web App, API Management and App Gateway. The Prompt Orchestration Engine includes LLM AI Application Development Toolkit Semantic Kernel, Azure Serverless Solutions, App Services and App Container Azure Kubernetes Service. The Session & Token Data Layer includes Azure Cache and Azure Analytics. The various Azure services include Unstructured Data, Azure Cognitive Services, Azure Storage, Structured Knowledge Base, Embedding and Azure Database. The diagram also features the OpenAI logo.

Dust and Scale AI Spellbook represent the trend towards no-code and low-code approaches in developing



applications with LLMs. While these platforms provide intuitive UI/UX designs, many developers might seek platforms that offer a broader range of functionalities and allow for more experimental flexibility beyond the constraints of low-code environments.

GPT-Index, which leverages LangChain, is particularly geared towards enhancing memory capabilities and facilitating data integration with LLMs. In comparison, LangChain provides a more extensive and abstract set of functionalities, enabling users to tailor and assemble components according to their specific needs and use cases.

For end users, one of LangChain's most formidable competitors is the ChatGPT plugin. LangChain, as middleware, leverages ChatGPT's linguistic understanding to transform natural language into a format compatible with Wolfram tools, a scientific search engine rich in knowledge and facts, to access Wolfram's robust mathematical reasoning capabilities (as seen in **Fig. 10.7**). However, shortly after securing a \$10 million investment, LangChain found itself facing a new challenge as OpenAI launched official ChatGPT plugins, including one for Wolfram. These plugins overlap significantly with LangChain's functionality by enabling Internet data access, integration with external knowledge bases, and the utilization of third-party tools. This development suggests that the plugins could compensate for some of ChatGPT's initial limitations, such as enhancing real-time updates and

incorporating external knowledge, potentially diminishing the role of middleware like LangChain within the broader ChatGPT ecosystem.

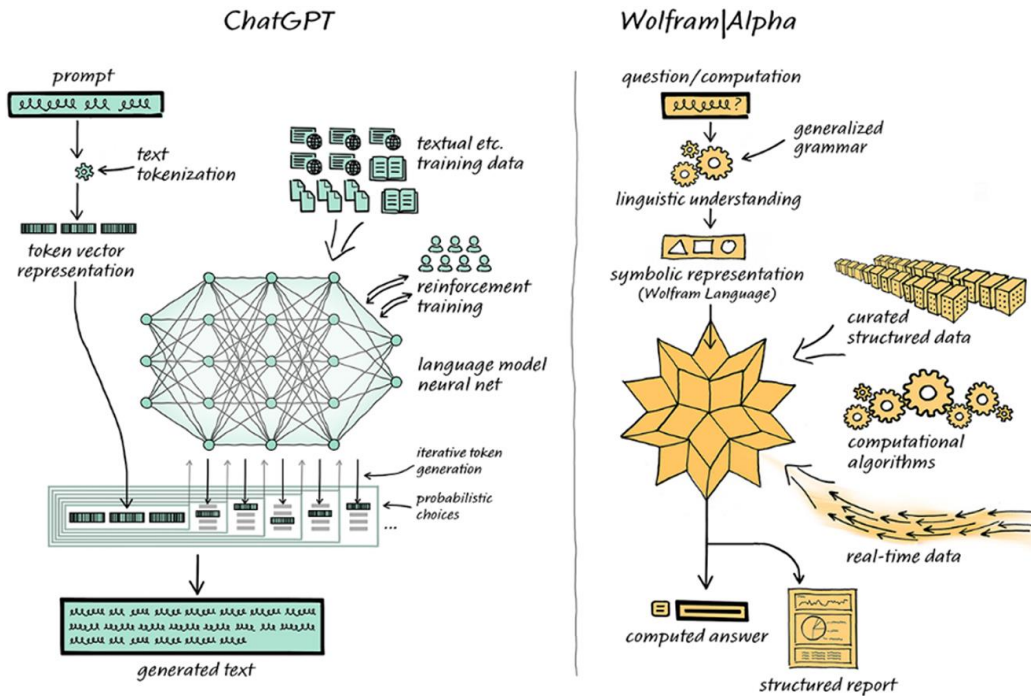


**FIGURE 10.7** ChatGPT and Wolfram|Alpha.

<https://writings.stephenwolfram.com/2023/01/wolframalpha-as-the-way-to-bring-computational-knowledge-superpowers-to-chatgpt/>.

On the left, ChatGPT starts with a prompt, which undergoes text tokenization to create a token vector representation. This representation is processed by a language model neural net, which is trained using textual training data and reinforcement training. The neural net generates iterative tokens and probabilistic choices, resulting in generated text. On the right, WolframAlpha begins with a question or computation, which is processed through generalized grammar and linguistic understanding.

This leads to symbolic representation using Wolfram Language. The symbolic representation accesses curated structured data, computational algorithms and real-time data to produce a computed answer and a structured report.

The release of ChatGPT plugins is widely viewed as a strategic move by OpenAI towards commercialization, aimed at creating an app store for the era of LLMs. Another viewpoint is that the initiative to release plugins was also motivated by OpenAI's interest in gathering insights on how users apply apps and APIs to address specific challenges.

The market for new-generation human-computer interactions, powered by LLMs capable of understanding user intents and adeptly selecting and employing tools to fulfill tasks, is heating up. Besides OpenAI, entities such as Adept AI, Inflection AI, and Meta's Toolformer are competing for leadership in this space. Importantly, for LLMs to serve as the backbone of future human-computer interactions, they must be both accurate and reliable.

ChatGPT plugins have encroached on the market space traditionally occupied by LangChain. These plugins rely solely on OpenAI's LLMs, which could pose unforeseen risks to their providers, including businesses. LangChain, by comparison, can work with various open-source and proprietary LLMs, offering more flexibility in integrating third-party resources and potentially garnering more trust

from niche market developers. Consequently, LangChain may be the superior choice for developers looking to create complex systems and manage their backend applications. This is largely thanks to LangChain's robust open-source community, which supports an extensive ecosystem of third-party tools—a distinct competitive edge.

## **10.4 Summary**

Middleware tools based on LLMs need to call these models and blend them with business logic. However, the sensitivity of business data limits their deployment in many corporations. Currently, there are no established quantitative methods to effectively gauge the output of these models, and despite their potential importance in production environments, concerns about latency and security are significant. It might be more practical to allow users to configure models and generate their own prompts for model invocation.

On the other hand, using middleware to integrate LLMs can considerably boost the productivity of small and medium-sized enterprises (SMEs). Over the next 3–5 years, it's anticipated that SMEs will increasingly adopt tools like LangChain and other third-party middleware to streamline their operations through the use of LLMs, helping them stay competitive in a fierce market. This indicates a promising growth trajectory for the middleware ecosystem.

---

## CHAPTER 11

# The future path of large language models

---

### Abstract

This chapter explores the future trajectory of large language models (LLMs) like GPT-4, emphasizing their potential to evolve into strong artificial intelligence (AI). The chapter discusses the challenges these models face, such as data resource limitations and the inherent constraints of autoregressive architectures. While GPT-4 has demonstrated remarkable capabilities, questions remain about whether scaling up data and parameters alone will lead to true general AI. The chapter also delves into concepts like embodied intelligence, highlighting projects like Google's PaLM-E and Microsoft's ChatGPT for Robotics, which integrate LLMs with robotics. These advancements suggest a future where AI could perform complex tasks across multiple domains. However, significant hurdles remain in achieving robust, adaptable AI systems that can autonomously learn and make decisions in real-world environments.

## Keywords

Artificial intelligence; large language models; GPT-4; strong AI; autoregressive models; data resource depletion; embodied intelligence; PaLM-E; robotics; task planning

GPT-4 has garnered worldwide attention due to its impressive capabilities. As the field of artificial intelligence (AI) advances, there is increasing concern about its potential to pose threats to human civilization. During the creation of this book, more than a thousand leaders in technology and business, spearheaded by Elon Musk, have urged tech companies to pause their research. They have initiated a thorough debate on whether GPT-5 and future iterations could match or even exceed human intelligence. Nonetheless, some argue that the AI journey led by GPT might be limited by the availability of data resources and the inherent constraints of its autoregressive models, which could prevent it from achieving true strong AI. This chapter delves into the myriad challenges that large language models (LLMs) face.

## 11.1 The path to strong artificial intelligence

GPT-4 seamlessly blends reasoning abilities with a vast repository of human knowledge. Its formidable logic and reasoning are not only the bedrock of GPT-4 exceptional user experience but also potentially chart a course for future advancements. This naturally raises the question:

Have the GPT series models laid the groundwork for the development of strong AI?

When delving into the workings of the GPT series, one cannot help but ponder the origins of these impressive capabilities. From a mathematical or machine learning standpoint, autoregressive language models construct a model of the probabilistic relationships between sequences of words. They use combinations of words in sentences—treated as mathematical vectors—as inputs to predict the probability distribution of various words in subsequent positions. Theoretically, the larger the sequence generated, the greater the accumulated errors.

The technological path that OpenAI has taken with its Transformer’s Decode architecture raises a question: Is it grounded in rational mathematical deduction, or is it akin to mystical “alchemy”? A clear pattern emerges from examining the evolution of LLMs: As these models, like GPT-3, GLaM, LaMDA, and Megatron-Turing NLG, are trained on increasingly vast datasets and expand in size, they reach a critical point. Beyond this point, they begin to display complex abilities and features that were unforeseen by their developers, known as the models’ emergent capabilities.

These emergent abilities defy prediction by traditional scaling laws, indicating that their development is not linear in relation to the increase in parameter size. As illustrated in **Fig. 11.1**, these specific capabilities might seem random or insignificant until they cross a certain threshold—likely



between  $10^{22}$  FLOPs and  $10^{23}$  FLOPs—after which there is a significant leap in their performance. The exact threshold for these emergent capabilities varies, depending on factors like the quality of the training data, with higher quality data possibly requiring fewer parameters and less training.

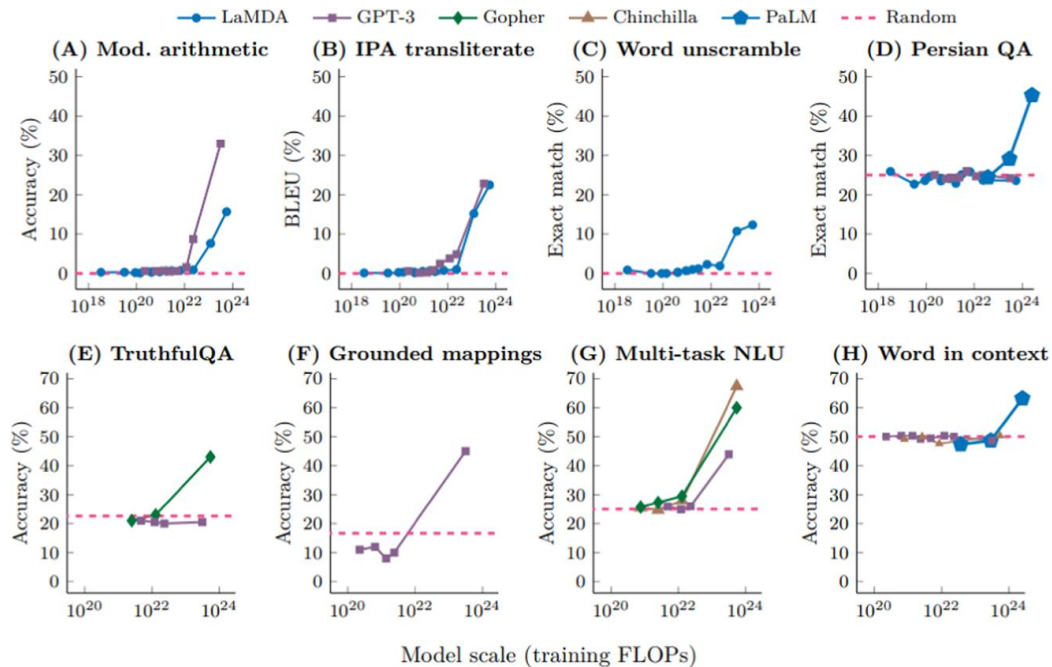


**FIGURE 11.1** Emergent capabilities of large language models. Image source: Paper “Emergent abilities of large language models”.

The image contains 8 graphs labeled A through H, each showing the performance of different AI models on specific tasks as the model scale increases. (A) Mod. arithmetic: The y-axis shows Accuracy (percent) from 0 to 50. The x-axis shows Model scale (training FLOPs) from  $10^{18}$  to  $10^{24}$ . LaMDA, GPT-3 and PaLM models are plotted. GPT-3 shows a sharp increase at  $10^{23}$  FLOPs, reaching about 33 percent accuracy. PaLM shows an increase at  $10^{24}$  FLOPs, reaching about 15 percent accuracy. (B) IPA transliterate: The y-axis shows BLEU (percent) from 0 to 50. The x-axis is the same as (A).

LaMDA and PaLM models are plotted, both showing a sharp increase at  $10^{24}$  FLOPs, reaching about 23 percent BLEU. (C) Word unscramble: The y-axis shows Exact match (percent) from 0 to 50. The x-axis is the same as (A). Only PaLM is plotted, showing an increase at  $10^{23}$  FLOPs, reaching about 12 percent exact match. (D) Persian QA: The y-axis shows Exact match (percent) from 0 to 50. The x-axis is the same as (A). Only PaLM is plotted, showing a sharp increase at  $10^{23}$  FLOPs, reaching about 45 percent exact match at  $10^{24}$  FLOPs. (E) TruthfulQA: The y-axis shows Accuracy (percent) from 0 to 70. The x-axis shows Model scale from  $10^{20}$  to  $10^{24}$  FLOPs. GPT-3 and Gopher models are plotted. Gopher shows an increase at  $10^{23}$  FLOPs, reaching about 43 percent accuracy. (F) Grounded mappings: The y-axis shows Accuracy (percent) from 0 to 70. The x-axis is the same as (E). Only GPT-3 is plotted, showing a sharp increase at  $10^{23}$  FLOPs, reaching about 45 percent accuracy. (G) Multi-task NLU: The y-axis shows Accuracy (percent) from 0 to 70. The x-axis is the same as (E). GPT-3, Gopher and Chinchilla models are plotted. All three models show sharp increases at  $10^{23}$  FLOPs, with Chinchilla reaching the highest accuracy of about 68 percent. (H) Word in context: The y-axis shows Accuracy (percent) from 0 to 70. The x-axis is the same as (E). GPT-3 and PaLM models are plotted. PaLM shows a

sharp increase at 10 raised to the power of 23 FLOPs, reaching about 63 percent accuracy. All graphs include a dashed line representing Random performance. The legend indicates different models: LaMDA, GPT-3, Gopher, Chinchilla, PaLM and Random. The detailed data points are as follows: - For model scale (FLOPs) of  $10^{18}$  end superscript, the performance of LaMDA is 0, GPT-3 is 0, Gopher is not available, Chinchilla is not available, PaLM is 0 and Random is 0. - For model scale (FLOPs) of  $10^{20}$  end superscript, the performance of LaMDA is 0, GPT-3 is 0, Gopher is not available, Chinchilla is not available, PaLM is 0 and Random is 0. - For model scale (FLOPs) of  $10^{22}$  end superscript, the performance of LaMDA is 0, GPT-3 is 8, Gopher is not available, Chinchilla is not available, PaLM is 7 and Random is 0. - For model scale (FLOPs) of  $10^{24}$  end superscript, the performance of LaMDA is 15, GPT-3 is 33, Gopher is not available, Chinchilla is not available, PaLM is 16 and Random is 0.

How do LLMs like Transformers develop their emergent capabilities? These models are structured hierarchically, consisting of multiple processing layers, typically with 6–8 encoders and decoders that represent varying degrees of abstraction. Each layer captures a different level of complexity, ranging from simple patterns to intricate relationships. As the model trains, the output of each layer feeds into the next, with every weight adjusted based on

data through a predefined loss function. This structured approach allows these models to autonomously identify underlying features and patterns in data and adaptively fine-tune their parameters according to statistical trends. As the model's parameters expand and cross a certain threshold, new and unexpected capabilities begin to manifest.

Interestingly, the human brain exhibits a similar hierarchical structure. It contains about 100 billion neurons spread across various layers of the cerebral cortex, each layer interconnected in a hierarchical fashion. The lower layers process basic features like edges and shapes, while the upper layers manage more complex information like objects, faces, scenes, and abstract concepts. Remarkably, the Transformer architecture shares some parallels with the cerebral cortex, which handles functions such as language, thought, perception, emotion, and planning. Both systems are layered, processing inputs at different levels of abstraction and integrating diverse information sources. Furthermore, the Transformer's "memory" function mirrors how the cerebral cortex stores and retrieves information over time to comprehend language and produce coherent thoughts.

The Transformer model boasts a sophisticated attention mechanism, allowing various parts of the network to hone in on the most pertinent information. At each layer, this mechanism calculates a weighted sum of the encoder output vectors, pinpointing the elements most relevant for

subsequent layers. Attention heads are capable of processing segments of the sequence in parallel, much like how humans can attend to multiple information streams at once. Crucially, it can zero in on key inputs and forge connections between disparate concepts. For instance, at a bustling party, while conversing with a friend, your brain can isolate their voice from a cacophony of background noise. It also links the ongoing conversation with related past discussions, demonstrating attention-driven connections between memories.

Moreover, the brain is adept at managing sequential information, such as language, motor skills, tasks, and temporal reasoning. The Transformer, designed specifically for processing sequential data like natural language, approaches a level of proficiency akin to human capability. In the brain, complex recurrent connections between neurons facilitate an “ongoing internal dialog” that sustains contextual awareness. Conversely, the Transformer employs its self-attention mechanism to capture dependencies among elements within the input sequence. Thus, both the brain and the Transformer are equipped with tailored architectures and network components for learning and processing sequential information, though they do so in distinct ways and with different efficacy levels.

While Transformers and the human brain process information in similar ways, Transformers remain quite restricted in their application scope, lacking the robust

functions derived from the brain's biological complexity. The parallels between deep neural networks and the brain offer insights into the fundamentals of natural language processing, yet pinpointing further distinctions between these systems is essential for creating algorithms that mimic human learning and reasoning.

Both Transformers and the human brain exhibit similarities in their hierarchical structure, attention mechanisms, and handling of sequential information, hinting at a shared foundation in their approaches. GPT-4, a landmark model, is theorized to have reached a parameter scale comparable to the number of neurons in the brain, incorporating multimodal training data. This integration heralds the emergence of new multimodal abilities, such as capabilities that combine audio, visual, and text semantics to form a more advanced intelligence, not just an additive combination of these elements. It is speculated that as more modal data is incorporated and the parameter size increases, an even broader range of multimodal capabilities will emerge. Whether the GPT-N series could lead to strong AI and whether humanity's destiny aligns with Elon Musk's view that "carbon-based civilization is merely a precursor to a silicon-based one" are provocative questions for our future. These developments remind us that the future has already begun.

## **11.2 Data resource depletion**

In assessing the potential and future trajectories of GPT-5 and subsequent GPT-N models, it's crucial to focus on three key aspects: the volume of data, the quality of data, and the sources from which the data are drawn. These factors are likely to play pivotal roles in determining whether GPT-5 and its future iterations can match or even exceed human intelligence.

Recent media speculations have begun forecasting the release date for GPT-5, with expectations set for groundbreaking advancements in multimodal processing. Reports suggest that GPT-5 could be trained using approximately 25,000 GPUs. TechRadar has noted that Chachi BT has been trained on 10,000 NVIDIA GPUs, each surpassing the performance of the A100 model. Regarding the timeline for GPT-5 debut, predictions from Geordie Rybass are often referenced; he anticipates the model or its equivalents might launch in late spring or early summer of 2024.

Research from DeepMind has highlighted a critical balance between the size of model parameters and the volume of training data required. Notably, models like GPT-3 and Palm possess far more parameters than necessary, pointing towards a greater need for extensive, high-quality data. Consequently, the claim that GPT-4 requires one trillion parameters may not hold true. In fact, GPT-5 could potentially maintain or even reduce the parameter count compared to GPT-4. A LessWrong blog post from July 2022 suggests that the performance limitations in current



language models are more due to data constraints than the sheer scale of the models. With sufficient data, there's no necessity for models to escalate to 500 billion, a trillion, or more parameters.

The quality of data plays a pivotal role in refining GPT models, yet sourcing high-quality data remains challenging. Currently, GPT-3 and other models train on approximately 300 billion tokens. Considering that DeepMind's Chinchilla model was trained on around 1.4 trillion tokens, it's likely that GPT-5 will significantly expand its data volume. Known sources for high-quality data include scientific articles, books, web-scraped content, news articles, code, and Wikipedia, ranging between about 4.6 trillion to 17 trillion words in total. This indicates we are nearing the limits of available high-quality data, a threshold that may be reached between 2023 and 2027, profoundly impacting the immediate future of AI.

The uncertainty surrounding data sources remains a significant issue. Companies such as Google and OpenAI have kept their data sources under wraps, likely to sidestep disputes related to ownership and compensation. Moreover, as legal complexities escalate in areas like AI-generated imagery, pinpointing the origins of data is becoming increasingly important. Despite these challenges, GPT-5 aims to leverage past experiences to amass a greater quantity of high-quality data. Since the transition of GPT-4 to Microsoft, the repository of high-quality data has consistently grown by approximately 10% per year, even

without enhancements in data utilization or extraction techniques.

Looking beyond just data-related challenges, GPT-5 is poised for breakthroughs across several domains. Researchers might unearth techniques for deriving high-quality data from lesser-quality sources. Furthermore, implementing automated “chain-of-thought prompting” strategies could notably boost the model’s effectiveness. Although constraints related to performance and costs may impede training efforts, recycling data across multiple training sessions has proven to be an effective tactic. Additionally, the strategic creation and curation of datasets are vital for enhancing the model’s capabilities in solving complex mathematical problems and beyond.

If GPT-5 effectively leverages its 9 trillion high-quality tokens, it’s poised for a transformative leap in performance, with potential significant impacts on the job market. GPT-5 could outperform humans in reading comprehension, logical reasoning, critical thinking, and even in subjects like high school physics and mathematics. Moreover, as technologies advance in text-to-speech, image-to-text, text-to-image, and creating virtual avatars from text-to-video, AI teachers could soon become a reality. However, the release of GPT-5 remains uncertain, partly because its launch may hinge on the progress of OpenAI’s internal safety research. Sam Altman, CEO of OpenAI, has indicated that the model will only be released after thorough alignment, safety checks, and collaboration with external auditors.

In this setting, aligning AI safety research with model performance enhancements is essential. While the exact release date of GPT-5 remains unknown, it is poised to introduce disruptive innovations in the future, driven by an influx of high-quality data and further optimization strategies. The evolution of GPT-5 and future models will be shaped by the quality, sources, and strategies employed in data use. Despite prevailing uncertainties, with robust high-quality data and enhanced model training and data utilization strategies, GPT-5 is expected to significantly elevate its performance soon. This advancement could see it excel in fields like reading comprehension, logical and critical thinking, high school physics, and mathematics, potentially reshaping the job market. Nonetheless, it remains crucial to balance technological progression with safety and ethical considerations, ensuring that advances in AI come with necessary safeguards.

## **11.3 Limitations of autoregressive models**

Many experts argue that the inherent limitations of their architecture might prevent autoregressive language models like the GPT series from reaching strong AI simply by scaling up multimodal data and parameters. Scholars, including Yi Ma, have pointed out that autonomous agents need to efficiently learn from past experiences and adapt to current contexts to thrive in complex environments. These capabilities are essential for collecting data, making

decisions, and executing actions. Yet, the AI advancements of the last decade have predominantly been driven by deep learning methods that rely on “brute force” training. While these models gain certain abilities in perception and decision-making, the features they learn are often not easily interpretable.

Furthermore, the overreliance on increasing computational power for training has led to a surge in the size and cost of deep learning AI models, causing various practical issues. These include a lack of diversity in learned representations, poor training stability, and a diminished sensitivity to adaptiveness and catastrophic forgetting, complicating their deployment in real-world settings.

Yi Ma identifies the separation in training discriminative models for classification and generative models for sampling or replay within current deep networks as a fundamental issue. These models are usually open-loop systems that depend on supervised or self-supervised training from start to finish. Yet, it has long been recognized, with early insights from Wiener, that these systems lack the capability to self-correct prediction errors and adapt to environmental changes.

As depicted in **Fig. 11.2**, Yi Ma argues that true general AI must embrace principles of simplicity and self-consistency. The simplicity principle emphasizes extracting and representing information from data in a way that enhances learning efficiency. This approach is deeply intertwined with information encoding theory, which looks

at precise quantification and efficient representation of data information. This principle has robust support in neuroscience, evident in brain functionality such as sparse coding and subspace encoding.

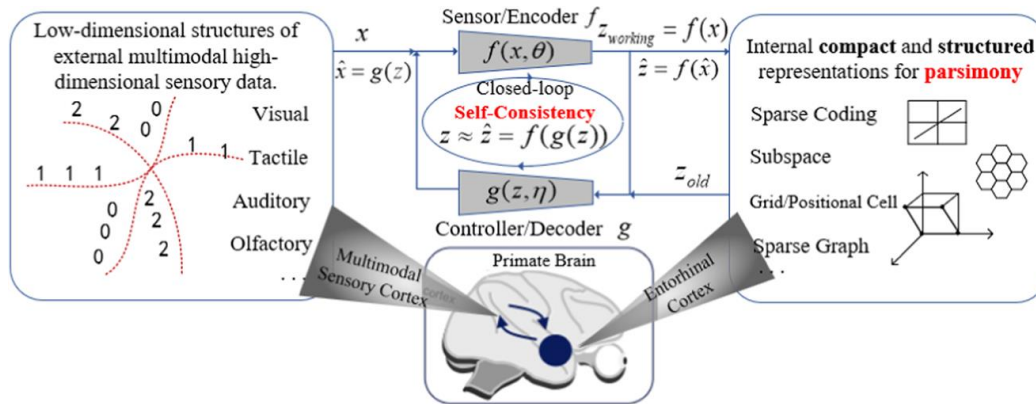


**FIGURE 11.2** Universal machine learning framework. Image source: Paper “On the principles of parsimony and self-consistency for the emergence of intelligence”.

It shows the processing of low-dimensional structures of external multimodal high-dimensional sensory data, including visual, tactile, auditory and olfactory inputs. The sensory data is processed through a sensor/encoder function  $f$  such that  $z_{working} = f(x)$ , resulting in  $\hat{x} = f(x)$ . This is followed by a closed-loop self-consistency process where  $z \approx \hat{z} = f(g(z))$ . The controller/decoder function  $g$  of  $z$  is also shown. The internal compact and structured representations for parsimony include sparse coding, subspace, grid/positional cell and sparse graph. The diagram also highlights the roles of the multimodal sensory cortex and entorhinal cortex in the primate brain.

The simplicity principle in deep learning offers solutions to several pervasive issues, such as neural collapse, mode collapse, and susceptibility to deformations and adversarial attacks. By emphasizing simplicity in both model design and the training process, models can be developed to be more efficient, stable, and adaptable.

The self-consistency principle is about achieving learning goals effectively and efficiently through computation. It is intimately linked to control/game theory, which provides a robust computational framework through closed-loop feedback systems for reaching any quantifiable objective. In deep learning, self-consistency addresses critical challenges like catastrophic forgetting and poor adaptability to changes in the environment. By creating a unified closed-loop system that incorporates both discriminative and generative models, it fosters autonomous learning, which in turn boosts the efficiency, stability, and adaptability of the learning process.

However, autoregressive LLMs like the GPT series, which rely on an extensive array of neurons to learn vast amounts of knowledge and language, consequently see a substantial increase in both size and computational costs. This growth clearly conflicts with the simplicity principle. Additionally, being open-loop systems, GPT models require supervised or self-supervised end-to-end training and lack the ability to autonomously correct errors in predictions.

Turing Award laureate LeCun has expressed skepticism about the future of such models. From a mathematical or

machine learning viewpoint, LLMs essentially map the probabilistic relationships between sequences of words, treating combinations of words in sentences as input vectors to predict the probability distribution of words in subsequent positions. Theoretically, the larger the output sequence, the higher the accumulation of errors, making it challenging for these models to overcome inherent issues such as factual inaccuracies, logical flaws, contradictions, and limitations in reasoning.

LeCun asserts that the cornerstone of future AI research should be the development of “world models.” Unlike existing LLMs, humans and animals exhibit far more efficient learning capabilities. For example, a novice teenage driver can master driving in merely 20 h, while current autonomous driving systems require extensive training on millions or even billions of labeled data points, or millions of reinforcement learning trials in simulated settings. Despite this extensive training, their capabilities still don’t match up to human standards. The key challenges in machine learning today involve mastering world representations and predictive models, reasoning, and planning complex action sequences.

LeCun points out that the real challenge for AI in the coming decades will revolve around crafting architectures for world models and developing effective training paradigms. World model training, a prime example of self-supervised learning, draws its core idea from pattern completion. Predicting future or momentarily unobserved



inputs extends this concept. However, as the world is only partially predictable, a significant issue is representing uncertainty in these predictions. Probabilistic models struggle in continuous spaces, and generative models must account for every world detail. To address this, LeCun has introduced the Joint-Embedding Predictive Architecture, enabling a predictive model to encompass multiple predictions.

In conclusion, despite the successes of LLMs like the GPT series, they confront substantial challenges. For AI to reach a stage of true generality, future research should adhere to the principles of simplicity and self-consistency and delve into developing world models and innovative training paradigms.

## **11.4 Embodied intelligence**

Back in 1950, Alan Turing pioneered the notion of “embodied intelligence.” This concept revolves around the idea that an intelligent entity, like a robot, can dynamically engage with its surroundings. Through sensing, acting, and learning, such agents gather knowledge and bolster their ability to adapt to environmental changes.

### **11.4.1 Challenges of embodied intelligence**

Over the last decade, robotics has seen remarkable strides in perception, representation, and action. Deep learning, for example, now outperforms humans in image classification, while humanoid robots from Boston

Dynamics showcase exceptional limb control and coordination. Despite these breakthroughs, AI continues to grapple with challenges in extracting environmental insights from data, decision-making, and universal learning. The concept of “embodied intelligence” underscores essential skills robots must develop:

**Perception:** Robots need the ability to sense their environment, gathering multimodal data like images, sounds, and tactile information through cameras and sensors.

**Representation:** Robots must be capable of extracting and portraying information from their sensory data to better comprehend and interpret their surroundings.

**Decision-making:** With the information at hand, robots should make informed decisions, such as plotting the best paths in navigation tasks.

**Action:** Robots must execute these decisions, controlling motors or actuators to move or handle objects.

**Learning:** Through ongoing interactions, robots should continually learn, enhancing the efficacy and adaptability of their actions and decisions.

Today’s LLMs, like ChatGPT, have showcased their aptitude for understanding tasks, planning, and leveraging third-party tools, signifying a stride towards general intelligence. Particularly, once these models gain the ability to handle multimodal data such as images, videos, and audio, they will unify skills in representation, decision-making, and learning, essentially equipping robots with a

real “brain.” This capability for perception, sourced from cameras and sensors, combined with action through robotic motion systems, paves the way for embodied intelligence. This evolution is set to drive profound societal changes.

### **11.4.2 PaLM-E**

Google’s PaLM-E project has showcased how powerful LLMs can be in tasks requiring embodied reasoning. PaLM-E, a sophisticated model built solely with a Transformer decoder architecture, undergoes pretraining on multimodal utterances. These utterances blend visual inputs, continuous state estimates, and textual encodings, aligning disparate types of data—images, sensor readings—alongside natural language into a unified vector space.

The unique structure of PaLM-E’s utterances marries text with continuous, multisensory observations, weaving them into intricate multimodal sentences. For instance, consider the query, “What happened between <img 1> and <img 2>?”, where each <img i> represents an image embedding processed by a Vision Transformer. As illustrated in **Fig. 11.1**, PaLM-E generates responses autoregressively, crafting either direct answers or a series of text-based decisions. These outputs are utilized in a variety of practical applications, from continuous robotic maneuver planning to visual question answering and generating captions for images (**Fig. 11.3**).

As depicted in **Fig. 11.4**, the project’s demonstration video illustrates an embodied reasoning task where a robot is tasked with fetching chips from a drawer. This task demands more than just robotic functionality; it requires an understanding of human language and intentions. The robot must also adeptly observe and interact with its surroundings, devising a sequence of subgoals and the steps needed to achieve them.



**FIGURE 11.4** Example of an embodied reasoning task with PaLM-E.

The left image depicts the robotic arm reaching into an open drawer, where a packet of chips is visible. The right image shows the robotic arm having retrieved the packet of chips from the drawer. The background includes a counter with various items such as condiment bottles and disposable cups. The robotic arm is marked with the number 06.

### 11.4.3 ChatGPT for robotics

In contrast to the PaLM-E project, Microsoft's ChatGPT for Robotics (GPTR) operates on a nonmultimodal version of ChatGPT, which does not possess multimodal sensory capabilities. Instead, it relies on auxiliary tools to transform environmental information into textual descriptions that feed into ChatGPT. The process involves crafting prompts that detail task constraints, the weight and size of objects, environmental conditions (like avoiding collisions during

navigation), the current status of the robot and its target, along with examples of how to complete tasks (refer to **Fig. 11.5**).

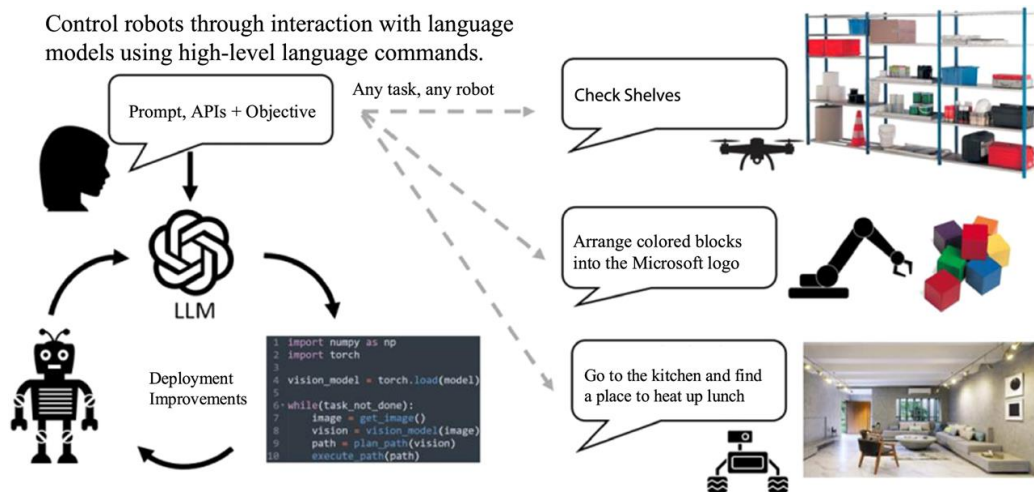


**FIGURE 11.5** Controlling robots through high-level language commands interacting with language models.

The process starts with Prompt, APIs plus Objective which is connected to LLM. From LLM, there are two arrows: one leading to Deployment Improvements and another leading to Any task, any robot. Any task, any robot is connected to three tasks: Check Shelves, Arrange colored blocks into the Microsoft logo and Go to the kitchen and find a place to heat up lunch.

Moreover, the process of breaking down tasks and making decisions in ChatGPT requires managing the robot through textual commands. This demands the encapsulation of the robot's basic action libraries and

descriptions of these functions. With the right prompts in place, ChatGPT is capable of generating the necessary code to accomplish targeted tasks, thereby enabling the control of robotic actions through its foundational action libraries.

GPTR stands out as a cost-effective embodied intelligence solution primarily because it eliminates the need for task-specific fine-tuning of LLMs. Leveraging a broad existing knowledge base and sophisticated reasoning abilities, GPTR can directly produce code that controls robots, dramatically cutting down the time and cost usually required for traditional programming.

In conventional robotic programming, each task demands the creation of bespoke control algorithms—a process that's both time-intensive and reliant on highly skilled engineers. However, GPTR circumvents these hurdles by utilizing a pretrained LLM to swiftly generate effective control code. In doing so, the LLM taps into a wide array of knowledge it has gathered during training, encompassing programming tactics, algorithm design, and optimization techniques, to tackle specific challenges in robotic tasks.

**Fig. 11.6** effectively outlines the four crucial steps GPTR follows to perform tasks under diverse conditions: Initially, it involves encapsulating the functions of the library used to control robotic actions, including movements, stops, and acquiring target location data. Following this, a prompt is constructed that encompasses the task and environmental descriptions along with the library functions. Subsequently, a simulation run is conducted, ensuring continuous

interaction with ChatGPT to guarantee the task's objectives are met. Ultimately, the code is deployed directly onto the machinery.



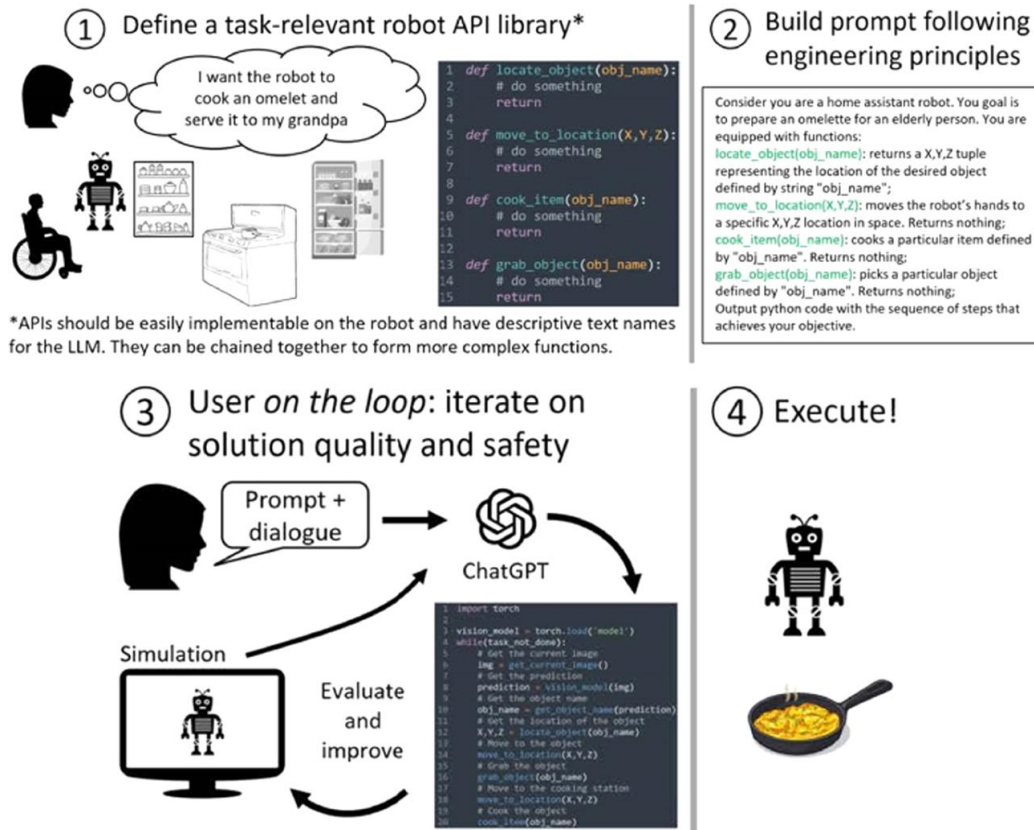


**FIGURE 11.6** Four steps of task execution in ChatGPT for robotics. Image source: Paper “ChatGPT for Robotics: Design Principles and Model Abilities”.

It outlines the four crucial steps GPTR follows to perform tasks under diverse conditions: Initially, it involves encapsulating the functions of the library used to control robotic actions, including movements, stops and acquiring target location data. The steps are as follows: 1. Define a task-relevant robot API library. An example is given where a person wants a robot to cook an omelet and serve it to their grandpa. The API functions include `locateobject`, `movetolocation`, `cookitem` and `grab_object`. APIs should be

easily implementable on the robot and have descriptive text names for the LLM. They can be chained together to form more complex functions. 2. Build prompt following engineering principles. The prompt should consider the robot's goal and the functions available. An example prompt is provided for a home assistant robot to prepare an omelet for an elderly person. 3. User on the loop: iterate on solution quality and safety. This involves using prompt and dialogue with ChatGPT, running simulations and evaluating and improving the solution. 4. Execute! The final step is to execute the task using the robot.

GPTR is adept at handling three distinct types of tasks: The primary task type is zero-shot task planning, where it formulates plans for new tasks without prior specific knowledge. For instance, as depicted in **Fig. 11.7**, in drone inspections, ChatGPT adeptly interprets user intent from inputs, devising viable inspection strategies and patrol routes for drones. This enables even nontechnical users to oversee drone operations effectively.

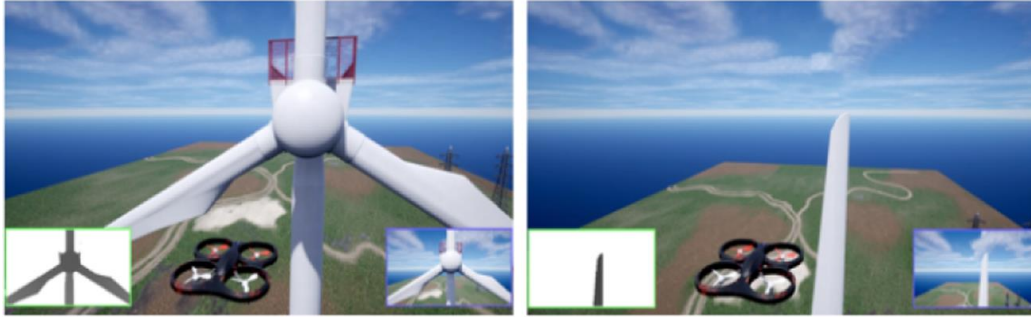


**FIGURE 11.7** Example of zero-shot task planning for drone inspection.

The left section shows a close-up view from the base of the wind turbine looking up towards the rotor hub and blades, with a drone flying nearby. In the inset at the bottom left corner, there is a shadow of the wind turbine on the ground. The right section shows the wind turbine from a distance with a drone flying closer to the viewer. The inset at the bottom right corner shows a side view of the wind turbine blade. Both sections have a backdrop of a coastline and ocean.

Furthermore, ChatGPT acts as a natural language interface between users and robots, making it straightforward for nontechnical individuals to interact with robots and steer them through complex tasks. Yet, in real-world scenarios, robots must manage ever-evolving environmental data and tackle uncertainties. Thus, LLMs like GPTR often require integration with other AI technologies, such as reinforcement learning or computer vision, to foster robust and reliable embodied intelligence.

The second category encompasses complex tasks facilitated by interactive dialog. These tasks require ongoing interactions between the robot and the user to meet sophisticated objectives. Such engagements foster intricate control patterns, allowing ChatGPT to assimilate and amalgamate simple control schemes to tackle bigger, more intricate tasks. Moreover, ChatGPT can translate high-level semantic demands for generating control code into actionable, low-level coding tasks, simplifying interactions for nontechnical users.

As illustrated in **Fig. 11.8**, consider a nontechnical user directing ChatGPT to devise a goal achievement and obstacle avoidance algorithm for a drone equipped with a forward distance sensor. While ChatGPT can compose most parts of the obstacle avoidance algorithm, it still needs user input on the drone's direction to pinpoint any overlooked steps. Despite feedback being in high-level semantic text, ChatGPT can make targeted code adjustments to refine its solution. These instances highlight how ChatGPT progressively refines its solutions through interactive dialogs, thus enhancing performance in complex scenarios. This method is particularly advantageous for nontechnical users, who can guide ChatGPT using natural language without delving into the technical specifics. Nevertheless, to ensure precise comprehension and adjustments by ChatGPT, users must offer distinct and detailed feedback. Sometimes, iterating on ChatGPT's improvements might be necessary to reach optimal outcomes.

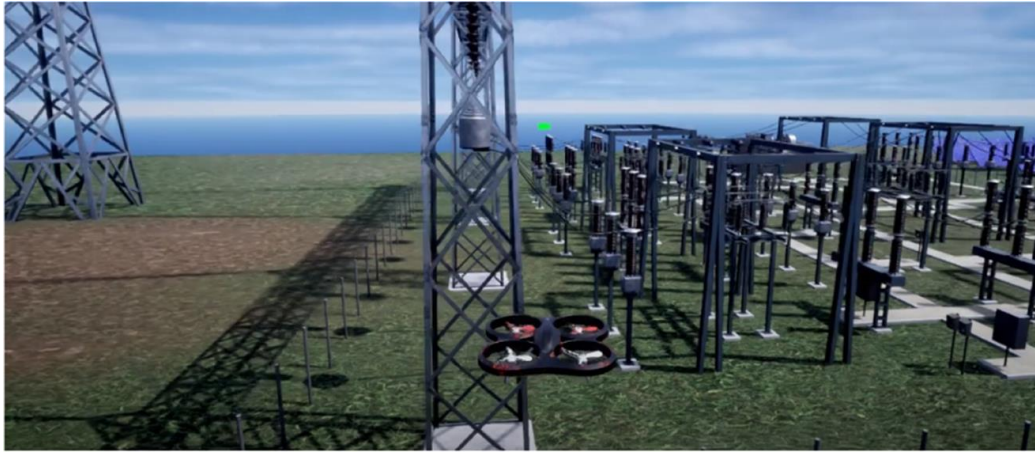


**FIGURE 11.8** Drone obstacle avoidance task.

The course consists of various metal structures, resembling scaffolding, arranged in a pattern. In the background, there is a green square target visible in the distance, which appears to be the drone's destination. The environment suggests an open field with a clear sky and the shadow of a tower is cast on the ground, indicating the presence of a light source in the simulation. The drone is equipped with four rotors and is in flight, navigating through the obstacles. There are no people or other discernible objects in the image.

The third task type involves the perception-action feedback loop, requiring robots to initiate a series of actions based on their environmental perceptions. ChatGPT's abilities in this domain can be gauged from two distinct angles.

Initially, consider how ChatGPT employs API libraries to establish a perception-action feedback loop. In its code

outputs, ChatGPT precisely utilizes perception functionalities such as image capture and object detection to gather essential navigation and control information for robots. This showcases ChatGPT's prowess in handling visual perception data, ensuring efficient navigation and operation in unfamiliar settings.

Furthermore, assess ChatGPT's capability to create a sustainable perception-action feedback loop. For this exploration, researchers continuously feed perception information into the model via text dialogs, first translating environmental data into text before submitting it to ChatGPT. This process reveals that ChatGPT can interpret these data and craft appropriate actions based on them, illustrating its adeptness at translating sensory inputs into actionable responses.

In a recent experiment involving the perception-action feedback loop, ChatGPT was equipped with a computer vision model, part of its functional toolkit, to aid in exploring and navigating toward a user-specified object in unfamiliar environments. Utilizing an object detection API with YOLOv8 backend, it identified object boundaries, while ChatGPT was responsible for generating navigation code, estimating objects' relative angles for approach. When supplemented with depth sensor data, ChatGPT could further refine its algorithm, showcasing its modular adaptability.

As illustrated in **Fig. 11.9**, the setting was constructed using the Habitat simulator, directing the agent to a

designated interest area. This setup tested ChatGPT's role in a perception-action feedback loop through dialog-based interactions. Environmental perception data—descriptions of visible objects in polar coordinates—was fed into ChatGPT via textual dialog. ChatGPT's responses were confined to movement distances and turning angles, which the agent executed, producing fresh environmental data in the process.



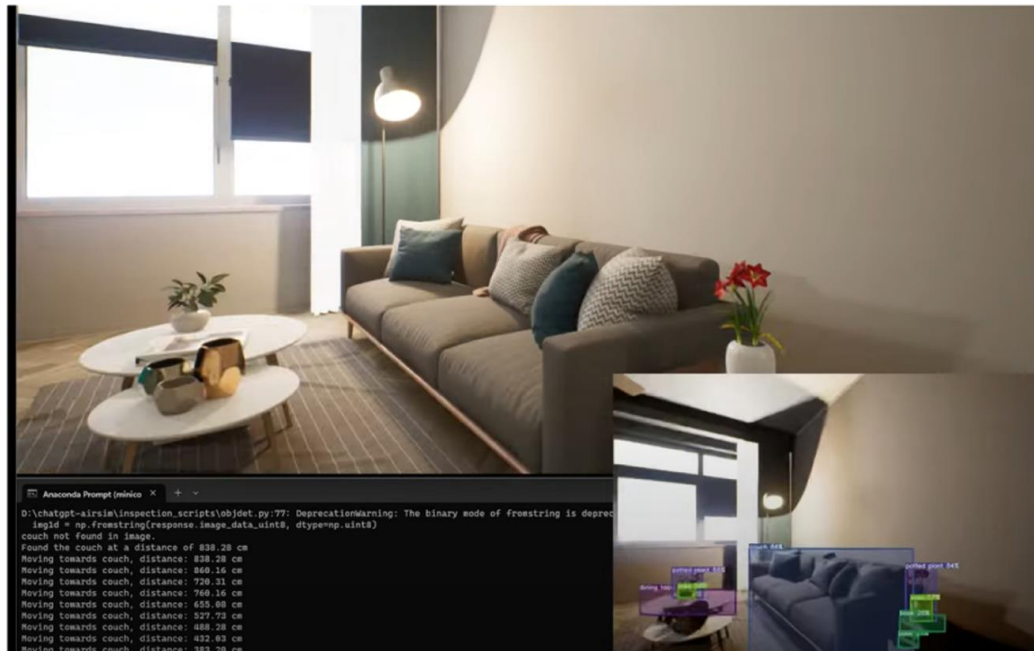


**FIGURE 11.9** Example of the perception-action feedback loop in ChatGPT.

The image displays a simulated indoor setting resembling a living room with a couch, coffee tables and decorative items. In the foreground, there is a command prompt window with multiple lines of text indicating an action Moving towards couch followed by varying distances in centimeters. The background shows a larger view of the simulated room and in the top right corner, there is a smaller inset image with colored boxes around objects such as a potted plant, dining table and couch, likely representing object detection within the simulation environment.

The trial confirmed the model’s capability to handle straightforward navigation tasks. However, it also



highlighted the need for further research into how to precisely depict environmental scenes in text or vector formats for more intricate tasks and settings.

To summarize, the perception-action feedback loop task compels the robot to devise a series of actions driven by its environmental perceptions. Continual input of this environmental data into ChatGPT, coupled with monitoring of its outputs, reveals that the model adeptly interprets observational data and formulates suitable actions accordingly. These findings demonstrate ChatGPT's potential as a control system within a perception-action feedback loop, facilitating effective navigation in complex environments.

In real-world applications, identifying more effective methods to contextualize scenes is crucial to enhance the model's environmental comprehension. Moreover, research must continue into optimizing the efficiency and simplicity with which sensory information is fed into the model. Practical scenarios may also bring additional challenges, including computational resource constraints, the model's real-time performance, and safety and reliability concerns.

Presently, initiatives like Microsoft's GPTR and Google's PaLM-E highlight the potential influence of LLMs in robotics. Despite their limitations in reasoning and task breakdown, such as the granularity of specialized robotic APIs for particular tasks, GPTR relies on manually curated prior knowledge to compile action primitives into skill libraries, leading to issues in design granularity and

overlooking safety in real-world environments. Nonetheless, leveraging LLMs for embodied intelligence could significantly boost existing productivity, marking this as a field of substantial commercial promise.

## **11.5 Summary**

With the unveiling of GPT-4 and the ongoing development of GPT-5, AI is progressing into a groundbreaking new phase. Future iterations of the GPT series are expected to push the boundaries of knowledge-intensive tasks, utilizing even larger scales of multimodal data and increasingly complex models. Despite these advancements, current LLMs exhibit limitations, particularly in complex reasoning tasks. For instance, their ability to process simple character copy reasoning or arithmetic diminishes significantly with longer strings or numbers. Moreover, their capacity for intricate behavior planning remains somewhat underdeveloped. Yet, as the scale of these models increases, they start to exhibit predictable scale-dependent behaviors.

This situation prompts two crucial inquiries: To what extent can scaling up LLMs address these challenges? And considering the emergent capabilities of these models, what unforeseen abilities might develop with further increases in scale and more diverse multimodal data integration? While there are critiques of the GPT research direction, the abilities displayed by GPT-4—from understanding across various modalities and domains to its

potential in intelligent agency and task planning in robotics—signal the dawn of a new era in general AI. The future has arrived, and embracing it is our only option.

---

# Index

---

*Note:* Page numbers followed by “*f*” and “*t*” refer to figures and tables, respectively.

## A

Action probability distribution, [124-125](#)

Action-value function, [127](#), [128](#)

Actor-Critic algorithm, [128-130](#), [128f](#)

- adaptive exploration/exploitation, [130](#)

- advantage function estimation, [130](#)

- dual learning process, [130](#)

- dual-network training, stability of, [130](#)

- exploration strategy issues, [130](#)

- hyperparameter tuning challenges, [130](#)

- sample complexity issues, [130](#)

- TD error as learning signal, [130](#)

Actor formulates policies, [128](#)

Adafactor optimizer, **120**

AdaGrad, **42**

Adam, **42**

Adam optimizer, **22, 116**

Adaptive Moment Estimation, **42**

Adept AI, **186**

Advantage function, **127-128**

Agent, **180f**

Agent-environment interaction, **125f**

AgentGPT, **182**

AGI, *See* **Artificial general intelligence**

All-Reduce operation, **118**

Alpaca model, **170, 170f**

AlphaGo's mastery, **4**

Alternating dense and sparse attention patterns, **83**

Annotation standards, **140-142**

Annotators, **140, 141-142**

Apple's Siri, **1**

Artificial general intelligence (AGI), **3-4**

Artificial intelligence—ChatGPT, new milestone in  
capability level of, **3-6**

development history of, **1-3**

large language model

    impact of, **13-16**

    limitations of, **19-20**

    technology stack of, **12-13**

large language models, technical evolution of, **6-11**

    human feedback reinforcement learning, **11**

    supervised fine-tuning, **10-11**

    symbolism *versus* connectionism, **6-7**

    transformer, **7-9**

    unsupervised pretraining, **9-10**

training/deploying large models, challenges of, **16-19**

    computational power, **16-17**

    data, **17-18**

    engineering, **18-19**

ArXiv, **100**

Attention heads, **29, 189**

Attention score matrix, **23-24, 24-25**

    word embeddings weighted by, **27f**

Attention weight matrix, **26, 26f**

AutoGPT, **15f, 181-183, 182, 182f**  
Autonomous ChatDoctor, **172f**  
Autoregressive language model, **51, 52**  
Autoregressive models, limitations of, **192-195**  
Autoregressive prediction, **76**  
Autoregressive process, **76-77, 77f**  
Award laureate LeCun, **194**  
Azure Search, **177**

## **B**

Baby AGI, **182**  
Baseline estimator, **135**  
Base model, **169-170**  
Bayesian framework, **94**  
Bayesian inference, **92-94**  
Bayesian inference of concept distributions, **88-94**  
BBPE, *See* **Byte-level byte-pair encoding**  
BERT, **61**  
BF16, *See* **bfloat16**  
bfloat16 (BF16), **120-121, 120-121**  
BLOOM, **119**

BLOOM-176B, **101-102**

Boston Dynamics showcase, **195-196**

BPE, *See* **Byte-pair encoding**

BPE algorithm, **69**

Brute force training, **192-193**

Byte-level byte-pair encoding (BBPE), **70, 79-81**

Byte-pair encoding (BPE), **66**

## **C**

CAI, *See* **Constitutional artificial intelligence**

Calculation methods, **27f**

Callback module, **179**

Chachi BT, **190-191**

Chain-of-thought (CoT), **94, 102, 191-192**

Channel inference, **90-91**

Character merging examples, **68t**

Character occurrences, **68t**

Character pairs, frequency counting of, **66**

ChatDoctor, **171, 171f**

ChatGPT, **144**

ChatGPT for robotics (GPTR), **197-203**



- ChatGPT plugins, **185**
- Chinese corpora, **98**
- Chinese corpus, **17-18**
- Clipped PPO objective function, **133-134**
- Clipping objective function, **133**
  - graph of different values, **134f**
- Closed-loop feedback systems, **194**
- Code analyses, **183**
- Code-cushman-001, **137-138**
- Code-davinci-002, **139**
- Codex, **18, 137**
- Coefficient matrix, **26**
- Combined position encoding, **40**
- Common Crawl, **99-101, 100**
- Common-sense knowledge, **9-10**
- Communication operations, **115f**
- Computational challenges, **103**
- Computational hardware, **12-13**
- Computational power, **16-17**
  - 3.14E23 FLOPS, **16-17**
  - 758 A100 GPUs, **16-17**

- 1000 A100 GPUs, **16-17**
- 2048 Nvidia A100 GPUs, **16-17**
- inference and deployment stage, **17**
- NVIDIA 80GB A100 GPUs, **16-17**
- pretraining and fine-tuning stage, **16-17**
- Computational resource constraints, **203**
- Computational resources, **88**
- Computer science, **73**
- Constitutional artificial intelligence (CAI), **161-162**, **161f**
- Context-focused head, **30**
- Contextual learning, **89**, **92**
- Contextual learning *versus* traditional fine-tuning, **86f**
- Conventional robotic programming, **198-199**
- Correct labels, **90-91**
- Cosine annealing, **43**
- Cosine functions, **22**, **38**, **41**
- CoT, *See* **Chain-of-thought**
- Covariate shift, **64**
- Cross-entropy, **22**, **41-42**
- Cross-layer sparse connections, **84**

## D

DAG, *See* **directed acyclic graph**

Data augmentation method, **44**

Data parallelism, **103-105**, **104f**, **107-109**, **109**

Data resource depletion, **190-192**

Davinci API, **137**

Decoder, **36**, **61-62**, **62f**

    component, **8-9**

Deep learning, **195-196**

    models, **6-7**

DeepMind, **191**

    Chinchilla model, **191**

Deep neural network training, **35**

DeepNorm, **120**

Deep residual networks, **35**

Deepspeed, **13**, **115-116**, **119-120**

Delimiter, **57**

Dense connections, **84**

Dense layers, **163**

Dialog mode, **178**

Dialogue generation, **10-11**

Digital landscape, **69**

Dimension d\_model, **39**

Directed acyclic graph (DAG), **110-111**

Direct inference, **90-91**

Distributed training, **109**

    patterns, **103-109**

Divide-and-conquer approach, **97-98**

Downstream tasks, **90f**

Drone obstacle avoidance task, **201f**

Dropout regularization, **43, 43f, 44**

Dropout strategies, **31-32**

Dropout technique, **54**

Dust and Scale AI Spellbook, **184**

## **E**

Early stopping method, **44**

ELIZA, **1**

Embedding mode, **178**

Embodied intelligence, **195-203**

    action, **196**

    decision-making, **196**

- learning, **196**
- perception, **196**
- representation, **196**
- Encoder, **36, 61-62, 62*f***
  - decoder architecture, **8-9, 37**
  - stack, **8-9**
- Entropy loss, **133**
- Ever-evolving environmental data, **200**
- Exponential decay, **43**
- Extensive corpus statistics, **9**
- Extensive labeled data, **88**
- Extensive model expansion, **8**

## **F**

- Facebook's FairScale, **110**
- Factual knowledge, **9-10**
- Feedforward network module, **8-9**
- Feedforward neural network (FNN), **21, 32-34**
  - input layer, **32-33**
  - linear layer, **33**
  - output layer, **32-33**

Few-sample tasks, **140**

Few-shot learning ability, **86, 87-88, 89, 92-93**

Fiction genres, **100**

Filtering process, **160-161**

Fine-tuning, **72, 93-94**

- datasets, sources of, **140**
- phase, **50, 55**
- process, **75, 86, 89f**
- steps, **57-58**

Floating-point formats, **121f**

FNN, *See* **Feedforward neural network**

FNN sublayer, **36**

Foundational Transformer study, **30**

FP16, **120-121**

FP32, **120-121**

Full-order ranking approach, **150**

## **G**

Gated recurrent units (GRUs), **6-7**

Gaussian distribution, **164**

Gaussian Error Linear Unit (GELU), **33**

GELU, *See* **Gaussian Error Linear Unit**

GELU activation function, **34f**

Generalization enhancement, **34**

Generative pretraining (GPT), **47-48, 48-51**

models, **56f**

Generative pretraining process, **51-54**

error backpropagation process in, **53-54**

objectives of, **51-52**

GitHub, **100, 101-102, 169**

GLaM, **187-188**

Google Brain team, **2, 94**

Google's Pathways, **110**

asynchronous distributed data flow, **111f**

GPT-2, **2, 61-62, 63-71, 63f**

autoregressive generation process of, **74-77**

autoregressive process, **76-77**

layer normalization, **64**

learnable relative positional encoding, **70-71**

model architecture of, **64f**

orthogonal initialization, **65-66**

reversible tokenization, **66-70**

subword unit embeddings, **75-76**

GPT-3.0, **99-101**

GPT-3.5, **99-101**

GPT-3 model architecture, **80f, 187-188**

dataset used for training, **100t**

embedding layer, **81**

input and BBPE tokenization, **81**

layer normalization, **81**

output layer, **81**

positional encoding, **81**

training dataset, **18f**

transformer layers, **81**

GPT-3 SET model, **150-151**

GPT-3 to GPT-3.5

technology roadmap from, **138f**

GPT-4, **191**

GPT-4 balloon image multimodal, **3f**

GPT-4 intelligence

autonomous learning and reasoning, **6**

generalization ability, **5**

knowledge understanding and application, **5**



- task performance, **5**
- GPT-4 with human examination levels, **5f**
- GPT-5, **191**
- GPT-Index, **184**
- GPTQ quantization technique, **167-168**
- GPT, *See* **Generative pretraining**
- GPT series evolution timeline, **2f**
- GPT series model architecture, **9f**
- GPU kernels, **167-168**
- GPU's VRAM, **118**
- Gradient accumulation, **45**
- Gradient calculations, **127**
- Gradient clipping techniques, **22, 44**
- Gradient descent method, **53**
- Gradient issues, **64**
- Gradual magnitude pruning, **167**
- GRUs, *See* **Gated recurrent units**
- GSM8K, **95**

## **H**

- Habitat simulator, **202**

HealthCareMagic, **171**

Hessian matrix, **132**

HFRL, *See* **Human feedback reinforcement learning**

Higher-ranked response, **148-149**

High-frequency character pairs, **66**

High-level language commands, **198f**

High-speed Ethernet, **112**

HuggingGPT, **182**

Human brain, **189**

Human-computer interfaces, **14**

Human feedback reinforcement learning (HFRL), **11**

- in ChatGPT, **137-139**

- InstructGPT training dataset, **139-144**

  - annotation standards, **140-142**

  - data analysis, **142-144**

  - fine-tuning datasets, sources of, **140**

- multiturn dialogue capability, **152-153**

- necessity of, **153-155**

- PPO in InstructGPT, **150-152**

- reward modeling algorithms, **148-150**

- loss function, **149-150**
- reward scores, **148-149**
- training stages of, **144-148**
  - reinforcement learning, **147-148**
  - reward modeling, **146-147**
  - supervised fine-tuning, **145-146**
- Human-generated annotations, **157**
- Hyperparameter adjustment, **58**

## **I**

- Image-to-text, **192**
- Implicit fine-tuning, **88-92**
- Importance sampling, **126-127**
  - in policy gradient methods, **126f**
- Improper initialization, **65**
- In-context learning, **85-88, 86-88**
- Indexing module, **179f**
- Inference and deployment stage, **17**
- InfiniBand, **112**
- Inflection AI, **186**
- Initial K-1 pairs, **140**

Initial supervised fine-tuning, **11**

Input-first instance generation, **159-160, 160f**

Input-label mappings, **90**

Input layer, **32-33**

Input matrix, **27**

Input sequence, **23**

Instance generation, **159-160**

Instruct-davinci-beta, **138**

InstructGPT, **3, 99-101, 144, 150**

- reinforcement learning process of, **151f**
- reward model for multiturn dialogue in, **152f**

InstructGPT training dataset, **139-144**

- data distribution of, **143t**

Instruction-following model, **170-171**

Instruction generation, **158, 159f**

INT8 integers, **166**

INT32 outputs, **166**

Interactive dialog, **200**

Inter-GPU communication, **114**

Intricate language patterns, **87**

Inverse Hessian sequence, **167**

Iterative merging process, [67](#)

IT ticketing systems, [177](#)

## **J**

Jason Wei, [94](#)

Joint-Embedding Predictive Architecture, [195](#)

Judicial field, [173-175](#)

JusticGPT, [173](#), [174](#)

## **K**

Kaiming He, [35](#)

Key vector, [27-28](#)

Keyword mining approach, [172](#)

KL divergence, [93](#), [131](#), [131f](#)

## **L**

L1 regularization method, [44](#)

Label smoothing technique, [45](#)

LAMBADA language modeling task, [74](#)

LaMDA, [95](#), [187-188](#)

LangChain, [177-181](#), [181t](#)

Large language model (LLM), **12-13**, **12f**, **91**

accuracy, **19**

application layer, **13**

authenticity, **19**

capabilities of, **14-15**

distributed training patterns, **103-109**

data parallelism, **103-105**

model parallelism, **105-109**

emergent capabilities of, **188f**

ethical implications, **19-20**

framework layer, **13**

infrastructure layer, **12-13**

interpretability, **19**

medium-sized enterprises, **15-16**

middleware based on, **15-16**, **15f**

middleware layer, **13**

model layer, **13**

open-source data for, **174t**

pre-training datasets, **99-101**

privacy concerns, **20**

processing of pretraining data, **101-103**

small-sized enterprises, **15-16**

social implications, **19-20**

technical approaches, **109-119**

    Megatron-LM, **113-115**

    pathways, **110-113**

    ZeRO, **115-119**

training of, **17**

training dataset for, **101*t***

training strategies, examples of, **119-122**

    BF16, **120-121**

    optimizing training settings, **120**

    other factors, **121-122**

    parameter stability, **120**

    training framework, **119-120**

as universal interface, **14*f***

Large language models, future path of

    autoregressive models, limitations of, **192-195**

    data resource depletion, **190-192**

    embodied intelligence, **195-203**

        challenges of, **195-196**

        ChatGPT for robotics, **197-203**

PaLM-E project, **196-197**

path to strong artificial intelligence, **187-190**

Large language models, low-resource domain transfer  
of

case studies, **169-175**

base model, **169-170**

instruction-following model, **170-171**

judicial field, **173-175**

medical field, **171-173**

constitutional artificial intelligence, **161-162**

low-rank adaptation, **163-165**

choice of rank, **164-165**

model training and deployment, **163-164**

quantization, **165-166**

self-instruct, **157-161**

filtering, **160-161**

instance generation, **159-160**

instruction generation, **158**

task classification identification, **158**

SparseGPT, **166-169**

LawGPT\_zh, **173, 175f**



Lawyer LLaMA, **173**

Layered multihead attention, **32**

Layer normalization, **22, 36-37, 64**

LayerNorm, *See* **Layer normalization**

Learnable relative positional encodings, **70-71, 83**

Learned position encoding, **40**

Learning rate adjustment strategy, **42-43**

Learning rate warmup strategy/techniques, **22, 44, 45f**

LessWrong blog post, **191**

LexiLaw, **173**

LibGen18, **100**

Linguistic knowledge, **9-10**

Lingvo, **110**

Linux operating systems, **4**

LLaMA with parameter scales, **169t**

LLM-generated instruction, **158**

Local banded attention, **83-84**

- build the attention matrix, **83**
- determine window size, **83**
- execute local banded attention, **83**

Local banded sparse connections, **84**

Local information extraction, **34**

Local multihead attention, **32**

Long-range dependencies, **83**

Long short-term memory networks (LSTMs), **6-7, 28**

Long-term memories, **181**

LoRA, *See* **Low-rank adaptation**

Loss function, **41-42, 54-55, 149-150**

Low-rank adaptation (LoRA), **163-165**

Low-rank decomposition, **164**

LSTMs, *See* **Long short-term memory networks**

## **M**

Machine learning (ML), **110-111**

Machine learning phase, **6, 19f**

Manual annotations, **157**

Masked self-attention, **9, 10f, 47, 50f**  
    mechanism, **62**

Mask matrix, **48-49**

Mathematical vectors, **187**

Matrix computation process, **25f**

Matrix multiplication across multiple GPUs, division of, **106f**

Matrix multiplications, **113**

Medical field, **171-173**

Megatron-LM, **13, 113-115**

Megatron-Turing NLG, **187-188**

Memory function, **189**

Memory wall, **109**

Meta-learning, **85-88, 85**

Meta LLaMA model, **16-17, 103**

Meta-optimization process, **89f**

Metaoptimizers, **88-89**

Meta's LLaMA 7B, **170**

Meta's Toolformer, **186**

Microsoft's DeepSpeed ZeRO, **110**

Microsoft's New Bing, **13**

Microsoft's PipeDream, **110**

Microsoft's XiaoIce, **1**

Middleware

- AutoGPT, **181-183**

- competitors in, **183-186**

LangChain, **177-181**

Mixed precision acceleration ratio and training loss, **45f**

Mixed-precision decomposition strategy, **165-166**

Mixed precision training iteration/method, **45, 116f**

ML, *See* **Machine learning**

Moats, **14-15**

Model application, **58**

Model fine-tuning, **58**

Model-generated probability distribution, **51**

Model gradients, **116**

Model initialization, **58**

Model parallelism, **105-109, 105f**

Model parameters, **116**

Model performance evaluation, **58**

Model postpruning, **167**

Model's real-time performance, **203**

Model training and deployment, **163-164**

Monte Carlo method, **125, 129**

MTL, *See* **Multitask learning**

Multihead attention mechanism, **21, 28-32, 29f, 37**

- visualization of, **31f**
- weighted fusion of, **30f**
- Multihead attention module, **8-9**
- Multilayer perceptron, **32-33**
- MultiNLI datasets, **164-165**
- Multiple-choice tasks, **90-91**
- Multiple-choice token, **57**
- Multiple decoder components, **76**
- Multitask and zero-shot learning, **72-74**
- Multitask learning (MTL), **61**
- Multitask supervised learning concept, **71f**
- Multitask unsupervised learning concept, **71f**
- Multiturn dialogue capability, **152-153**

## **N**

- National Institutes of Health, **100**
- Natural language inference (NLI), **56-57**
- Natural language processing (NLP), **2, 21, 47, 138**
- Natural language text, **72-73**
- Negative gradients, **134**
- Neural network operations, **65**

Neural network parameters, **124**

Neural networks, **107**

Neuron's output, **36-37**

Next-generation human-computer interactions, **183-184**

2048 Nvidia A100 GPUs, **103**

NLI, *See* **Natural language inference**

NLP, *See* **Natural language processing**

NLP tasks, **67**

Noam scheduling, **42**

Nonfiction genres, **100**

Nonlinearity introduction, **34**

Nonlinear transformation, **35**

Nonprofit AI research institute, **1**

Normalization types, **65f**

Numerical stability, **37**

nuQmm, **167-168**

NVIDIA's Megatron-LM, **110**

NVIDIA's V100, **121**

NVMe protocol, **119**

## O

One-dimensional sharding, **105-106**

One-hot encoding, **26**

One-shot learning ability, **86, 87**

Ongoing internal dialog, **189-190**

Online learning, **148**

OpenAI, **1**

Open-source strategy, **110**

OPT, **119**

OPT-175B, **101-102, 102, 165**

Optimization, **41-46**

- objectives, **131-132**

- techniques, **28**

Optimizers, **42, 64**

Optimizing training settings, **120**

Orthogonal initialization, **65-66**

Orthogonal matrices, **65**

Output-first instance generation, **159-160, 160f**

Output layer, **32-33**

Output matrix, **27**

Output vector, **30**

## **P**

PaLM, **95, 101, 119, 119-120**

PaLM-E project, **196-197, 197*f*, 197*f***

Parallel computations, **107**

Parallel computing capabilities, **8**

Parallelism types, **108*t***

Parameter stability, **120**

Pathway, **13**

Pathways features

- architecture, **111**

- asynchronous execution, **110**

- centralized scheduling, **112**

- DAG, **111-112**

- distributed processing, **110**

- fault tolerance, **110-111**

- flexibility, **110**

- island accelerator subsets, **112**

- overview of, **111*f***

Perception-action feedback loop, **201, 202, 202*f***

Piece-wise constant strategy, **43**

Pipeline bubbles, **107, 107-109**



- Pipeline parallelism, **106, 107-109, 109**
- Pipeline parallel training, **107f**
- PM, *See* **Preference model**
- Policy gradient methods, **124, 125, 128**
- Positional encoding, **22, 153**
- Positional signals, **39-40**
- Position-based feedforward networks, **37**
- Position embedding matrix, **52**
- Position encoding, **37-41**
  - design and implementation of, **38-40**
  - workflow of, **38f**
- Posterior probability distribution, **93**
- PPO, *See* **Proximal policy optimization**
- PPO dataset, **142-143**
- PPO in InstructGPT, **150-152**
- PPO model and SFT model, comparison, **154f**
- Preference model (PM), **162**
- Pre-trained language models, **91**
- Pretraining data
  - de-duplication of, **102**
  - diversity of, **102**

- quality and quantity of, **101**
- Pretraining mechanism, **9-10**
- Private clouds, **12-13**
- Process parallelism, **118f**
- Project Gutenberg, **100**
- Projection value, **25-26**
- Prompt template library, **178f**
- Proximal policy optimization (PPO), **150-151**
  - Actor-Critic algorithm, **128-130**
    - algorithm steps, **129**
    - challenges, **130**
    - issues, **130**
    - value function and policy update, **130**
  - principles of, **132-135**
  - traditional policy gradient methods, **123-128**
    - advantage function, **127-128**
    - importance sampling, **126-127**
    - principles of, **123-125**
  - trust region policy optimization, **131-132**
    - limitations, **132**
    - optimization objectives, **131-132**

Pruning technique, **166-167**

PTD-P, **115**

Public clouds, **12-13**

Pushshift.io Reddit, **102**

Python code, **137-138**

PyTorch, **13, 113**

## **Q**

QR decomposition, **66*f***

Quantization, **165-166**

Query vector, **27-28**

Question-answering task, **10-11, 55-56**

## **R**

Random labels, **90-91**

Reasoning chains, **97*f***

Rectified Linear Unit (ReLU), **33**

Recurrent neural networks (RNNs), **2, 21**

Reference policy, **126**

Regularization coefficient/techniques, **43-44, 54-55**

Regularization term, **54-55**

Reinforced learning from AI feedback (RLAIF), **161**

Reinforcement learning (RL), **161**

Reinforcement learning from human feedback (RLHF), **138**

Reinforcement learning mechanism, **147-148, 147f**

Reinforcement learning with human feedback, **11f**

Relative positional encodings, **70**

Relative position encoding, **40**

ReLU, *See* **Rectified Linear Unit**

Reparameterization method, **163f**

Representational state transfer (REST), **177**

Residual connections, **22, 35-36, 37**

Reversible tokenization, **66-70**

Reward model (RM), **11**

Reward modeling algorithms, **146-147, 146f, 148-150**

Reward modeling phase, **139-140**

Reward modeling ranking algorithm, **149f**

Reward scores, **148-149**

RLAIF, *See* **Reinforced learning from AI feedback**

RL-CAI model, **162**

RLHF, *See* **Reinforcement learning from human feedback**

RLHF fine-tuning, **154-155**

RL, *See* **Reinforcement learning**

RM, *See* **Reward model**

RM dataset, **142**

RMSprop, **42**

RNNs, *See* **Recurrent neural networks**

## **S**

San Francisco, **1**

Sci-Hub, **100**

Self-ask with search, **180f**

Self-attention and masked self-attention, differences, **63f**

Self-attention mechanism, **7, 21, 23-28, 24f, 82**

calculation process of, **23-25**

Self-attention module, **164-165**

Self-attention sublayer, **48**

Self-consistency principle, **194**

Self-instruct, **157-161**

Self-instructed annotation method, **157, 158f**

Semantic Kernel project, **184, 184f**

Semantic Textual Similarity Benchmark (STS-B), **74**

Semistructured sparsification, **168-169**

Sentiment analysis, **10-11, 74**

SerpApi, **179**

SFT, *See* **Supervised fine-tuning**

SFT dataset, **142**

SGD, *See* **Stochastic Gradient Descent**

Short-term memories, **181**

Silicon Valley figures, **1**

Similarity marker, **57**

Simple tasks, **140**

Sine functions, **22, 38, 41**

SL-CAI models, **162**

SL, *See* **Supervised learning**

Small-scale models, **14-15**

Smashwords, **100**

Softmax normalization, **23-24**

Sparse attention and content-based learning

- Bayesian inference of concept distributions, **88-94**

- implicit fine-tuning, **88-92**
- GPT-3, **79-81**
- in-context learning, **85-88, 86-88**
- meta-learning, **85-88, 85**
- sparse transformer, **81-84**
  - characteristics of, **82-83**
  - cross-layer sparse connections, **84**
  - local banded attention, **83-84**
- thought chains, **94-98**
- Sparse attention patterns, **82-83**
- SparseGPT, **166-169, 167f**
- Sparse multihead attention, **32**
- Sparse transformers, **30, 79-81, 82f**
- Special classification token, **56**
- Specialized training process, **85**
- Special separator, **55-56**
- StackExchange, **100**
- Standard prompts *versus* reasoning chain prompts, **97f**
- Stanford University's Alpaca project, **171**
- State-of-the-art models, **104f**
- State-value function, **128**

Step-by-step problem-solving accuracy, **97-98**

Step-by-step reasoning concepts, **95**

Stochastic Gradient Descent (SGD), **42, 47-48**

Stores 3TB, **17-18**

STS-B, *See* **Semantic Textual Similarity Benchmark**

Subsequent feedforward sublayer, **48**

Subword unit embeddings, **75-76**

Supervised fine-tuning (SFT), **10-11, 54, 54-58, 58, 139-140, 145f, 170-171**

- fine-tuning steps, **57-58**
- principles of, **54-55**
- for specific tasks, **55-57**

Supervised learning (SL), **161**

Symbolism *versus* connectionism, **6-7**

- deep learning, **6**
- feature engineering, **6**
- generalization abilities, **6**
- insufficient parallel computing capability, **7**
- internal gating mechanisms, **7**
- limited robustness, **6**
- long-range dependency issues, **7**



- machine learning, **6**
- model capacity limitations, **7**
- Syntax-focused head, **30**

## **T**

- Target policy, **126**
- Target task selection, **57**
- Task classification identification, **158**
- Task-specific input transformations, **57**
- TD, *See* **Temporal difference**
- TechRadar, **190-191**
- Temporal difference (TD), **129**
- TensorFlow, **13, 113**
- Tensor parallelism, **105-106, 107-109**
- Text classification, **56**
- Text-davinci-001, **138**
- Text-davinci-003, **139**
- Text mode, **178**
- Text summarization task, **73**
- Text-to-image, **192**
- Text-to-speech, **192**

Thought chains, **94-98**

Token embedding, **75f**

Traditional absolute positional encodings, **70**

Traditional fine-tuning methods, **13-14**

Traditional RNNs, **6-7**

Traditional transformer, **82f**

Trained reward model, **151**

Training data preparation, **58**

Training framework, **119-120**

Trajectories, **124**

Transfer learning scenarios, **69**

Transformer, **7-9, 8f, 164-165**

Transformer-based decoder structure, **81**

Transformer block, **114f**

Transformer decoder, **48f**

Transformer model, **2, 21-22, 49f, 165t**

- feedforward neural network, **32-34**
- layer normalization, **36-37**
- learning rate adjustment strategy, **42-43**
- loss function, **41-42**
- multihead attention mechanism, **28-32**

- implementation of, **29-30**
- optimization of, **32**
- role of, **30-32**
- optimization, **41-46**
- optimizer, **42**
- position encoding, **37-41**
  - advantages and limitations of, **41**
  - design and implementation of, **38-40**
  - variants of, **40**
- regularization, **43-44**
- residual connection, **35-36**
- self-attention mechanism, **23-28**
  - advantages and limitations of, **28**
  - calculation process of, **23-25**
  - essence of, **25-28**
- training, **41-46**
- Transformer's decode architecture, **187-188**
- Transformer's self-attention mechanism, **153**
- Transformer stack layer, **52**
- TRPO, *See* **Trust region policy optimization**
- Trust region policy optimization (TRPO), **131-132**

- computational complexity, **132**
- handling discrete actions, **132**
- hyperparameter tuning, **132**
- need for second-order derivatives, **132**

TruthfulQA dataset, **154**

## U

Underutilization, **107**

Unidirectional language model, **9**

Uniform gradient scaling, **64**

Universal machine learning framework, **193f**

Unstructured sparsification, **168-169**

Unsupervised learning, **54**

Unsupervised multitask learning, **71-72**

Unsupervised multitask learning, **71-72**

Unsupervised pretraining phase, **9-10, 52, 85**

Use cases based on user data, **141t**

## V

V100 NVIDIA GPU, **103**

Value estimation loss, **133**

Value vector, [27-28](#)  
Vector space, [25-26](#)  
Vector-wise quantization, [165-166](#)  
Virtual avatars, [192](#)  
Vision transformer, [196-197](#)  
VisualGPT, [182](#)  
Vocabulary initialization, [66](#)  
Voting mechanism, [95-97](#)  
VRAM consumption, [107-109](#)  
VRAM fragmentation, [118](#)  
VRAM limitations, [103](#)

## W

Weak AI, [4](#)  
WebText, [2](#), [63](#), [100](#)  
WebText2, [99-101](#)  
WebText dataset, [71](#)  
Weight decay, [43](#), [54](#)  
Weight initialization strategy, [44](#), [65](#)  
Well-optimized cluster, [115](#)  
Wifi, [162](#)

Wikipedia, **99-100, 99-101, 169**

WikiSQL, **164-165**

Wolfram tools, **185, 185*f***

World knowledge, **9-10**

World models, **194-195**

World model training, **195**

WuDaoCorpora, **17-18**

## **X**

Xiangtan University's Smart Justice research team, **173**

Xuezhi Wang, **94**

## **Y**

Yi Ma, **193**

YOLOv8 backend, **201-202**

## **Z**

ZeRO-1, **117**

ZeRO-2, **117**

ZeRO-3, **117**

ZeRO-Infinity, **119**

ZeRO-Offload technique, **118, 119f**

ZeroQuant, **167-168**

Zero Redundancy Optimizer (ZeRO), **115-119**

communication volumes, **117f**

memory savings, **117f**

three-tiered system, **117**

ZeRO, *See* **Zero Redundancy Optimizer**

Zero-shot learning, **61, 72-74**

Zero-shot learning ability, **61, 76, 86, 87**

Zero-shot learning CoT, **94f**

Zero-shot reasoning chains and zero-shot learning, comparison, **96t**

Zero-shot task planning, **199-200**