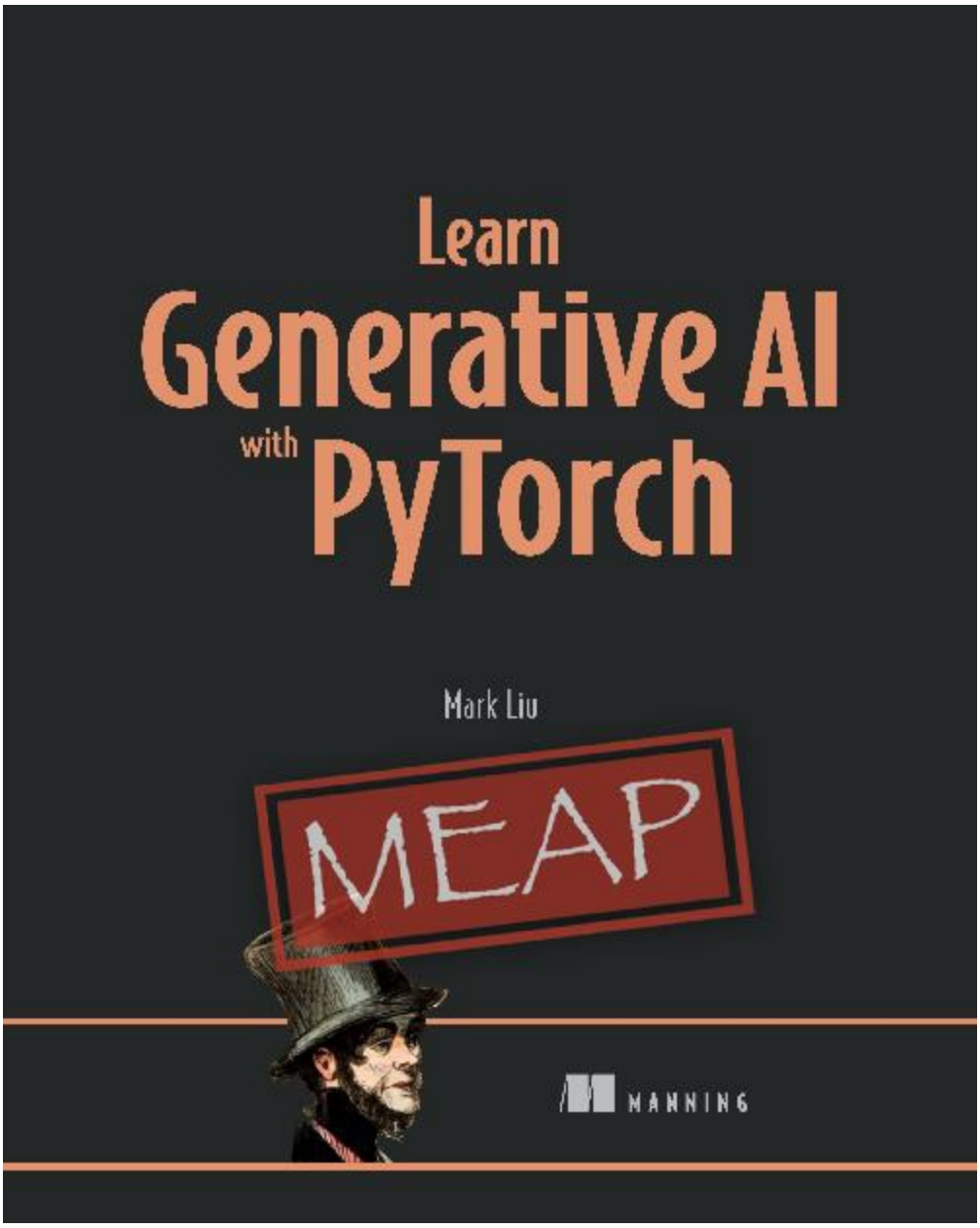
Learn Generative AI with PyTorch

Mark Liu

MEAP

MANNING


Learn
Generative AI
with PyTorch
Mark Liu
MEAP
MANNING

Learn Generative AI with PyTorch

1. welcome
2. 1_What_Is_Generative_AI_and_Why_PyTorch?
3. 2_Deep_Learning_with_PyTorch
4. 3_Generative_Adversarial_Networks_(GANs):_Shape_and_Number_Ge
5. 4_Image_Generation_with_GANs
6. 5_Selecting_Characteristics_in_Generated_Images
7. 6_CycleGAN:_Converting_Blond_Hair_to_Black_Hair
8. 7_Image_Generation_with_Variational_Autoencoders_(VAEs)
9. 8_Text_Generation_with_Recurrent_Neural_Networks_(RNNs)
10. 9_A_Line-by-Line_Implementation_of_Attention_and_Transformer
11. Appendix._Installing_Python,_Jupyter_Notebook,_and_PyTorch

welcome

Welcome to the MEAP for *Learn Generative AI with PyTorch*. This book is designed for those who have a good grasp of Python and a basic understanding of machine learning, particularly neural networks. It aims to guide you through the creation of generative models from the ground up.

My fascination with generative AI began a few years back, when I first saw models converting horse images into zebra images and Transformers producing lifelike text. This book is born out of my journey in building and understanding these models from scratch. It's the book I wish I had during my experiments with various generative models. It begins with simple models, helping readers build foundational deep learning skills before advancing to more complex challenges. I chose PyTorch for its dynamic computational graph and clear syntax after experimenting with TensorFlow.

All generative models in this book are deep neural networks. The book starts with a comprehensive deep learning project in PyTorch, ideal for those new to the field. Each chapter is carefully structured to build upon the previous one, especially beneficial for readers new to deep learning in PyTorch. You'll start by creating basic content like shapes, numbers, and images using Generative Adversarial Networks (GANs) with straightforward architectures. As you progress, the complexity increases, culminating in building advanced models like Transformers.

This book's purpose is to empower you to build and understand generative models thoroughly. Following Richard Feynman's philosophy - "What I cannot create, I do not understand" - it encourages hands-on learning. You're invited to delve into the Python code and Jupyter notebooks available on the book's GitHub repository at https://github.com/markhliu/DGAI and engage with exercises in each chapter.

Your questions, comments, and suggestions are invaluable for refining this book. Please share your thoughts in the livebook discussion forum, contributing to the creation of an outstanding resource in generative AI.

—Mark Liu

In this book

welcome 1 What Is Generative AI and Why PyTorch? 2 Deep Learning with PyTorch 3 Generative Adversarial Networks (GANs): Shape and Number Generation 4 Image Generation with GANs 5 Selecting Characteristics in Generated Images 6 CycleGAN: Converting Blond Hair to Black Hair 7 Image Generation with Variational Autoencoders (VAEs) 8 Text Generation with Recurrent Neural Networks (RNNs) 9 A Line-by-Line Implementation of Attention and Transformer
Appendix. Installing Python, Jupyter Notebook, and PyTorch

1 What Is Generative AI and Why PyTorch?

This chapter covers

- What is generative AI and how is it different from its non-generative counterparts
- Why is PyTorch suitable for deep learning and generative AI
- The idea behind Generative Adversarial Networks (GANs)
- The advantages of the attention mechanism and Transformers
- The benefits of building generative AI models from scratch

Generative AI has significantly impacted the global landscape, capturing widespread attention and becoming a focal point since the advent of ChatGPT in November 2022. This technological advancement has revolutionized numerous aspects of everyday life, marking a new era in technology and inspiring a host of startups to explore the extensive possibilities offered by generative AI.

Consider the advancements made by Midjourney, a pioneering company, which now creates high-resolution, realistic images from brief text inputs. Similarly, Freshworks, a leading software company, has accelerated application development dramatically, reducing the time required from an average of ten weeks to mere days, a feat achieved through the capabilities of ChatGPT.[1] Notably, elements of this very introduction have been enhanced by AI, demonstrating its capability to refine content to be more engaging.[2]

The repercussions of this technological advancement extend far beyond these examples. Industries are experiencing significant disruption due to the advanced capabilities of generative AI. This technology now produces essays comparable to those written by humans, composes music reminiscent of classical compositions, and rapidly generates complex legal documents, tasks that typically require considerable human effort and time. Following the release of ChatGPT, Cheggmate, an educational platform, witnessed a

significant decrease in its stock value.[3] Furthermore, the Writers Guild of America, during a recent strike, reached a consensus to put guardrails around AI's encroachment on scriptwriting and editing.[4]

This raises several questions: What is generative AI, and how does it differ from other AI technologies? Why is it causing such widespread disruption across various sectors? What is the underlying mechanism of generative AI, and why is it important to understand?

This book offers an in-depth exploration of generative AI, a groundbreaking technology reshaping numerous industries through its efficient and rapid content creation capabilities. Specifically, you'll learn to use state-of-the-art generative models to create various forms of content: shapes, numbers, patterns, images, text, and audio. Further, instead of treating these models as black boxes, you'll learn to create them from scratch, so you have a deep understanding of the inner workings of generative AI. In the words of physicist Richard Feynman, "What I cannot create, I do not understand."

All these models are based on deep neural networks. Python and PyTorch have been selected for building these models, with Python preferred for its user-friendly syntax, cross-platform compatibility, and robust community support. PyTorch is chosen over other frameworks like TensorFlow for its dynamic computational graph, enhancing adaptability to various model architectures and facilitating debugging. Python is widely regarded as the primary tool for machine learning, and PyTorch has become increasingly popular in the field of AI. Therefore, using Python and PyTorch allows you to follow the new developments in generative AI. Because PyTorch allows for graphics processing unit (GPU) training acceleration, you'll train these models in a matter of minutes or hours and witness generative AI in action!

1.1 Introducing Generative AI and PyTorch

This section explains what generative AI is and how it's different from its non-generative counterparts: discriminative models. Generative AI is a category of technologies with the remarkable capacity to produce diverse forms of new content, including textual narratives, graphical designs, video sequences, source code, and intricate patterns. Generative AI crafts entirely

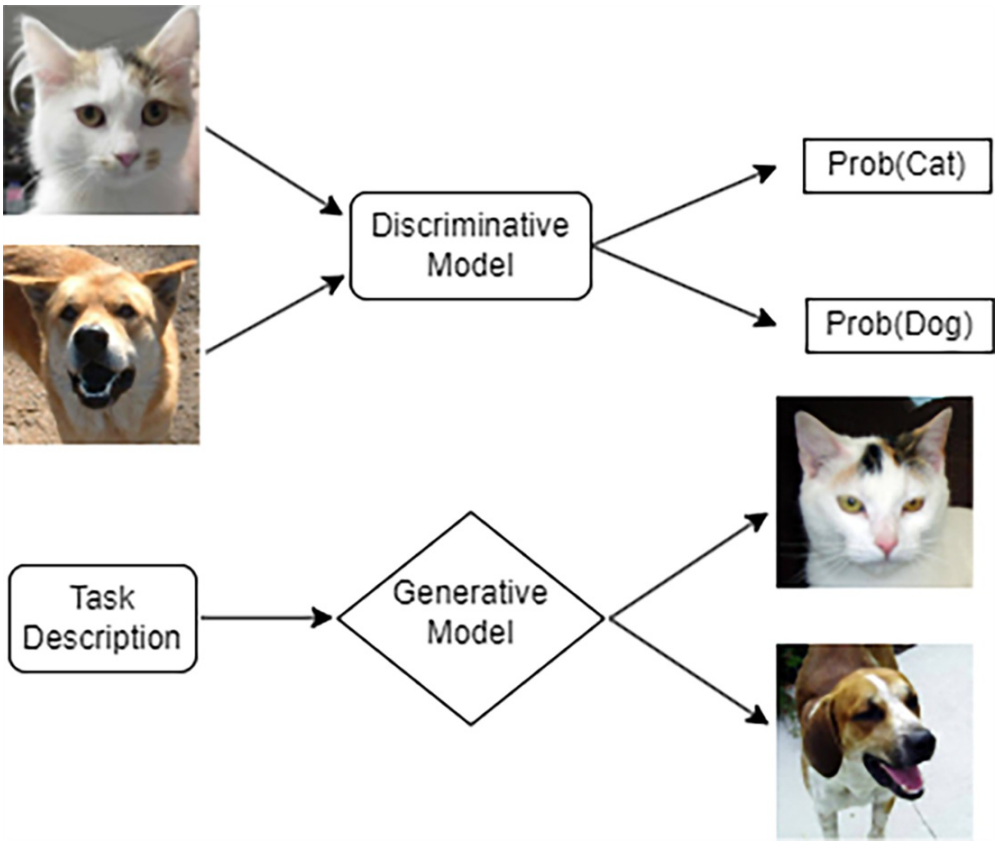
new worlds of novel and innovative content, with ChatGPT being a notable example in generating text. In contrast, discriminative modeling predominantly concerns itself with the task of recognizing and categorizing pre-existing content.

This section also explains why we choose Python and PyTorch for deep learning and generative modeling in this book.

1.1.1 What Is Generative AI?

Discriminative models and generative models are two fundamental approaches in AI to understanding and processing data. Each has its own characteristics and applications. Discriminative models specialize in discerning disparities among distinct data instances and learning the boundary between classes. Figure 1.1 illustrates the difference between these two modeling methods. For instance, when confronted with an array of images featuring dogs and cats, a discriminative model adeptly determines whether each image portrays a dog or a cat by capturing a few key features that distinguish one from the other (e.g., cats have small noses and pointy ears). As the top half of figure 1.1 shows, a discriminative model takes data as inputs and produces probabilities of different labels, which we denote by Prob(dog) and Prob(cat). We can then label the inputs based on the highest predicted probabilities.

Figure 1.1 A comparison of generative models versus discriminative models. A discriminative model (top half of the figure) takes data as inputs and produces probabilities of different labels, which we denote by Prob(dog) and Prob(cat). In contrast, a generative model (bottom half) acquires an in-depth understanding of the defining characteristics of these images to synthesize new images representing dogs and cats.



In stark contrast, generative models exhibit a unique ability to generate novel instances of data. In the context of our dog and cat example, a generative model acquires an in-depth understanding of the defining characteristics of these images to synthesize new images representing dogs and cats. As the bottom half of figure 1.1 shows, a generative model takes task descriptions as inputs and produces brand-new images of dogs and cats.

From a statistical perspective, when presented with data examples with features X which describe the input and various corresponding labels Y, discriminative models undertake the responsibility of predicting conditional probabilities, specifically the probability prob(Y|X). Conversely, generative models embark on a distinct journey, seeking to learn the joint probability distribution of the input features X and the target variable Y, denoted as prob(X,Y). Armed with this knowledge, they sample from the distribution to conjure fresh instances of X.

Within the realm of generative models, a multitude of specialized models exist, each tailored to generate content of specific types. In this book, our

focus primarily centers on two prominent technologies: Generative Adversarial Networks (GANs) and Transformers. The word "adversarial" in GANs refers to the fact that the two neural networks compete against each other in a zero-sum game framework: the generative network tries to create data instances indistinguishable from real samples, while the discriminative network tries to identify the generated samples from real ones. The competition between the two networks leads to the improvement of both, eventually enabling the generator to create highly realistic data. Transformers are deep neural networks that can efficiently solve sequence-to-sequence prediction tasks, and we'll explain them in more detail later in this chapter.

GANs, celebrated for their ease of implementation and versatility, empower individuals with even rudimentary knowledge of deep learning to construct their generative models from the ground up. These versatile models can give rise to a plethora of creations, from geometric shapes and intricate patterns, as exemplified in Chapter 3 of this book, to high-quality color images like human faces, which you'll learn to generate in Chapter 4. Furthermore, GANs exhibit the ability to transform image content, seamlessly morphing a human face image with blond hair into one with black hair, a phenomenon discussed in Chapter 6. Notably, they extend their creative prowess to the realm of music generation, producing realistic-sounding musical compositions.

In contrast, the art of text generation poses formidable challenges, chiefly due to the sequential nature of textual information, where the order and arrangement of individual characters and words hold significant meaning. To confront this complexity, we turn to Transformers, deep neural networks designed to proficiently address sequence-to-sequence prediction tasks. Unlike their predecessors, such as recurrent neural networks (RNNs) or convolutional neural networks (CNNs), Transformers excel in capturing intricate, long-range dependencies inherent in both input and output sequences. Notably, their capacity for parallel training (a distributed training method in which a model is trained on multiple devices simultaneously) has substantially reduced training times, rendering the processing of extensive datasets feasible.

The revolutionary architecture of Transformers underpins the emergence of

large language models (LLMs; deep neural networks with a massive number of parameters and trained on large datasets), including ChatGPT, BERT, DALL-E, and T5. This transformative architecture serves as the bedrock of the recent surge in AI advancement, ushered in by the introduction of ChatGPT and other pre-trained transformer (GPT) models.

In the subsequent sections, we delve into the comprehensive inner workings of these two pioneering technologies: their underlying mechanisms and the myriad possibilities they unlock.

1.1.2 The Python Programming Language

I assume you have a working knowledge of Python. To follow the content in the book, you need to know the Python basics such as functions, classes, lists, dictionaries, and so on. If not, there are plenty of free resources online to get you started. Follow the instructions in the Appendix to install Python. After that, create a virtual environment for this book and install Jupyter Notebook as the computing environment for projects in this book.

Python has established itself as the leading programming language globally since the latter part of 2018, as documented by The Economist.[5] Python is not only free for everyone to use but also allows other users to create and tweak libraries. Python has a massive community-driven ecosystem, so you can easily find resources and assistance from fellow Python enthusiasts. Plus, Python programmers love to share their code, so instead of reinventing the wheel, you can import pre-made modules and share your own with the Python community.

No matter if you're on Windows, Mac, or Linux, Python's got you covered. It's a cross-platform language, although the process of installing software and libraries might vary a bit depending on your operating system. But don't worry; we'll show you how to do it in the appendix of this book. Once everything's set up, Python code behaves the same across different systems.

Python is an expressive language that's suitable for general application development. Its syntax is easy to grasp, making it straightforward for AI enthusiasts to understand and work with. Python has a supportive AI

community, with plenty of online groups and forums where programmers share their challenges and help each other out. If you run into any issues with the Python libraries mentioned in this book, you can search these forums or visit sites like Stack Overflow (https://stackoverflow.com/questions/tagged/python) for answers. And if all else fails, don't hesitate to reach out to me for assistance.

Lastly, Python offers a treasure trove of libraries that make creating generative models easy (relative to other languages such as C++ or R). In this journey, we'll exclusively use PyTorch as your AI framework, and I'll explain why we pick it over competitors like TensorFlow shortly.

1.1.3 Using PyTorch as Your AI Framework

Now that we have settled on using Python as the programming language for this book, we'll choose a suitable AI framework for generative modeling. The two most popular AI frameworks in Python are PyTorch and TensorFlow. In this book, we'll choose PyTorch over TensorFlow and I strongly encourage you to do the same, for the following reasons.

PyTorch is an open-source machine learning (ML) library developed by Meta's AI Research lab. It's based on the Python programming language and the Torch library, an ML library used for creating deep neural networks in Lua, an alternative programming language to Python. It was created with the goal of providing researchers and developers with a flexible and intuitive platform for building and training deep learning models.

A computational graph is a fundamental concept in deep learning that plays a crucial role in the efficient computation of complex mathematical operations, especially those involving multi-dimensional arrays or tensors. A computational graph is a directed graph where the nodes represent mathematical operations, and the edges represent tensors (data) that flow between these operations. One of the key uses of computational graphs is in the implementation of backpropagation and gradient descent algorithms. The graph structure allows for the efficient calculation of gradients required to update the model parameters during training. Unlike static graphs in frameworks like TensorFlow, PyTorch creates and modifies the graph on the

fly, which is called dynamic computational graph. This makes it more adaptable to varying model architectures and simplifies debugging. Further, just like TensorFlow, PyTroch provides accelerated computation through GPU training, which can reduce training time by 80-90% compared to central processing unit (CPU) training.

PyTorch's design aligns well with the Python programming language. Its syntax is concise and easy to understand, making it accessible to both newcomers and experienced developers. Researchers and developers alike appreciate PyTorch for its flexibility. It empowers them to experiment with novel ideas quickly, thanks to its dynamic graph and simple interface. This flexibility is crucial in the rapidly evolving fields of generative AI. PyTorch also has a rapidly growing community that actively contributes to its development. This results in an extensive ecosystem of libraries, tools, and resources, making it easier for developers to find solutions to their AI challenges.

PyTorch excels in transfer learning, a technique where pre-trained models designed for a general task are fine-tuned for specific tasks. Researchers and practitioners can easily leverage pre-trained models, saving time and computational resources. This feature is especially important in the age of pre-trained LLMs, which allows us to adopt LLMs for downstream tasks such as classification, text summarization, and text generation.

PyTorch is compatible with other Python libraries, such as NumPy and SciPy. This interoperability allows data scientists and engineers to seamlessly integrate PyTorch into their existing workflows, enhancing productivity. PyTorch is also known for its commitment to community-driven development. It evolves rapidly, with regular updates and enhancements based on real-world usage and user feedback, ensuring that it remains at the cutting edge of AI research and development.

The appendix of this book provides detailed instructions on how to install PyTorch on your computer. Follow the instructions to install PyTorch in the virtual environment for this book. In case you don't have a Compute Unified Device Architecture (CUDA)-enabled GPU installed on your computer, all programs in this book are compatible with CPU training as well. Better yet, I'll provide the trained models on the book's GitHub repository

https://github.com/markhliu/DGAI so you can see the trained models in action. In Chapter 2 of the book, you'll dive deep into PyTorch. You'll first learn the data type in PyTroch, Tensor, which holds numbers and matrices and provides functions to conduct operations. You'll then learn to perform an end-to-end deep learning project by using PyTorch. Specifically, you'll create a neural network in PyTorch and use clothing image item images and corresponding labels to train the network. Once done, you use the trained model to classify clothing items into ten different label types. The project will get you ready to use PyTorch to build and train various generative models in later chapters.

1.2 Generative Adversarial Networks (GANs)

This section first provides a high-level overview of how Generative Adversarial Networks (GANs) work. We then use the generation of Anime faces as an example to show you the inner workings of GANs. Finally, we'll discuss the practical uses of GANs.

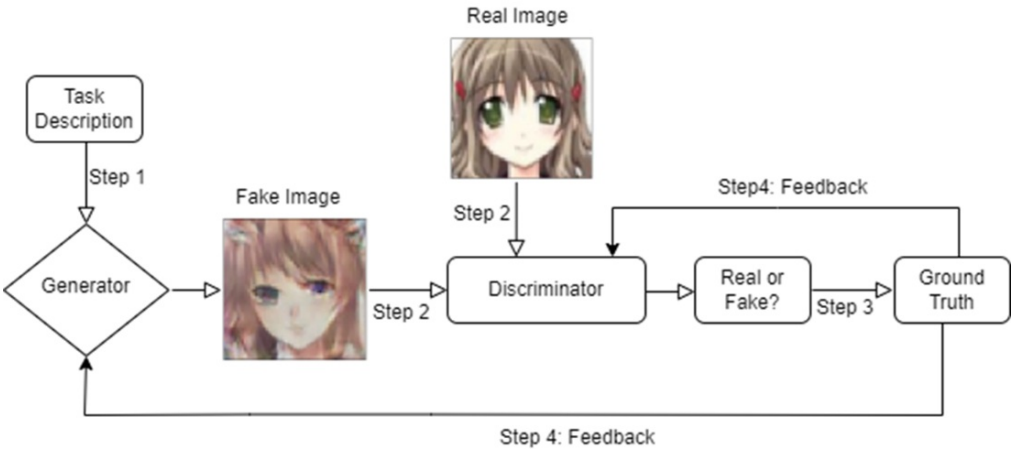
1.2.1 A High-Level Overview of GANs

Generative Adversarial Networks (GANs) represent a category of generative models initially proposed by Ian Goodfellow and his collaborators in 2014. [6] GANs have become extremely popular in recent years because they are easy to build and train and they can generate a wide variety of content. As you'll see from the illustrating example in the next subsection, GANs employ a dual-network architecture comprising a generative model tasked with capturing the underlying data distribution and a discriminative model that serves to estimate the likelihood that a given sample originates from the authentic training dataset (considered as "real") rather than being a product of the generative model (considered as "fake"). The primary objective of the model is to produce new data instances that closely resemble those in the training dataset. The nature of the data generated by GANs is contingent upon the composition of the training dataset. For example, if the training data consists of grayscale images of clothing items, the synthesized images will closely resemble such clothing items. Conversely, if the training dataset comprises color images of human faces, the generated images will also

resemble human faces.

Now, take a look at figure 1.2 – the architecture of our GAN and its components. To train the model, both real samples from the training dataset (as shown at the top of figure 1.2) and fake samples created by the generator (left) are presented to the discriminator (middle). The principal aim of the generator is to create data instances that are virtually indistinguishable from the examples found within the training dataset. Conversely, the discriminator strives to distinguish fake samples generated by the generator from real samples. These two networks engage in a continual competitive process similar to a cat-and-mouse game, trying to outperform each other iteratively.

Figure 1.2 Generative Adversarial Networks (GANs) architecture and its components. GANs employ a dual-network architecture comprising a generative model (left) tasked with capturing the underlying data distribution and a discriminative model (center) that serves to estimate the likelihood that a given sample originates from the authentic training dataset (considered as "real") rather than being a product of the generative model (considered as "fake").



The training process of the GAN model involves multiple iterations. In each iteration, the generator takes some form of task description and uses it to create fake images (step 1). The fake images, along with real images from the training set, are presented to the discriminator (step 2). The discriminator tries to classify each sample as either real or fake. It then compares the classification with the actual labels, the ground truth (step 3). Both the discriminator and the generator receive feedback (step 4) from the classification and improve their capabilities: while the discriminator adapts its ability to identify fake samples, the generator learns to enhance its capacity to generate convincing samples. As training advances, an

equilibrium is reached when neither network can further improve. At this point, the generator becomes capable of producing data instances that are practically indistinguishable from real samples.

To understand exactly how GANs work, let’s look at an illustrating example.

1.2.2 An Illustrating Example: Generating Anime Faces

Picture this: You're a passionate Anime enthusiast, and you're on a thrilling quest to create your very own Anime faces using a powerful tool known as a deep convolutional GAN (or DCGAN for short; don't worry, we'll dive deeper into this in Chapter 4).

If you look at the top middle of figure 1.2, you'll spot a picture that reads "Real Image." We’ll use 63,632 colorful images of Anime faces as our training dataset. And if you flip to figure 1.3, you'll see 32 examples from our training set. These special images play a crucial role as they form half of the inputs to our discriminator network.

Figure 1.3 Examples from the Anime faces training dataset.

The left of figure 1.2 is the generator network. To generate different images every time, the generator takes as input a vector Z from the latent space. We could think of this vector as a “task description.” During training, we use random values to fill in the Z vector, so the network generates different

images every time.[7] These fake images are the other half of the inputs to the discriminator network.

But here's the twist: before we teach our two networks the art of creation and detection, the images produced by the generator are, well, gibberish! They look nothing like the realistic Anime faces you have seen in figure 1.3. In fact, they resemble nothing more than static on a TV screen (you'll witness this firsthand in Chapter 4).

We train the model for multiple iterations. Each time, we randomly pluck 128 genuine Anime face images from our training stockpile. Then, we obtain 128 fake images from the generator. We present all 256 images to the discriminator, who must decide the likelihood of each image being genuine.

You may wonder: how do the discriminator and the generator learn during each iteration of training? Once the predictions are made, the discriminator doesn't just sit back; it learns from its prediction blunders for each image. With this newfound knowledge, it fine-tunes its parameters, shaping itself to make better predictions in the next round. The generator isn't idle either. It takes notes from the image generation process and the prediction outcomes. With that knowledge in hand, it adjusts its own network parameters, striving to create increasingly lifelike images in the next iteration. The goal? To reduce the odds of the discriminator sniffing out its fakes.

As we journey through these iterations, a remarkable transformation takes place. The generator network evolves, producing Anime faces that grow more and more realistic, akin to those in our training collection. Meanwhile, the discriminator network hones its skills, becoming a seasoned detective when it comes to spotting fakes. It's a captivating dance between creation and detection.

Gradually, a magical moment arrives. An equilibrium, a perfect balance, is achieved. The images created by the generator become so astonishingly real that they are indistinguishable from the genuine Anime faces in our training archives. At this point, the discriminator is so confused that it assigns a 50% chance of authenticity to every image, whether it’s from our training set or was crafted by the generator.

Finally, behold some examples of the artwork of the generator, as shown in figure 1.4: they do look indistinguishable from those in our training set.

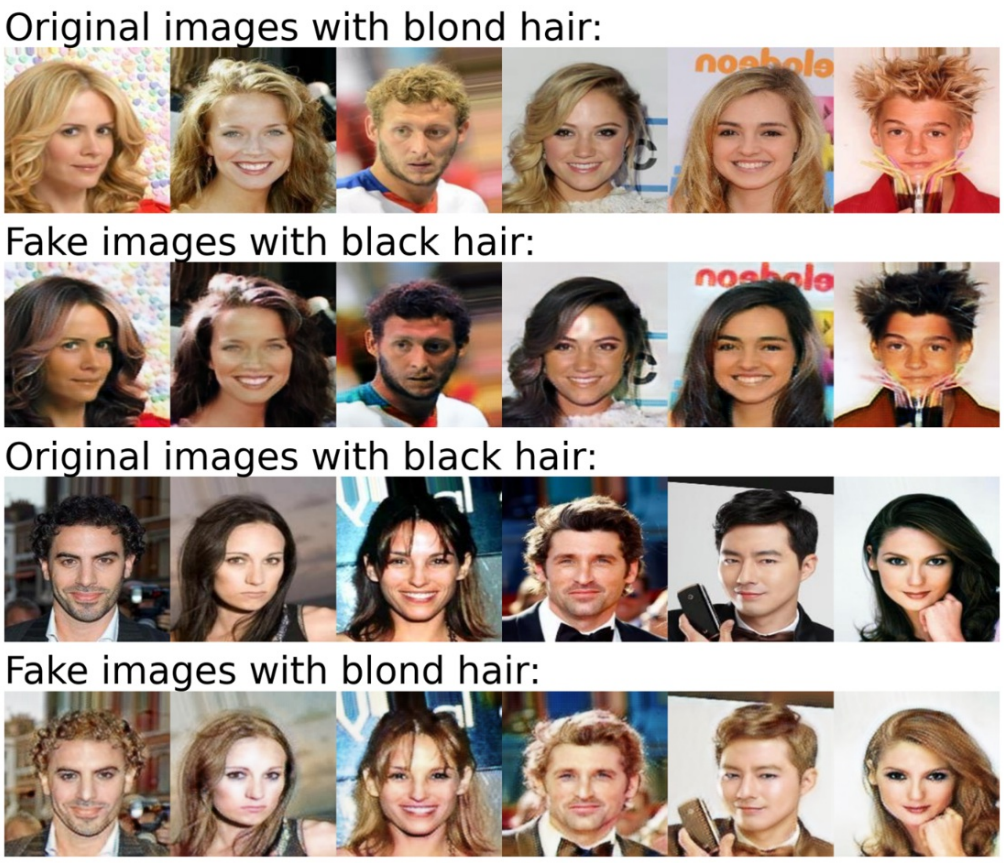
Figure 1.4 Generated Anime face images by the trained generator in DCGAN.

1.2.3 Why Should You Care About GANs?

GANs are easy to implement and versatile: you'll learn to generate geometric shapes, intricate patterns, high-resolution images, and realistic-sounding music in this book alone.

The practical use of GANs doesn't stop at generating realistic data. GANs can also translate attributes in one image domain to another. As you'll see in Chapter 6, you can train a CycleGAN (a type of generative model in the GAN family) to convert blond hair to black hair in human face images. The same trained model can also convert black hair to blond hair. Figure 1.5 shows four rows of images. The first row is the original images with blond hair. The trained CycleGAN converts them to images with black hair (second row). The last two rows are the original images with black hair and the converted image with blond hair, respectively.

Figure 1.5 Changing hair color with CycleGAN. If we feed images with blond hair (first row) to a trained CycleGAN model, the model converts blond hair to black hair in these images (second row). The same trained model can also convert black hair (third row) to blond hair (bottom row).



Think about all the amazing skills you'll pick up from training GANs—they're not just cool, they're super practical too! Let's say you run an online clothing store with a 'Make to Order' strategy (which allows users to customize their purchases before manufacturing). Your website showcases tons of unique designs for customers to pick from, but here's the catch: you only make the clothes once someone places an order. Creating high-quality images of these clothes can be quite expensive since you have to produce the items and then photograph them.

Now, GANs to the rescue! You don't need a massive collection of manufactured clothing items and their images; instead, you can use something like CycleGAN to transform features from one set of images into another, creating a whole new array of styles. This is just one nifty way to use GANs. The possibilities are endless because these models are super versatile and can handle all sorts of data—making them a game-changer for practical applications.

1.3 Transformers

Transformers are deep neural networks that excel at sequence-to-sequence prediction problems such as taking an input sentence and predicting the most likely next words. This section introduces you to the key innovation in Transformers: the self-attention mechanism. We'll then discuss the Transformer architecture and different types of Transformers. Finally, we'll discuss some recent developments in Transformers such as multimodal models (Transformers whose inputs include not only text but also other data types such as audio and images) and pre-trained LLMs (models trained on large textual data that can perform various downstream tasks).

Before the Transformer architecture was invented in 2017 by a group of Google researchers,[8] natural language processing (NLP) and other sequence-to-sequence prediction tasks were handled mainly by recurrent neural networks (RNNs) such as long short-term memory (LSTM) algorithms. However, RNNs process inputs sequentially, which means the model processes one input at a time, in sequence, instead of looking at the entire sequence simultaneously. The fact that RNNs conduct computation along the symbol positions of the input and output sequences prevents parallel training, which makes training slow. This, in turn, makes it impossible to train the models on huge datasets. Further, when recurrent models process a sequence, they gradually lose information on earlier elements in the sequence, which makes these models unsuccessful in capturing long-term dependencies in sequences.

The key innovation of Transformers is the self-attention mechanism, which excels at capturing long-term dependencies in a sequence. Further, since the inputs are not handled sequentially in the model, Transformers can be trained in parallel, and this greatly reduces the training time. More importantly, parallel training makes it possible to train Transformers on large amounts of data and this makes LLMs extremely intelligent and knowledgeable. This leads to the rise of LLMs such as ChatGPT and the current AI boom.

1.3.1 The Self-Attention Mechanism

The self-attention mechanism assigns weights on how an element is related to

other elements in a sequence. The higher the weight, the more closely the two elements are related. These weights are learned from large sets of training data in the training process. Therefore, a trained LLM such as ChatGPT is able to figure out the relationship between any two words in a sentence, hence making sense of the human language.

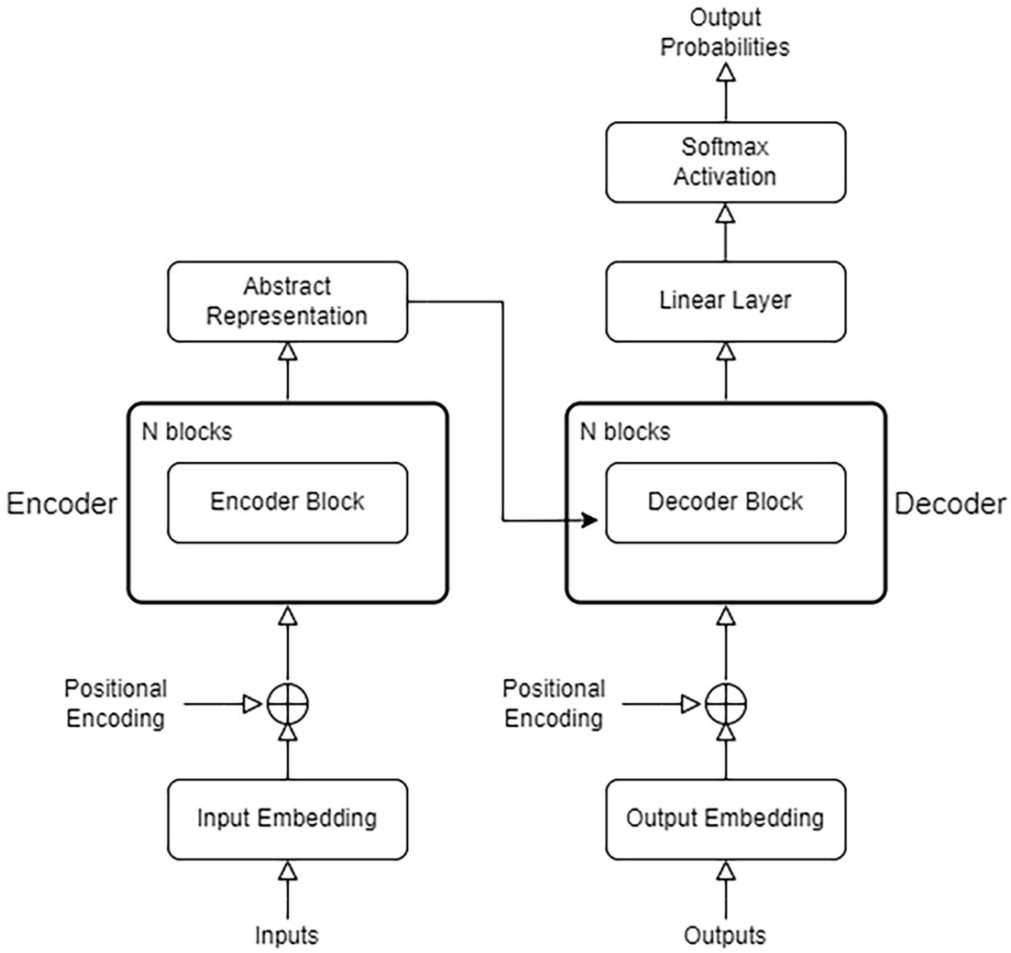
You may wonder: How does the attention mechanism assign scores to elements in a sequence to capture the long-term dependencies? Well, the attention weights are calculated by first passing the inputs, through three neural network layers, query Q, key K, and value V (which we'll explain in detail in Chapter 9). The method of using query, key, and value to calculate attention comes from retrieval systems. For example, you may go to a public library to search for a book. You can type in, say, "machine learning in finance," in the library's search engine. In this case, the query Q is "machine learning in finance." The keys K are the book titles, book descriptions, and so on. The library's retrieval system will recommend a list of books (values V) for you based on the similarities between the query and the keys. Naturally, books with the phrases "machine learning" or "finance" or both in titles or descriptions come up on top while books with neither phrase in the title or description will show up at the bottom of the list because these books will be assigned a low matching score.

In Chapter 9, you'll learn the details of the self-attention mechanism. Better yet, you'll implement the self-attention mechanism from scratch to build a Transformer to successfully translate English to French.

1.3.2 The Transformer Architecture

Transformers were first proposed when designing models for machine language translation (e.g., English to German or English to French). Figure 1.6 is a diagram of the Transformer architecture. The left side is the encoder and the right side is the decoder.

Figure 1.6 The Transformer architecture. The encoder in the Transformer (left side of the diagram) learns the meaning of the input sequence and converts it into a vector that represents this meaning before passing it to the decoder (right side of the diagram). The decoder constructs the output (e.g., the French translation of an English phrase) by predicting one word at a time, based on previous words in the sequence and the vector representation from the encoder.



The encoder in the Transformer “learns” the meaning of the input sequence (e.g., an English sentence) and converts it into a vector that represents this meaning before passing it to the decoder. The decoder constructs the output (e.g., the French translation of an English phrase) by predicting one word at a time, based on previous words in the sequence and the continuous vector representation from the encoder. In Chapter 9, you’ll learn to create such a Transformer from scratch and train it to translate English to French. The trained model will provide exactly the same translation (Je ne parle pas français) to two slightly different English sentences: “I don’t speak French” and “I do not speak French.” This shows that the encoder has captured the meaning of the two sentences and converted them to identical vector representations before passing them to the decoder.

There are three types of Transformers: encoder-only Transformers, decoder-only Transformers, and encoder-decoder Transformers. An encoder-only

Transformer has no decoder and is capable of converting a sequence into an abstract continuous vector representation for various downstream tasks such as sentiment analysis, named entity recognition, and text generation. In Chapter 11, you'll build and train an encoder-only Transformer from scratch and use it to generate text in a certain style. A decoder-only Transformer has only a decoder but no encoder and it's well suited for text generation, language modeling, and creative writing. GPT-2 (the predecessor of ChatGPT) and ChatGPT are both decoder-only Transformers. In Chapter 10, you'll learn to create GPT-2 from scratch and then extract the trained model weights from Hugging Face (an AI community that hosts and collaborates on ML models, datasets, and applications). You'll load the weights to your GPT-2 model and start generating text.

Encoder-decoder Transformers are needed for handling complicated tasks such as multi-modal models that can handle text-to-image tasks or speech recognition tasks. Encoder-decoder Transformers combine the strengths of both encoders and decoders. Encoders are efficient in processing and understanding input data, while decoders excel in generating output. This combination allows the model to effectively understand complex inputs (like text or speech) and generate intricate outputs (like images or transcribed text).

1.3.3 Multimodal Transformers and Pre-Trained LLMs

Recent developments in generative AI give rise to various multimodal models: Transformers that can use not only text but also other data types such as audio and images as inputs. Text-to-image Transformers are one such example. DALL-E, ImageGen, and Stable Diffusion are all text-to-image models and they have garnered much media attention due to their ability to generate high-resolution images from textual prompts. Text-to-image Transformers incorporate the principles of diffusion models, which involve a series of transformations to gradually increase the complexity of data. Therefore, we first need to understand diffusion models before we discuss text-to-image Transformers.

Imagine you want to generate high-resolution flower images by using a diffusion-based model. You'll first obtain a training set of high-quality flower images. You then ask the model to gradually add noise to the flower images

(the so-called diffusion process) until they become completely random noisy images. You then train the model to progressively remove noise to generate new data samples. The diffusion process is illustrated in figure 1.7. The left column contains four original flower images. As we move to the right, some noise is added to the images in each step, until at the right column, the four images are completely noisy images.

Figure 1.7 The diffusion model adds more and more noise to the images and learns to reconstruct them. The left column contains four original flower images. As we move to the right, some noise is added to the images in each step, until at the right column, the four images are completely noisy images. We then use these images to train a diffusion-based model to progressively remove noise to generate new data samples.

You may be wondering: how are text-to-image Transformers related to diffusion models? Well, text-to-image Transformers take a text prompt as input and generate images that correspond to that textual description. The text prompt serves as a form of conditioning, and the model uses a series of neural network layers to transform that textual description into an image. Like diffusion models, text-to-image Transformers leverage a hierarchical architecture with multiple layers, each progressively adding more detail to the generated image. The core concept of iteratively refining the output is similar in both diffusion models and Text-to-image Transformers.

Diffusion models have now become more and more popular due to their

ability to provide more stable training and generate high-quality images, and they have outperformed other generative models such as GANs and Variational Autoencoders (VAEs). In Chapter 14, you'll first learn to train a simple diffusion model using the Oxford Flower data set. You'll also learn the basic idea behind multimodal Transformers and write a Python program to ask OpenAI's DALL-E 2 to generate images through a text prompt. For example, when I entered "an astronaut in a space suit riding a unicorn" as the prompt, DALL-E 2 generated the image as shown in figure 1.8.

Figure 1.8 Image generated by DALL-E 2 with text prompt "an astronaut in a space suit riding a unicorn".

In Chapter 14, you'll learn how to access pre-trained LLMs such as ChatGPT, GPT4, and DALL-E. These models are trained on large textual data and have learned general knowledge from the data. Hence, they can

perform various downstream tasks such as text generation, sentiment analysis, question answering, and named entity recognition. Since pre-trained LLMs are trained on information a few months ago, they cannot provide information on events and developments in the last one or two months, let alone real-time information such as weather conditions, flight status, or stock prices. We'll use the LangChain library to chain together LLMs with a real-time internet search API (serpaip) to create a know-it-all agent.

1.4 Why Build Generative Models from Scratch?

The goal of this book is to show you how to build and train all generative models from scratch. This way, you'll have a thorough understanding of the inner workings of these models and can make better use of them. Creating something from scratch is the best way to understand something. You'll accomplish this goal for GANs: all models, including DCGAN and CycleGAN, are built from the ground up and trained using well-curated data in the public domain.

For Transformers, you'll build and train all models from scratch except for LLMs. This exception is due to the vast amount of data and the supercomputing facilities needed to train certain LLMs. However, you'll make serious progress in this direction. Specifically, you'll implement in Chapter 9 the original groundbreaking 2017 paper "Attention is all you need" line by line with English to French translation as an example (the same Transformer can be trained on other datasets such as Chinese to English or English to German translations). You'll also build a small-size encoder-only Transformer and train it using Ernest Hemingway's work *the old man and the sea*. The trained model can generate text in Hemingway style. ChatGPT and GPT-4 are too large to build and train from scratch for our purposes, but you'll peek into their predecessor, GPT-2, and learn to build it from scratch. That's not all, you'll also extract the trained weights from Hugging Face and load them up to the GPT-2 model you built and start to generate realistic text that can pass as human-written.

In this sense, the book is taking a more fundamental approach than most books. Instead of treating generative AI models as a black box, readers like you have a chance to look under the hood and examine in detail the inner

workings of these models. The goal is for you to have a deeper understanding of generative models as a result. This, in turn, can potentially help you build better and more responsible generative AI for the following reasons.

First, having a deep understanding of the architecture of generative models helps readers like you make better practical uses of these models. For example, in Chapter 5, you'll learn how to select characteristics in generated images such as male or female features and with or without eyeglasses. By building a conditional GAN from the ground up, you understand that certain features of the generated images are determined by the random noise vector, Z, in the latent space. Therefore, you can choose different values of Z as inputs to the trained model to generate the desired characteristics (such as male or female features). This type of attribute selection is hard to do without understanding the design of the model.

For Transformers, knowing the architecture (and what encoders and decoders do) gives you the ability to create and train Transformers to generate the types of content you are interested in (Jane Austin style novels or Mozart style music, for example). This understanding also helps you with pre-trained LLMs. For example, while it is hard to train GPT-2 from scratch with its 1.5 billion parameters, you can add an additional layer to the model and fine-tune it for other downstream tasks such as text classification, sentiment analysis, and question-answering.

Second, a deep understanding of generative AI helps readers like you have an unbiased assessment of the dangers of AI. While the extraordinary powers of generative AI have benefited us in our daily lives and work, it also has the potential to create great harm. Elon Musk went so far as saying that "there's some chance that it goes wrong and destroys humanity."[9] More and more people in academics and in the tech industry are worried about the dangers posed by AI in general and by generative AI in particular. Generative AI, especially LLMs, can lead to unintended consequences, as many pioneers in the tech profession have warned.[10] It's not a coincidence that merely five months after the release of ChatGPT, many tech industry experts and entrepreneurs, including Steve Wozniak, Tristan Harris, Yoshua Bengio, and Sam Altman, signed an open letter calling for a pause in training any AI system that's more powerful than GPT-4 for at least six months.[11] A

thorough understanding of the architecture of generative models helps us provide a deep and unbiased evaluation of the benefits and potential dangers of AI.

1.5 Summary

- Generative AI is a type of technology with the capacity to produce diverse forms of new content, including texts, images, code, music, and patterns.
- Discriminative models specialize in assigning labels while generative models generate new instances of data.
- PyTorch, with its dynamic computational graphs and the ability for GPU training, is well suited for deep learning and generative modeling.
- GANs are a type of generative modeling method consisting of two neural networks: a generator and a discriminator. The goal of the generator is to create realistic data samples to maximize the chance that the discriminator thinks they are real. The goal of the discriminator is to correctly identify fake samples from real ones.
- Transformers are deep neural networks that use the attention mechanism to identify long-term dependencies among elements in a sequence. The original Transformer has an encoder and a decoder. When it's used for English-to-French translation, for example, the encoder converts the English sentence into a vector representation before passing it to the decoder. The decoder generates the French translation one word at a time, based on the vector representation from the encoder and the previously generated words.

[1] See the *Forbes* article "10 Amazing Real-World Examples of How Companies Are Using ChatGPT in 2023," by Bernard Barr, 2023, https://www.forbes.com/sites/bernardmarr/2023/05/30/10-amazing-real-world-examples-of-how-companies-are-using-chatgpt-in-2023/?sh=e3b169614418

[2] What better way to explain generative AI than letting generative AI do itself? I asked ChatGPT to rewrite an early draft of this introduction in a "more engaging manner" before finalizing it.

[3] Cheggmate charges college students to have their questions answered by human specialists. Many of these jobs can now be done by ChatGPT or similar tools at a fraction of the costs.

[4] See the *WIRED* article "Hollywood Writers Reached an AI Deal That Will Rewrite History," by Will Bedingfield, 2023, https://www.wired.com/story/us-writers-strike-ai-provisions-precedents/

[5] See the article "Python is becoming the world's most popular coding language" by the Data Team at Economist in 2018, https://www.economist.com/graphic-detail/2018/07/26/python-is-becoming-the-worlds-most-popular-coding-language

[6] Goodfellow et al, 2014, Generative Adversarial Nets, https://arxiv.org/abs/1406.2661

[7] By altering the random noise vector, different outputs can be generated. In Chapter 5, you'll learn how to select random noise vectors to generate images with certain characteristics (e.g., male or female features).

[8] Vaswani et al, "Attention is all you need," 2017, https://arxiv.org/abs/1706.03762

[9] See the article by Julia Mueller in The Hill, 2023, "Musk: There's a chance AI 'goes wrong and destroys humanity'", https://thehill.com/policy/technology/4008144-musk-theres-a-chance-ai-goes-wrong-and-destroys-humanity/

[10] See, e.g., Stuart Russell, 2023, "How to stop runaway AI", https://www.freethink.com/robots-ai/how-to-stop-runaway-ai

[11] See the article by Connie Loizos in TechCrunch, "1,100+ notable signatories just signed an open letter asking 'all AI labs to immediately pause for at least 6 months'", https://techcrunch.com/2023/03/28/1100-notable-signatories-just-signed-an-open-letter-asking-all-ai-labs-to-immediately-pause-for-at-least-6-months/

2 Deep Learning with PyTorch

This chapter covers

- What are PyTorch tensors and how to conduct operations on them
- Preparing data in PyTorch for deep learning
- Building and training deep neural networks with PyTorch
- Binary and multi-category classifications with deep learning
- Creating a validation set to determine when to stop training

In this book, we'll use deep neural networks to generate a wide range of content, including text, images, shapes, music, and more. I assume you possess a prior familiarity with the inner workings of machine learning (ML), and specifically, a foundational understanding of artificial neural networks. Throughout this chapter, I'll refresh your memory on a few key concepts such as loss functions, activation functions, optimizers, and the learning rate, which constitute indispensable elements in the development and training of deep neural networks. Should there be any gaps in your knowledge pertaining to these concepts, it is strongly encouraged that you rectify them before progressing further with projects in this book.[1]

Generative AI models are frequently confronted with the task of either binary or multi-category classification. For instance, in Generative Adversarial Networks (GANs), the discriminator undertakes the essential role of a binary classifier, its purpose being to distinguish between the fake samples created by the generator from real samples from the training set. Similarly, in the context of text generation models, whether in recurrent neural networks or Transformers, the overarching objective is to predict the subsequent character or word from an extensive array of possibilities (essentially a multi-category classification task).

In this chapter, you'll learn how to use PyTorch to create deep neural networks to perform binary and multi-category classifications so that you become well-versed in deep learning and classification tasks.

Specifically, you'll engage in an end-to-end deep learning project in PyTorch, on a quest to classify grayscale images of clothing items into different categories such as coats, bags, sneakers, shirts, and so on. The intention is to prepare you for the creation of deep neural networks, capable of performing both binary and multi-category classification tasks in PyTorch. This, in turn, will get you ready for the upcoming chapters, where you use deep neural networks in PyTorch to create various generative models.

To train generative AI models, we harness a diverse range of data formats such as raw text, audio files, image pixels, and arrays of numbers. Deep neural networks created in PyTorch cannot take these forms of data directly as inputs. Instead, we must first convert them into a format that the neural networks understand and accept. Specifically, you'll convert various forms of raw data into PyTorch tensors (fundamental data structures used to represent and manipulate data) before feeding them to generative AI models. Therefore, in this chapter, you'll also learn the basics of data types, how to create various forms of PyTorch tensors, and how to use them in deep learning.

Knowing how to perform classification tasks has many practical applications in our society. Classifications are widely used in healthcare for diagnostic purposes, such as identifying whether a patient has a particular disease (e.g., positive or negative for a specific cancer based on medical imaging or test results). They play a vital role in many business tasks (stock recommendations, credit card fraud detection, and so on). Classification tasks are also integral to many systems and services that we use daily such as spam detection and facial recognition.

2.1 Data Types in PyTorch

We'll use datasets from a wide range of sources and formats in this book, and the first step in deep learning is to transform the inputs into arrays of numbers.

In this section, you'll learn how PyTorch converts different formats of data into algebraic structures known as *tensors*. Tensors can be represented as multidimensional arrays of numbers, similar to NumPy arrays but with

several key differences, chief among them the ability of GPU accelerated training. There are different types of tensors depending on their end-use and you'll learn how to create different types of tensors and when to use each type. We'll discuss the data structure in PyTorch in this section by using the heights of the 46 U.S. presidents as our running example.

2.1.1 Create PyTorch Tensors

When training deep neural networks, we feed the models with arrays of numbers as inputs. Depending on what a generative model is trying to create, these numbers have different types. For example, when generating images, the inputs are raw pixels in the form of integers between 0 and 255 but we'll convert them to floating-point numbers between -1 and 1; when generating text, there is a "vocabulary" akin to a dictionary, and the input is a sequence of integers telling you which entry in the dictionary the word corresponds to.

NOTE

The code for this chapter, as well as other chapters in this book, are available at the book's GitHub repository https://github.com/markhliu/DGAI.

Imagine you want to use PyTorch to calculate the average height of the 46 U.S. presidents. We can first collect the heights of the 46 U.S. presidents in centimeters and store them in a Python list, as follows:

```
heights = [189, 170, 189, 163, 183, 171, 185,
           168, 173, 183, 173, 173, 175, 178,
           183, 193, 178, 173, 174, 183, 183,
           180, 168, 180, 170, 178, 182, 180,
           183, 178, 182, 188, 175, 179, 183,
           193, 182, 183, 177, 185, 188, 188,
           182, 185, 191, 183]
```

The numbers are in chronological order: the first value in the list, 189, indicates that the first U.S. president, George Washington, was 189 centimeters tall. The last value shows that Joe Biden's height is 183 centimeters. We can convert a Python list into a PyTorch tensor by using the `tensor()` method in PyTorch, like so:

```
import torch

heights_tensor = torch.tensor(heights,    #A
           dtype=torch.float64)    #B
```

We specify the data type by using the `dtype` argument in the `tensor()` method. The default data type in PyTorch tensors is `float32`, a 32-bit floating-point number. In the above code cell, we converted the data type to `float64`, double-precision floating-point numbers. `float64` provides more precise results than `float32`, but it takes longer to compute. There is a trade-off between precision and computational costs. Which data type to use depends on the task at hand.

Table 2.1 lists different data types and the corresponding PyTorch tensor types. These include integers and floating-point numbers with different precisions. Integers can also be either signed or unsigned.

Table 2.1 Data and tensor types in PyTorch

PyTorch tensor type	dtype argument in tensor()	Data type
FloatTensor	torch.float32 or torch.float	32-bit floating point
HalfTensor	torch.float16 or torch.half	16-bit floating point
DoubleTensor	torch.float64 or torch.double	64-bit floating point
CharTensor	torch.int8	8-bit integer (signed)
ByteTensor	torch.uint8	8-bit integer (unsigned)
ShortTensor	torch.int16 or torch.short	16-bit integer (signed)
IntTensor	torch.int32 or torch.int	32-bit integer (signed)
LongTensor	torch.int64 or torch.long	64-bit integer (signed)

To create a tensor with a certain data type, you can do it in one of the two ways. The first way is to use the PyTorch class as specified in the first column of Table 2.1. The second way is to use the `torch.tensor()` method

and specify the data type using the `dtype` argument (the value of the argument is listed in the second column of Table 2.1). For example, to convert the Python list `[1, 2, 3]` into a PyTorch tensor with 32-bit integers in it, you can use the following two methods:

Listing 2.1 Two ways of specifying tensor types

```
t1=torch.IntTensor([1, 2, 3])    #A
t2=torch.tensor([1, 2, 3],
             dtype=torch.int)    #B
print(t1)
print(t2)
```

This leads to the following output:

```
tensor([1, 2, 3], dtype=torch.int32)
tensor([1, 2, 3], dtype=torch.int32)
```

Exercise 2.1

Use two different methods to convert the Python list `[5, 8, 10]` into a PyTorch tensor with 64-bit floating-point numbers in it. Consult the second row in table 2.1 for this question.

Many times, you need to create a PyTorch tensor with values 0 everywhere. For example, in GANs, we create a tensor of zeros as the labels for fake samples, as you'll see in Chapter 3. The `zeros()` method in PyTorch generates a tensor of zeros with a certain shape. In PyTorch, a tensor is an n-dimensional array, and its shape is a tuple representing the size along each of its dimensions. The following lines of code generate a tensor of zeros with two rows and three columns:

```
tensor1 = torch.zeros(2, 3)
print(tensor1)
```

The output is as follows:

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

The tensor has a shape of (2, 3), which means the tensor is a two-dimensional

array; there are two elements in the first dimension and three elements in the second dimension. Here, we didn't specify the data type and the output has the default data type of *float32*.

From time to time, you need to create a PyTorch tensor with values 1 everywhere. For example, in GANs, we create a tensor of ones as the labels for real samples. Below, we use the `ones()` method to create a three-dimensional tensor with values 1 everywhere:

```
tensor2 = torch.ones(1,4,5)
print(tensor2)
```

The output is as follows:

```
tensor([[[1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1.]]])
```

We have generated a three-dimensional PyTorch tensor. The shape of the tensor is (1, 4, 5).

Exercise 2.2

Create a three-dimensional PyTorch tensor with values 0 in it. Make the shape of the tensor (2, 3, 4).

You can also use a NumPy array instead of a Python list in the tensor constructor, like so:

```
import numpy as np

nparr=np.array(range(10))
pt_tensor=torch.tensor(nparr, dtype=torch.int)
print(pt_tensor)
```

The output is:

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=torch.int32)
```

2.1.2 Index and Slice PyTorch Tensors

We use square brackets ([ ]) to index and slice PyTorch tensors, as we do with Python lists. Indexing and slicing allow us to operate on one or more elements in a tensor, instead of on all elements. To continue our example of the heights of the 46 U.S. presidents, if we want to access the height of the third president, Thomas Jefferson, we can do the following:

```
height = heights_tensor[2]
print(height)
```

This leads to an output of:

```
tensor(189., dtype=torch.float64)
```

The output shows that the height of Thomas Jefferson was 189 centimeters.

We can use negative indexing to count from the back of the tensor. For example, to find out the height of Donald Trump, who is the second to last president in the list, we use index -2, as follows:

```
height = heights_tensor[-2]
print(height)
```

The output is:

```
tensor(191., dtype=torch.float64)
```

The output shows that Trump's height is 191 centimeters.

What if we want to know the heights of the last five presidents in the tensor `heights_tensor`? We can obtain a slice of the tensor like this:

```
five_heights = heights_tensor[-5:]
print(five_heights)
```

The colon (:) is used to separate the starting and end index. If no starting index is provided, the default is 0; if no end index is provided, you include the very last element in the tensor (as we did in the code cell above). Negative indexing means you count from the back. The output is:

```
tensor([188., 182., 185., 191., 183.], dtype=torch.float64)
```

The results show that the last five presidents in the tensor (Bush, Clinton, Obama, Trump, and Biden) are 188, 182, 185, 191, and 183 centimeters tall, respectively.

Exercise 2.3

Use slicing to obtain the heights of the first five U.S. presidents in the tensor `heights_tensor`.

2.1.3 PyTorch Tensor Shapes

PyTorch tensors have an attribute *shape*, which tells us the dimensions of a tensor. It's important to know the shapes of PyTorch tensors because mismatched shapes will lead to errors when we operate on them. For example, if we want to find out the shape of the tensor `heights_tensor`, we can do this:

```
print(heights_tensor.shape)
```

The output is

```
torch.Size([46])
```

This tells us that `heights_tensor` is a one-dimensional tensor with 46 values in it.

You can also change the shape of a PyTorch tensor. To learn how, let's first convert the heights from centimeters to feet. Since a foot is about 30.48 centimeters, we can accomplish this by dividing the tensor by 30.48:

```
heights_in_feet = heights_tensor / 30.48
print(heights_in_feet)
```

This leads to the following output (I omitted some values to save space; the complete output is in the book's GitHub repository):

```
tensor([6.2008, 5.5774, 6.2008, 5.3478, 6.0039, 5.6102, 6.0696, …
        6.0039], dtype=torch.float64)
```

The new tensor, `heights_in_feet`, stores the heights in feet. For example, the last value in the tensor shows that Joe Biden is 6.0039 feet tall.

We can use the `cat()` method in PyTorch to concatenate the two tensors:

```
heights_2_measures = torch.cat(
    [heights_tensor,heights_in_feet], dim=0)
print(heights_2_measures.shape)
```

The `dim` argument is used in various tensor operations to specify the dimension along which the operation is to be performed. In the above code cell, `dim=0` means we concatenate the two tensors along the first dimension. This leads to the following output:

```
torch.Size([92])
```

The resulting tensor is one-dimensional with 92 values, with some values in centimeters and others in feet. Therefore, we need to reshape it into two rows and 46 columns so that the first row represents heights in centimeters and the second in feet:

```
heights_reshaped = heights_2_measures.reshape(2, 46)
```

The new tensor, `heights_reshaped`, is two-dimensional with a shape of (2, 46). We can index and slice multi-dimensional tensors using square brackets as well. For example, to print out the height of Trump in feet, we can do this:

```
print(heights_reshaped[1,-2])
```

This leads to a result of:

```
tensor(6.2664, dtype=torch.float64)
```

The command `heights_reshaped[1,-2]` tells Python to look for the value in the second row and the second to last column, which returns Trump's height in feet, 6.2664.

tip

The number of indexes needed to refer to scalar values within the tensor is

the same as the dimensionality of the tensor. That's why we used only one index to locate values in the one-dimensional tensor `heights_tensor` but we used two indexes to locate values in the two-dimensional tensor `heights_reshaped`.

Exercise 2.4

Use indexing to obtain the height of Joe Biden in the tensor `heights_reshaped` in centimeters.

2.1.4 Mathematical Operations on PyTorch Tensors

We can conduct mathematical operations on PyTorch tensors by using different methods such as `mean()`, `median()`, `sum()`, `max()`, and so on. For example, to find the median height of the 46 presidents in centimeters, we can do this:

```
print(torch.median(heights_reshaped[0,:]))
```

This leads to an output of:

```
tensor(182., dtype=torch.float64)
```

To find out the average height in both rows, we can use the `dim=1` argument in the `mean()` method:

```
print(torch.mean(heights_reshaped,dim=1))
```

The `dim=1` argument indicates that the averages are calculated along the dimension indexed 1 (columns), effectively collapsing rows (the dimension indexed 0). The output is:

```
tensor([180.0652,   5.9077], dtype=torch.float64)
```

The results show that the average values in the two rows are 180.0652 centimeters and 5.9077 feet.

To find out the tallest president, we can do this:

```
values, indices = torch.max(heights_reshaped, dim=1)
print(values)
print(indices)
```

The output is as follows:

```
tensor([193.0000,   6.3320], dtype=torch.float64)
tensor([15, 15])
```

The `torch.max()` method returns two tensors: a tensor `values` with the tallest president's height (in centimeters and in feet), and a tensor `indices` with the indexes of the president with the maximum height. The results show that the 16th president (Lincoln) is the tallest, at 193 centimeters, or 6.332 feet.

Exercise 2.5

Use the `torch.min()` method to find out the index and height of the shortest U.S. president.

2.2 An End-to-End Deep Learning Project with PyTorch

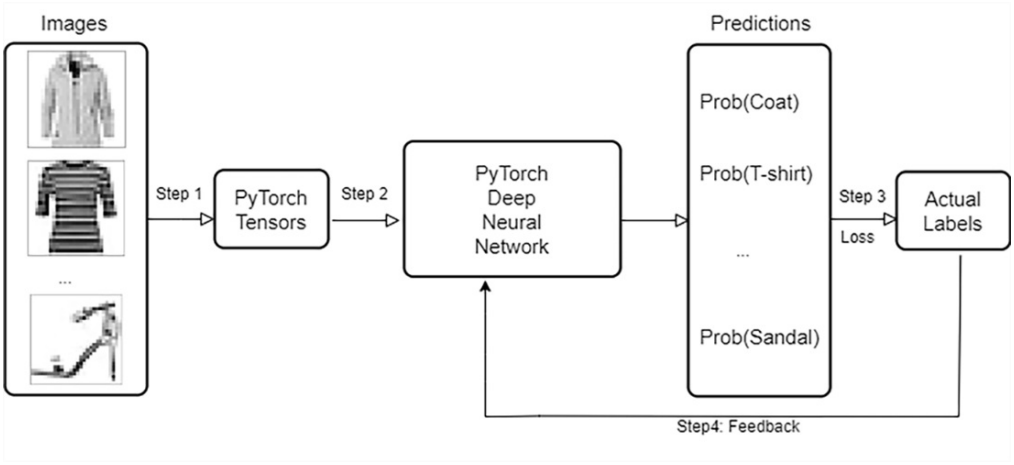
In the next few sections, you'll work through an example deep learning project with PyTorch, learning to classify grayscale images of clothing items into one of the ten types.

In this section, we'll first provide a high-level overview of the steps involved. We then discuss how to obtain training data for this project and how to preprocess the data.

2.2.1 Deep learning in PyTorch: A high-level overview.

Our job in this project is to create and train a deep neural network in PyTorch to classify grayscale images of clothing items. Figure 2.1 provides a diagram of the steps involved.

Figure 2.1 A diagram of steps involved in deep learning in PyTorch.



First, we'll obtain a dataset of grayscale clothing images, as shown on the left of figure 2.1. The Images are in raw pixels and we'll convert them to PyTorch tensors in the form of float numbers (step 1). Each image comes with a label.

We'll then create a deep neural network in PyTorch, as shown in the center of figure 2.1. Some neural networks in this book involve convolutional neural networks (CNNs). For this simple classification problem, we'll use dense layers only for the moment.

We'll select a loss function for multi-class classification, and cross-entropy loss is commonly used for this task. We'll use the Adam optimizer (a variant of the gradient descent algorithm) to update the network's weights during training. We set the learning rate to 0.001.

We'll divide the training data into a train set and a validation set. In ML, we usually use the validation set to provide an unbiased evaluation of the model and to select the best hyperparameters such as the learning rate, number of epochs of training, and so on. The validation set can also be used to avoid overfitting the model in which the model works well in the training set but poorly on unseen data. An epoch is when all the training data is used to train the model once and only once.

During training, you'll iterate through the training data. During forward passes, you feed images through the network to obtain predictions (step 2), and compute the loss by comparing the predicted labels with the actual labels (step 3; see the right side of figure 2.1). You'll then backpropagate the

gradient through the network to update the weights. This is where the learning happens (step 4), as shown at the bottom of figure 2.1.

You'll use the validation set to determine when we should stop training. We calculate the loss in the validation set. If the model stops improving after a fixed number of epochs, we consider the model trained. We then evaluate the trained model on the test set to assess its performance in classifying images into different labels.

Now that you have a high-level overview of how deep learning in PyTorch works, let's dive into the end-to-end project!

2.2.2 Preprocessing Data

We'll be using the Fashion Modified National Institute of Standards and Technology (MNIST) dataset in this project. Along the way, you'll learn how to use the *datasets* and *transforms* packages in the Torchvision library, as well as the *Dataloader* packages in PyTorch that will help you for the rest of the book. You'll use these tools to preprocess data throughout the book.

We first import needed libraries and instantiate a `Compose()` class in the *transforms* packages to transform raw images to PyTorch tensors:

Listing 2.2 Transforming raw image data to PyTorch tensors

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as T

torch.manual_seed(42)
transform=T.Compose([    #A
    T.ToTensor(),    #B
    T.Normalize([0.5],[0.5])])    #C
```

We use the `manual_seed()` method in PyTorch to fix the random state so that results are reproducible. The *transforms* package in Torchvision can help create a series of transformations to preprocess images. The `ToTensor()` class converts image data (in either PIL image formats or NumPy arrays) into

PyTorch tensors. In particular, the image data are integers ranging from 0 to 255 and the `ToTensor()` class converts them to float tensors with values in the range of 0.0 and 1.0.

The `Normalize()` class normalizes tensor images with mean and standard deviation for *n* channels. The Fashion MNIST data are grayscale images of clothing items so there is only one color channel. Later in this book, we'll deal with images of three different color channels (red, green, and blue; RGB). In the code cell above, `Normalize([0.5],[0.5])` means that we subtract 0.5 from the data and divide the difference by 0.5. The resulting image data range from -1 to 1. Normalizing the input data to the range [-1, 1] helps in faster convergence during training and you'll do this often in this book.

NOTE

The code in listing 2.2 only defines the data transformation process. It doesn't perform the actual transformation, which happens in the next code cell.

Next, we use the *datasets* package in Torchvision to download the dataset to a folder on your computer and perform the transformation, like so:

```
train_set=torchvision.datasets.FashionMNIST(    #A
    root=".",    #B
    train=True,    #C
    download=True,    #D
    transform=transform)    #E
test_set=torchvision.datasets.FashionMNIST(root=".",
    train=False,download=True,transform=transform)
```

You can print out the first sample in the training set as follows:

```
print(train_set[0])
```

The first sample consists of a tensor with 784 values and a label 9. The 784 numbers are the values in the 28 by 28 grayscale image (28*28=784), and the label 9 means it's an ankle boot. You maybe wondering: how do you know the label 9 indicates an ankle boot. There are ten different types of clothing items. The labels in the dataset are numbered from 0 to 9. You can search

online and find out the text labels for the ten categories.[2] The list `text_labels` below contains the ten text labels corresponding to the numerical labels 0 to 9. For example, if an item has a numerical label of 0 in the dataset, the corresponding text label is "t-shirt". The list `text_labels` is defined as follows:

```
text_labels=['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
             'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
```

We can plot the data to visualize the clothes items in the dataset:

Listing 2.3 Visualizing the clothing items

```
import matplotlib.pyplot as plt

plt.figure(dpi=300,figsize=(8,4))
for i in range(24):
    ax=plt.subplot(3, 8, i + 1)    #A
    img=train_set[i][0]    #B
    img=img/2+0.5    #C
    img=img.reshape(28, 28)    #D
    plt.imshow(img,
               cmap="binary")
    plt.axis('off')
    plt.title(text_labels[train_set[i][1]],    #E
        fontsize=8)
plt.show()
```

The plot in Figure 2.2 shows 32 clothes items such as coats, pullovers, sandals, and so on.

Fig 2.2 Grayscale images of clothes items in the Fashion MNIST dataset.

You'll learn how to create deep neural networks with PyTorch to perform binary and multi-category classification problems in the next two sections.

2.3 Binary Classification

In this section, we'll first create batches of data for training. We then build a deep neural network in PyTorch for this purpose and train the model using the data. Finally, we'll use the trained model to make predictions and test how accurate the predictions are. The steps involved with binary and multi-category classifications are similar, with a few notable exceptions that I'll highlight later.

2.3.1 Creating Batches

We'll create a training set and a test set that contain only two types of clothing items: t-shirts and ankle boots.[3] The code cell below accomplishes that goal:

```
binary_train_set=[x for x in train_set if x[1] in [0,9]]
binary_test_set=[x for x in test_set if x[1] in [0,9]]
```

We only keep samples with numerical labels 0 and 9 so as to create a binary classification problem with a balanced training set. Next, we create batches for training the deep neural network:

Listing 2.4 Creating batches for training and testing

```
batch_size=64
binary_train_loader=torch.utils.data.DataLoader(
    binary_train_set,    #A
    batch_size=batch_size,    #B
    shuffle=True)    #C
binary_test_loader=torch.utils.data.DataLoader(
    binary_test_set,    #D
    batch_size=batch_size,shuffle=True)
```

The `DataLoader` class in the PyTorch *utils* package helps create data iterators in batches. We set the batch size to 64. We created two data loaders in listing 2.3: a training set and a test set for binary classification. We shuffle the observations when creating batches to avoid correlations among the original dataset: the training is more stable if different labels are evenly distributed in the data loader.

2.3.2 Building and Training A Binary Classification Model

We'll first create a binary classification model. We then train the model by using the images of t-shirts and ankle boots. Once it's trained, we'll see if the model can tell t-shirts from ankle boots.

We first use PyTorch to create the following neural network by using the Pytorch `nn.Sequential` class (in later chapters, you'll also learn to use the `nn.Module` class to create PyTorch neural networks):

Listing 2.5 Creating a binary classification model

```
import torch.nn as nn

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

binary_model=nn.Sequential(    #B
    nn.Linear(28*28,256),    #C
    nn.ReLU(),    #D
    nn.Linear(256,128),
    nn.ReLU(),
    nn.Linear(128,32),
    nn.ReLU(),
    nn.Linear(32,1),
    nn.Dropout(p=0.25),
```

```
    nn.Sigmoid()).to(device)    #E
```

The `Linear()` class in PyTorch creates a linear transformation of the incoming data. This effectively creates a dense layer in the neural network. The input shape is 784 because we'll later flatten the two-dimensional image to a one-dimensional vector with 28×28=784 values in it. We flatten the 2D image into a 1D tensor because dense layers only takes 1D inputs. In later chapters, you'll see that you don't need to flatten images when you use convolutional layers. There are three hidden layers in the network, with 256, 128, and 32 neurons in them, respectively. The numbers 256, 128, and 32 are chosen somewhat arbitrarily: changing them to, say, 300, 200, and 50 won't affect the training process. We apply the ReLU activation function on the three hidden layers. Activation functions decide whether a neuron should be turned on or not based on the weighted sum. They introduce nonlinearity to the output of a neuron so that the network can learn nonlinear relations between inputs and outputs. ReLU is your go-to activation function with very few exceptions and you'll encounter a few other activation functions in later chapters.

The output of the last layer of the model contains a single value, and we use the sigmoid activation function to squeeze the number to the range [0, 1] so that it can be interpreted as the probability that the object is an ankle boot. With the complementary probability, the object is a t-shirt.

Below, we set the learning rate and define the optimizer and the loss function:

```
lr=0.001
optimizer=torch.optim.Adam(binary_model.parameters(),lr=lr)
loss_fn=nn.BCELoss()
```

We set the learning rate to 0.001. What learning rate to set is an empirical question and it comes with experience. It can also be determined by using hyperparameter tuning using a validation set. Most optimizers in PyTorch use a default learning rate of 0.001. The Adam optimizer is a variant of the gradient descent algorithm, which is used to determine how much to adjust the model parameters in each training step. The Adam optimizer was first introduced in 2014 by Diederik Kingma and Jimmy Ba.[4] In the traditional gradient descent algorithm, only gradients in the current iteration are considered. The Adam optimizer, in contrast, takes into consideration

gradients in previous iterations as well.

We use `nn.BCELoss()`, which is the binary cross-entropy loss function. Loss functions measure how well a machine learning model performs. The training of a model involves adjusting parameters to minimize the loss function. The binary cross-entropy loss function is widely used in machine learning, particularly in binary classification problems. It measures the performance of a classification model whose output is a probability value between 0 and 1. The cross-entropy loss increases as the predicted probability diverges from the actual label.

We train the neural network we just created as follows:

Listing 2.6 Traning a binary classification model

```
for i in range(50):    #A
    tloss=0
    for imgs,labels in binary_train_loader:    #B
        imgs=imgs.reshape(-1,28*28)    #C
        imgs=imgs.to(device)
        labels=torch.FloatTensor(\
          [x if x==0 else 1 for x in labels])    #D
        labels=labels.reshape(-1,1).to(device)
        preds=binary_model(imgs)
        loss=loss_fn(preds,labels)    #E
        optimizer.zero_grad()
        loss.backward()    #F
        optimizer.step()
        tloss+=loss
    print(f"at epoch {i}, loss is {tloss}")
```

We train the model for 50 epochs for simplicity (an epoch is when the training data is used to train the model once). In the next section, you'll use a validation set and an early stopping class to determine how many epochs to train. In binary classifications, we label the targets as 0s and 1s. Since we have kept only t-shirts and ankle boots with labels 0 and 9, respectively, we convert them to 0 and 1 in listing 2.6. As a result, the labels for the two categories of clothing items are 0 and 1, respectively.

The above training takes a few minutes if you use GPU training. It takes

longer if you use CPU training, but the training time should be less than an hour.

2.3.3 Testing the Binary Classification Model

The prediction from the trained binary classification model is a number between 0 and 1. We'll use the `torch.where()` method to convert the predictions into 0s and 1s and compare them with the actual labels. If the predicted probability is less than 0.5, we label the prediction as 0. Otherwise, we label the prediction as 1.

In listing 2.7, we use the trained model to make predictions on the test dataset as follows:

Listing 2.7 Calculating the accuracy of the predictions

```
results=[]
for imgs,labels in binary_test_loader:    #A
    imgs=imgs.reshape(-1,28*28).to(device)
    labels=(labels/9).reshape(-1,1).to(device)
    preds=binary_model(imgs)
    pred10=torch.where(preds>0.5,1,0)    #B
    correct=(pred10==labels)    #C
    results.append(correct.detach().cpu()\
      .numpy().mean())    #D
accuracy=np.array(results).mean()    #E
print(f"the accuracy of the predictions is {accuracy}")
```

We iterate through all batches of data in the test set. The trained model produces a probability that the image is an ankle boot. We then convert the probability into 0 or 1 based on the cutoff value of 0.5, by using the `torch.where()` method. The predictions are either 0 (i.e., a t-shirt) or 1 (an ankle boot) after the conversion. We compare the predictions with the actual labels and see how many times the model gets it right. Results show that the accuracy of the predictions is 87.84% in the test set.

2.4 Multi-Category Classification

In this section, we'll build a deep neural network in PyTorch to classify the

clothing items into one of the ten categories. We'll then train the model with the Fashion MNIST dataset. Finally, we'll use the trained model to make predictions and see how accurate they are.

We first create a validation set and define an early stopping class so that we can determine when to stop training.

2.4.1 Validation Set and Early Stopping

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

To give you an example, we'll create a validation set in the multi-category classification to determine the optimal number of epochs to train. The reason we do this in the validation set instead of the training set is to avoid overfitting, when a model performs well in the training set but poorly in out-of-the-sample tests (i.e., on unseen data).

Below, we divide 60000 observations of the training dataset into a train set and a validation set:

```
train_set,val_set=torch.utils.data.random_split(\
    train_set,[50000,10000])
```

The original train set now becomes two sets: the new train set with 50000 observations and a validation set with the remaining 10000 observations.

We use the `DataLoader` class in the PyTorch *utils* package to convert the train, validation, and test sets into three data iterators in batches, as follows:

```
train_loader=torch.utils.data.DataLoader(
    train_set,
    batch_size=batch_size,
    shuffle=True)
val_loader=torch.utils.data.DataLoader(
    val_set,
    batch_size=batch_size,
```

```
    shuffle=True)
test_loader=torch.utils.data.DataLoader(
    test_set,
    batch_size=batch_size,
    shuffle=True)
```

Next, we define an `EarlyStop()` class and create an instance of the class:

Listing 2.8 The EarlyStop() class to determine when to stop training

```
class EarlyStop:
    def __init__(self, patience=10):    #A
        self.patience = patience
        self.steps = 0
        self.min_loss = float('inf')
    def stop(self, val_loss):    #B
        if val_loss < self.min_loss:    #C
            self.min_loss = val_loss
            self.steps = 0
        elif val_loss >= self.min_loss:    #D
            self.steps += 1
        if self.steps >= self.patience:
            return True
        else:
            return False
stopper=EarlyStop()
```

The `EarlyStop()` class determines if the loss in the validation set has stopped improving in the last `patience=10` epochs. We set the default value of `patience` argument to 10, but you can choose a different value when you instantiate the class. The value of `patience` measures how many epochs you want to train since the last time the model reached the minimum loss. The `stop()` method keeps a record of the minimum loss and the number of epochs since the minimum loss, and compares the number to the value of `patience`. The method returns a value of `True` if the number of epochs since the minimum loss is greater than the value of `patience`.

2.4.2 Build and Train a Multi-Category Classification Model

The Fashion MNIST dataset contains ten different categories of clothing items. Therefore, we create a multi-category classification model to classify them. Next, you'll learn how to create such a model and train it. You'll also

learn how to make predictions using the trained model and assess the accuracy of the predictions.

We use PyTorch to create the following neural network for multi-category classification:

Listing 2.9 Creating a multi-category classification model

```
model=nn.Sequential(
    nn.Linear(28*28,256),
    nn.ReLU(),
    nn.Linear(256,128),
    nn.ReLU(),
    nn.Linear(128,64),
    nn.ReLU(),
    nn.Linear(64,10),    #A
    nn.Dropout(p=0.20),
nn.Softmax(dim=1)).to(device)    #B
```

Compared to the binary classification model we created in the last section, we have made a few changes here. First, the output now has ten values in it, representing the ten different types of clothing items in the dataset. Second, we have changed the number of neurons in the last hidden layer from 32 to 64. A rule of thumb in creating deep neural networks is to gradually increase or decrease the number of neurons from one layer to the next. Since the number of output neurons has increased from 1 (in binary classification) to 10 (in multi-category classification), we change the number of neurons from 32 to 64 in the second to last layer to match the increase.

Finally, we apply the softmax activation function instead of the sigmoid activation function on the output to squeeze the ten numbers into the range [0, 1]. The activation function on the output is always sigmoid in binary classifications and softmax in multi-category classifications. Further, the ten numbers add up to 1. The ten outputs can be interpreted as the probabilities corresponding to the ten types of clothing items.

We'll use the same learning rate and optimizer as those in the binary classification in the last section. However, we now use the cross-entropy loss function to conduct our multi-category classification. The preferred loss

function is binary cross-entropy loss in binary classifications and categorical cross-entropy loss in multi-category classifications.

```
lr=0.001
optimizer=torch.optim.Adam(model.parameters(),lr=lr)
loss_fn=nn.CrossEntropyLoss()
```

We'll define the train_epoch() below:

```
def train_epoch():
    tloss=0
    for n,(imgs,labels) in enumerate(train_loader):
        imgs=imgs.reshape(-1,28*28).to(device)
        labels=labels.reshape(-1,).to(device)
        preds=model(imgs)
        loss=loss_fn(preds,labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        tloss+=loss
    return tloss/n
```

The function trains the model for one epoch. The code is similar to what we have seen in the binary classification, except that the labels are from 0 to 9, instead of two numbers (0 and 1).

We also define a val_epoch() function as follows:

```
def val_epoch():
    vloss=0
    for n,(imgs,labels) in enumerate(val_loader):
        imgs=imgs.reshape(-1,28*28).to(device)
        labels=labels.reshape(-1,).to(device)
        preds=model(imgs)
        loss=loss_fn(preds,labels)
        vloss+=loss
    return vloss/n
```

The function uses the model to make predictions on images in the validation set and calculate what's the average loss per batch of data.

We now train the multi-category classifier as follows:

```
for i in range(1,101):
```

```
    tloss=train_epoch()
    vloss=val_epoch()
    print(f"at epoch {i}, tloss is {tloss}, vloss is {vloss}")
    if stopper.stop(vloss)==True:
        break
```

We train a maximum of 100 epochs. In each epoch, we first train the model using the training set. We then calculate the average loss per batch in the validation set. We use the `EarlyStop()` class to determine if the training should stop, by looking at the loss in the validation set. The training stops if the loss hasn't improved in the last ten epochs. After 25 epochs, the training stops.

The training takes about 5 minutes if you use GPU training, longer than the training process in binary classification since we have more observations in the training set now (ten clothing items instead of just two).

The output from the model is a vector of ten numbers. We use `torch.argmax()` to assign each observation a label based on the highest probability. We then compare the predicted label with the actual label.

To illustrate how the prediction works, we look at the predictions on the first five images in the test set:

Listing 2.10 Test the trained model on five images

```
plt.figure(dpi=300,figsize=(5,1))
for i in range(5):    #A
    ax=plt.subplot(1,5, i + 1)
    img=test_set[i][0]
    label=test_set[i][1]
    img=img/2+0.5
    img=img.reshape(28, 28)
    plt.imshow(img, cmap="binary")
    plt.axis('off')
    plt.title(text_labels[label]+f"; {label}", fontsize=8)
plt.show()
for i in range(5):
    img,label = test_set[i]    #B
    img=img.reshape(-1,28*28).to(device)
    pred=model(img)    #C
    index_pred=torch.argmax(pred,dim=1)    #D
```

```
    idx=index_pred.item()
    print(f"the label is {label}; the prediction is {idx}")    #E
```

We plot the first five clothing items in the test set in a five by one grid. We then use the trained model to make a prediction on each clothing item. The prediction is a tensor with ten values. The `torch.argmax()` method returns the position of the highest probability in the tensor and we use it as the predicted label. Finally, we print out both the actual label and the predicted label to compare and see if the predictions are correct.

After running the above code cell, you should see an image as shown in figure 2.3.

Figure 2.3 The first five clothing items in the test dataset and their respective labels. Each clothing item has a text label and a numerical label between 0 and 9.

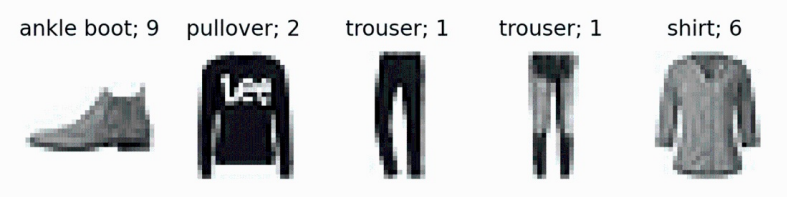


Figure 2.3 shows that the first five clothing items in the test set are ankle boot, pullover, trouser, trouser, and shirt, respectively, with numerical labels 9, 2, 1, 1, and 6.

The output after running the above code cell is as follows:

```
the label is 9; the prediction is 9
the label is 2; the prediction is 2
the label is 1; the prediction is 1
the label is 1; the prediction is 1
the label is 6; the prediction is 6
```

The above output shows that the model has made correct predictions on all five clothing items.

Fix the random state in PyTorch

The torch.manual_seed() method fixes the random state so results are the

same when you rerun your programs. However, you may get different results from those reported in this chapter even if you use the same random seed. This happens because different hardware and different versions of PyTorch handle floating point operations slightly differently. See, e.g., the explanations here https://discuss.pytorch.org/t/different-training-results-on-different-machines-with-simplified-test-code/59378/3. The difference is generally minor, though. So no need to be alarmed.

Next, we calculate the accuracy of the predictions on the whole test dataset as follows:

Listing 2.11 Testing the trained multi-category classification model

```
results=[]

for imgs,labels in test_loader:    #A
    imgs=imgs.reshape(-1,28*28).to(device)
    labels=(labels).reshape(-1,).to(device)
    preds=model(imgs)    #B
    pred10=torch.argmax(preds,dim=1)    #C
    correct=(pred10==labels)    #D
    results.append(correct.detach().cpu().numpy().mean())

accuracy=np.array(results).mean()    #E
print(f"the accuracy of the predictions is {accuracy}")
```

The output says:

```
the accuracy of the predictions is 0.7596536624203821
```

We iterate through all clothing items in the test set and use the trained model to make predictions. We then compare the predictions with the actual labels. The accuracy is about 76% in the out-of-sample test. Given that a random guess has an accuracy of about 10%, a 76% accuracy is fairly high. This indicates that we have built and trained two successful deep-learning models in PyTorch! You'll use these skills quite often later in this book. For example, in Chapter 3, the discriminator network you'll construct is essentially a binary classification model, similar to what you have created in this chapter.

2.5 Summary

- In PyTorch, we use tensors to hold various forms of input data so we can feed them to deep learning models.
- You can index and slice PyTorch tensors, reshape them, and conduct mathematical operations on them.
- Deep learning is a type of machine learning method that uses deep artificial neural networks to learn the relation between input and output data.
- Activation functions decide whether a neuron should be turned on or not based on the weighted sum. They introduce nonlinearity to the output of a neuron.
- Loss functions measure how well a machine learning model performs. The training of a model involves adjusting parameters to minimize the loss function.
- Binary classification is a machine learning model to classify observations into one of two categories.
- Multi-category classification is a machine learning model to classify observations into one of multiple categories.

[1] There are plenty of great ML books out there for you to choose from. Examples include Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow (2019, O'Reilly) and Machine Learning, Animated (2023, CRC Press). Both books use TensorFlow to create neural networks. If you prefer a book that uses PyTorch, I recommend Deep Learning with PyTorch (2020, Manning Publications).

[2] For example, I got the text labels here https://github.com/pranay414/Fashion-MNIST-Pytorch

[3] Later in this chapter when we discuss multi-category classification, you'll also learn to create a validation set to determine when to stop training.

[4] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2014, https://arxiv.org/abs/1412.6980

3 Generative Adversarial Networks (GANs): Shape and Number Generation

This chapter covers

- Creating a generator network and a discriminator network in GANs from scratch
- Using GANs to generate data points to form shapes (e.g., exponential growth curve)
- Generating a sequence of integers that are all multiples of five
- Learning how to train, save, load, and use GANs
- Assessing the performance of GANs and deciding when to stop training

Close to half of the generative models in this book belong to a category called Generative Adversarial Networks (GANs). The method was first proposed by Ian Goodfellow and his coauthors in 2014.[1] GANs, celebrated for their ease of implementation and versatility, empower individuals with even rudimentary knowledge of deep learning to construct their models from the ground up. The word "adversarial" in GAN refers to the fact that the two neural networks compete against each other in a zero-sum game framework. The generative network tries to create data instances indistinguishable from real samples. In contrast, the discriminative network tries to identify the generated samples from real ones. These versatile models can generate various content, from geometric shapes and intricate patterns to high-quality color images like human faces, and even realistic-sounding musical compositions.

In this chapter, I'll briefly review the theory behind Generative Adversarial Networks. Then, I'll show you how to implement that knowledge in PyTorch. You'll learn to build your first GAN from scratch so that all the details are demystified. To make the example relatable, imagine you put $1 in a savings account that pays 8% a year. You want to find out the balance in your

account based on the number of years you have invested. The true relation is an exponential growth curve. You'll learn to use GANs to generate data samples, pairs of values (x, y), that form such an exponential growth curve, with a mathematical relation $y=1.08^x$. Armed with this skill, you'll be able to generate data to mimic any shape: sine, cosine, quadratic, and so on.

In the second project in this chapter, you'll learn how to use GANs to generate a sequence of numbers that are all multiples of five. But you can change the pattern to multiples of two, three, seven, …, or other patterns. Along the way, you'll learn how to create a generator network and a discriminator network from scratch. You'll learn how to train, save, and use GANs. Further, you'll also learn to assess the performance of GANs either by visualizing samples generated by the generator network or by measuring the divergence between the generated sample distribution and the real data distribution.

Imagine that you need data to train a machine learning (ML) model to predict the relation between pairs of values (x, y). However, the training dataset is costly and time-consuming for human beings to prepare by hand. GANs can be well-suited to generate data in such cases: while the generated values of x and y generally conform to a mathematical relation, there is also noise in the generated data. The noise can be useful for preventing overfitting when the generated data is used to train the ML model.

In later chapters, you will build on the fundamental GAN architecture you learned here to generate other content such as high-resolution images and realistic-sounding music.

3.1 Steps Involved in Training GANs

In Chapter 1, you have seen a high-level overview of the theories behind GANs. In this section, I'll provide a summary of the steps involved in GANs in general and in creating data points to form an exponential growth curve in particular.

Imagine this: you plan to invest in a savings account that pays 8% annual interest. You put $1 in the account today and want to know how much money

you'll have in the account in the future.

The amount in your account in the future, y, depends on how long you invest in the savings account. Let's denote the number of years you invest by x, which can be a number, say, between 0 and 50. For example, if you invest for one year, the balance is $1.08; if you invest for two years, the balance is 1.08^2=$1.17. To generalize, the relationship between x and y is $y=1.08^x$. The function depicts an exponential growth curve. Note here that x can be a whole number such as 1 or 2, as well as a decimal number such as 1.14 or 2.35 and the formula still works.

Training GANs to generate data points that conform to a specific mathematical relation, like the example you have seen above, is a multi-step process. In your case, you want to generate data points (x, y) such that $y=1.08^x$. Below are the steps you should follow. When you generate other content such as a sequence of integers, images, or music, you follow similar steps, as you'll see in the second project in this chapter, as well as in other GAN models later in this book.

Figure 3.1 A diagram of the steps involved in training GANs to generate an exponential growth curve and the dual-network architecture in GANs. The generator obtains a random noise vector Z from the latent space (top left) to create a fake sample and presents it to the discriminator (middle). The discriminator classifies a sample as real or fake (created by the generator). The predictions are compared to the ground truth and both the discriminator and the generator learn from the predictions. After many iterations of training, the generator learns to create shapes that are indistinguishable from real samples.

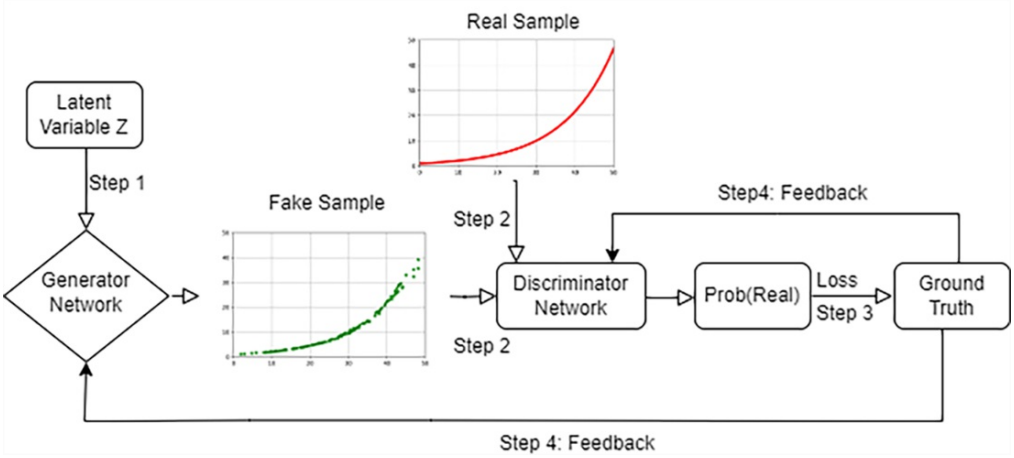


Figure 3.1 provides a diagram of the architecture of GANs and the steps

involved in generating an exponential growth curve. Before we start, we need to obtain a training dataset to train GANs. In our running example, we'll generate a dataset of (x, y) pairs using the mathematical relation $y=1.08^x$. We use the savings account example so that the numbers are relatable. The techniques you learn in this chapter can be applied to other shapes: sine, cosine, U-shape, and so on. You can choose a range of x values (say, 0 to 50) and calculate the corresponding y values. Since we usually train models in batches of data in deep learning, the number of observations in your training dataset is usually set to a multiple of the batch size. A real sample is located at the top of figure 3.1, which has an exponential growth curve shape.

Once you have the training set ready, you need to create two networks in GANs: a generator and a discriminator. The generator, located at the bottom left of figure 3.1, takes a random noise vector Z as the input and generates data points (step 1 of our training loop). The random noise vector Z used by the generator is obtained from the latent space, which represents the range of possible outputs the GAN can produce and is central to the GAN's ability to generate diverse data samples. In Chapter 5, we'll explore the latent space to select the attributes of the content created by the generator. The discriminator, located at the center of figure 3.1, evaluates whether a given data point (x, y) is real (from the training dataset) or fake (created by the generator); and this is step 2 of our training loop.

To know how to adjust model parameters, we must choose the right loss functions. We need to define the loss functions for both the generator and discriminator. For the generator, the loss function encourages it to generate data points that resemble data points from the training dataset, making the discriminator classify them as real. For the discriminator, the loss function encourages it to correctly classify real and generated data points.

In each iteration of the training loop, we alternate between training the discriminator and the generator. During each training iteration, we sample a batch of real (x, y) data points from the training dataset and a batch of fake data points generated by the generator. When training the discriminator, we compare the predictions by the discriminative model, which is a probability that the sample is from the training set, with the ground truth, which is 1 if the sample is real and 0 if the sample is fake (shown at the right of figure

3.1); this constitutes half of step 3 in the training loop. We adjust the weights in the discriminator network slightly so that in the next iteration, the predicted probability moves closer to the ground truth (half of step 4 in our training loop). When training the generator, we feed fake samples to the discriminative model and obtain a probability that the sample is real (the other half of step 3). We then adjust the weights in the generator network slightly so that in the next iteration, the predicted probability moves closer to 1 (since the generator wants to create samples to fool the discriminator into thinking they are real); this constitutes the other half of step 4. We repeat this process for many iterations, making the generator network create more realistic data points.

A natural question is when to stop training the GANs. For that, you evaluate the GAN's performance by generating a set of synthetic data points and comparing them to the real data points from the training dataset. In most cases, we use visualization techniques to assess how well the generated data conform to the desired relation. However, in our running example, since we know the distribution of the training data, we can calculate the Mean Squared Error (MSE) between the generated data and the true data distribution. We stop training GANs when the generated samples stop improving their qualities after a fixed number of rounds of training.

At this point, the model is considered trained. We then discard the discriminator and keep the generator. To create an exponential growth curve, we feed a random noise vector Z to the trained generator and obtain pairs of (x, y) to form the desired shape.

3.2 Preparing Training Data

In this section, you'll create the training dataset so that you can use it to train the GAN model later in this chapter. Specifically, you'll create pairs of data points (x, y) that conform to the exponential growth shape. You'll place them in batches so that they are ready to be fed to deep neural networks.

NOTE

The code for this chapter, as well as other chapters in this book, are available

at the book's GitHub repository https://github.com/markhliu/DGAI.

3.2.1 A Training Dataset that Forms an Exponential Growth Curve

We'll create a dataset that contains many observations of data pairs, (x, y), where x is uniformly distributed in the interval [0, 50] and y is related to x based on the formula $y=1.08^x$, like so:

Listing 3.1 Creating training data to form an exponential growth shape

```
import torch

torch.manual_seed(42)    #A

train_data=torch.zeros((2048, 2))    #B

train_data[:,0]=50*torch.rand(2048)    #C

train_data[:,1]= 1.08**train_data[:,0]    #D
```

First, we create 2048 values of x between 0 and 50, using the `torch.rand()` method. We use the `manual_seed()` method in PyTorch to fix the random state so that all results are reproducible. We first create a PyTorch tensor, `train_data`, with 2048 rows and two columns. The values of x are placed in the first column in the tensor `train_data`. The `rand()` method in PyTorch generates random values between 0.0 and 1.0. By multiplying the value by 50, the resulting values of x are between 0.0 and 50.0. We then fill the second column of `train_data` with values of $y=1.08^x$.

Exercise 3.1

Modify listing 3.1 so that the relation between x and y is y=sin(x) by using the `torch.sin()` function.

We plot the relation between x and y by using the *matplotlib* library:

Listing 3.2 Visualizing the relation between x and y

```
import matplotlib.pyplot as plt

fig=plt.figure(dpi=300,figsize=(8,6))
plt.plot(train_data[:,0],train_data[:,1],".",c="r")    #A
plt.xlabel("values of x",fontsize=15)
plt.ylabel("values of $y=1.08^x$",fontsize=15)    #B
plt.title("An exponential growth shape",fontsize=20)    #C
plt.show()
```

You will see an exponential growth curve shape after running listing 3.2, which is similar to the top graph in figure 3.1.

Exercise 3.2

Modify listing 3.2 to plot the relation between x and y=sin(x) based on your changes in exercise 3.1. Make sure you change the y-axis label and the title in the plot to reflect the changes you made.

3.2.2 Preparing the Training Dataset

We'll place the data samples you just created into batches so that we can feed them to the discriminator network. We use the `DataLoader()` class in PyTorch to wrap an iterable around the training dataset so that we can easily access the samples during training, like so:

```
from torch.utils.data import DataLoader

batch_size=128
train_loader=DataLoader(
    train_data,
    batch_size=batch_size,
    shuffle=True)
```

Make sure you select the total number of observations and the batch size so that all batches have the same number of samples in them. We chose 2048 observations with a batch size of 128. As a result, we have 2048/128=16 batches. The `shuffle=True` argument in `DataLoader()` shuffles the observations randomly before dividing them into batches.

NOTE

Shuffling makes sure that the data samples are evenly distributed and observations within a batch are not correlated, which, in turn, stabilizes training. In this specific example, shuffling makes sure that values of x fall randomly between 0 and 50, instead of clustering in a certain range, say, between 0 and 5.

You can access a batch of data by using the `next()` and `iter()` methods, like so:

```
batch0=next(iter(train_loader))
print(batch0)
```

You will see 128 pairs of numbers (x, y), where the value of x falls randomly between 0 and 50. Further, the values of x and y in each pair conform to the relation $y=1.08^x$.

3.3 Creating Generative Adversarial Networks (GANs)

Now that the training dataset is ready, we'll create a discriminator network and a generator network.

The discriminator network is a binary classifier, which is very similar to the binary classifier for clothing items we have created and trained in Chapter 2. Here, the discriminator's job is to classify the samples into either real or fake.

The generator network, on the other hand, tries to create data points (x, y) that are indistinguishable from those in the training set so that the discriminator will classify them as real.

3.3.1 The Discriminator Network

We use PyTorch to create a discriminator neural network. We'll use fully connected dense layers with ReLU activations. We'll also use dropout layers to prevent overfitting.

We create a sequential deep neural network in PyTorch to represent the

discriminator, as follows:

Listing 3.3 Creating a discriminator network

```
import torch.nn as nn

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

D=nn.Sequential(
    nn.Linear(2,256),    #B
    nn.ReLU(),
    nn.Dropout(0.3),    #C
    nn.Linear(256,128),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(128,64),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(64,1),    #D
    nn.Sigmoid()).to(device)
```

Make sure that in the first layer, the input shape is 2 because, in our sample, each data instance has two values in it: x and y. The number of inputs in the first layer should always match with the size of the input data. Also, make sure that the number of output features is 1 in the last layer: the output of the discriminator network is a single value. We use the sigmoid activation function to squeeze the output to the range [0, 1] so that it can be interpreted as the probability, p, that the sample is real. With the complementary probability, 1-p, the sample is fake. This is very similar to what we have done in Chapter 2 when a binary classifier attempts to identify a piece of clothing item as either an ankle boot or a t-shirt.

The hidden layers have 256, 128, and 64 neurons in them, respectively. There is nothing magic about these numbers, and you can easily change them and have similar results, as long as they are in a reasonable range. If the number of neurons in hidden layers is too large, it may lead to overfitting of the model; if the number is too small, it may lead to underfitting. The number of neurons can be optimized separately using a validation set through hyperparameter tuning.

Dropout layers randomly deactivate (or "drop out") a certain percentage of

neurons in the layer to which they are applied. This means that these neurons do not participate in forward or backward passes during training. Overfitting occurs when a model learns not only the underlying patterns in the training data but also the noise and random fluctuations, leading to poor performance on unseen data. Dropout layers are an effective way to prevent overfitting.

3.3.2 The Generator Network

The generator's job is to create a pair of numbers, (x, y), so that it can pass the screening of the discriminator. That is, the generator is trying to create a pair of numbers to maximize the probability that the discriminator thinks that the numbers are from the training dataset (i.e., they conform to the relation $y=1.08^x$).

We create the following neural network to represent the generator:

Listing 3.4 Creating a generator network

```
G=nn.Sequential(
    nn.Linear(2,16),    #A
    nn.ReLU(),
    nn.Linear(16,32),
    nn.ReLU(),
    nn.Linear(32,2)).to(device)    #B
```

We feed a random noise vector from a two-dimensional latent space, (z_1, z_2), to the generator. The generator then generates a pair of values, (x, y), based on the input from the latent space. Here we use a two-dimensional latent space, but changing the dimension to other numbers such as five or ten wouldn't affect our results.

The meaning of the latent space

The latent space in a GAN is a conceptual space where each point can be transformed into a realistic data instance by the generator. This space represents the range of possible outputs the GAN can produce and is central to the GAN's ability to generate varied and complex data. The latent space acquires its significance exclusively when it is employed in conjunction with

the generative model. Within this context, one can interpolate between points in the latent space to affect the attributes of output, which we'll discuss in Chapter 5.

3.3.3 Loss Functions, Optimizers, and Early Stopping

Since the discriminator network is essentially performing a binary classification task (identifying a data sample as real or fake), we use binary cross-entropy loss, the preferred loss function in binary classifications for the discriminator network. The discriminator is trying to maximize the accuracy of the binary classification: identify a real sample as real and a fake sample as fake. The weights in the discriminator network are updated based on the gradient of the loss function with respect to the weights.

The generator, on the other hand, is trying to minimize the probability that the fake sample is being identified as fake. Therefore, we'll also use binary cross-entropy loss for the generator network: the generator updates its network weights so that the generated samples will be classified as real by the discriminator in a binary classification problem.

The loss function is the binary cross-entropy loss since this is a binary classification problem. As we have done in Chapter 2, we use the Adam optimizer as the gradient descent algorithm. We set the learning rate to 0.0005. Let's code those steps in by using PyTorch:

```
loss_fn=nn.BCELoss()
lr=0.0005
optimD=torch.optim.Adam(D.parameters(),lr=lr)
optimG=torch.optim.Adam(G.parameters(),lr=lr)
```

One remaining question before we get to the actual training is this: how many epochs should we train the GANs? How do we know the model is well trained so that the generator is ready to create samples that can mimic the exponential growth curve shape? If you recall, in Chapter 2, we split the training set further into a train set and a validation set. We then used the loss in the validation set to determine whether the parameters had converged so that we could stop training. However, GANs are trained using a different approach compared to traditional supervised learning models (such as the

classification models you have seen in Chapter 2). Since the quality of the generated samples improves throughout training, the discriminator's task becomes more and more difficult (in a way, the discriminator in GANs is making predictions on a moving target). The loss from the discriminator network is not a good indicator of the quality of the model.

One common method to measure the performance of GANs is through visual inspection. Humans can assess the quality and realism of generated data instances by simply looking at them. This is a qualitative approach but can be very informative. But in our simple case, since we know the exact distribution of the training dataset, we'll look at the mean squared error of the generated samples relative to samples in the training set and use it as a measure of the performance of the generator. Let's code that in:

```
mse=nn.MSELoss()    #A

def performance(fake_samples):
    real=1.08**fake_samples[:,0]    #B
    mseloss=mse(fake_samples[:,1],real)   #C
    return mseloss
```

We'll stop training the model if the performance of the generator doesn't improve in, say, 1,000 epochs.

Therefore, we define an early stopping class, as we did in Chapter 2, to decide when to stop training the model:

Listing 3.5 An early stopping class to decide when to stop training

```
class EarlyStop:
    def __init__(self, patience=1000):    #A
        self.patience = patience
        self.steps = 0
        self.min_gdif = float('inf')
    def stop(self, gdif):    #B
        if gdif < self.min_gdif:    #C
            self.min_gdif = gdif
            self.steps = 0
        elif gdif >= self.min_gdif:
            self.steps += 1
        if self.steps >= self.patience:    #D
            return True
```

```
        else:
            return False
stopper=EarlyStop()
```

With that, we have all the components we need to train our GANs, which we'll do in the next section.

3.4 Train and Use GANs for Shape Generation

Now that we have the training data and two networks, we'll train the model. After that, we'll discard the discriminator and use the generator to generate data points to form an exponential growth curve shape.

3.4.1 The Training of GANs

We first create labels for real samples and fake samples, respectively.

Specifically, we'll label all real samples as 1s and all fake samples as 0s. During the training process, the discriminator compares its own predictions with the labels to receive feedback so that it can adjust model parameters to make better predictions in the next iteration.

Below, we define two tensors, `real_labels` and `fake_labels`, as follows:

```
real_labels=torch.ones((batch_size,1))
real_labels=real_labels.to(device)

fake_labels=torch.zeros((batch_size,1))
fake_labels=fake_labels.to(device)
```

The tensor `real_labels` is two-dimensional with a shape of `(batch_size, 1)`; that is, 128 rows and 1 column. We use 128 rows because we'll feed a batch of 128 real samples to the discriminator network to obtain 128 predictions. Similarly, the tensor `fake_labels` is two-dimensional with a shape of `(batch_size, 1)`. We'll feed a batch of 128 fake samples to the discriminator network to obtain 128 predictions and compare them with the ground truth: 128 labels of 0s. We move the two tensors to the GPU for fast training if your computer has a CUDA-enabled GPU.

To train the GANs, we define a few functions so that the training loop looks organized. The first function, `train_D_on_real()`, trains the discriminator network with a batch of real samples.

Listing 3.6 Defining a train_D_on_real() function

```
def train_D_on_real(real_samples):
    real_samples=real_samples.to(device)
    optimD.zero_grad()
    out_D=D(real_samples)    #A
    loss_D=loss_fn(out_D,real_labels)    #B
    loss_D.backward()
    optimD.step()    #C
    return loss_D
```

The function `train_D_on_real()` first moves the real samples to GPU if the computer has a CUDA-enabled GPU. The discriminator network, D, makes predictions on the batch of samples. The model then compares the discriminator's predictions, `out_D`, with the ground truth, `real_labels`, and calculates the loss of the predictions accordingly. The `backward()` method calculates the gradients of the loss function with respect to model parameters. The `step()` method adjusts the model parameters (that is, backpropagation). The `zero_grad()` method means that we explicitly set the gradients to zero before backpropagation. Otherwise, the accumulated gradients instead of the incremental gradients are used on every `backward()` call.

TIP

We call the `zero_grad()` method before updating model weights when training each batch of data. We explicitly set the gradients to zero before backpropagation to use incremental gradients instead of the accumulated gradients on every `backward()` call.

The second function, `train_D_on_fake()`, trains the discriminator network with a batch of fake samples.

Listing 3.7 Defining the train_D_on_fake() function

```
def train_D_on_fake():
    noise=torch.randn((batch_size,2))
```

```
    noise=noise.to(device)
    fake_samples=G(noise)     #A
    optimD.zero_grad()
    out_D=D(fake_samples)    #B
    loss_D=loss_fn(out_D,fake_labels)    #C
    loss_D.backward()
    optimD.step()    #D
    return loss_D
```

The function `train_D_on_real()` first feeds a batch of random noise vectors from the latent space to the generator to obtain a batch of fake samples. The function then presents the fake samples to the discriminator to obtain predictions. The function compares the discriminator's predictions, `out_D`, with the ground truth, `fake_labels`, and calculates the loss of the predictions accordingly. Finally, it adjusts the model parameters based on the gradients of the loss function with respect to model weights.

Note

We use the terms *weights* and *parameters* interchangeably. Strictly speaking, model parameters also include bias terms, but we use the term model weights loosely to include model biases. Similarly, we use the terms *adjusting weights*, *adjusting parameters*, and *backpropagation* interchangeably.

The third function, `train_G()`, trains the generator network with a batch of fake samples.

Listing 3.8 Defining the train_G() function

```
def train_G():
    noise=torch.randn((batch_size,2))
    noise=noise.to(device)
    optimG.zero_grad()
    fake_samples=G(noise)    #A
    out_G=D(fake_samples)    #B
    loss_G=loss_fn(out_G,real_labels)    #C
    loss_G.backward()
    optimG.step()     #D
    return loss_G, fake_samples
```

To train the generator, we first feed a batch of random noise vectors from the latent space to the generator to obtain a batch of fake samples. We then

present the fake samples to the discriminator network to obtain a batch of predictions. We compare the discriminator's predictions with `real_labels`, a tensor of 1s, and calculate the loss. It's important that we use a tensor of 1s, not a tensor of 0s, as the labels. Why? Because the objective of the generator is to fool the discriminator into thinking that fake samples are real. Finally, we adjust the model parameters based on the gradients of the loss function with respect to model weights so that in the next iteration, the generator can create more realistic samples.

Note

We use the tensor `real_labels` (a tensor of 1s) instead of `fake_labels` (a tensor of 0s) when calculating loss and assessing the generator network because the generator wants the discriminator to predict fake samples as real.

Finally, we define a function, `test_epoch()`, which prints out the losses for the discriminator and the generator periodically. Further, it plots the data points generated by the generator and compares them to those in the training set. The function `test_epoch()` is defined as follows:

Listing 3.9 Define the test_epoch() function

```
import os
os.makedirs("files", exist_ok=True)    #A

def test_epoch(epoch,gloss,dloss,n):
    if epoch==0 or (epoch+1)%25==0:
        g=gloss.item()/n
        d=dloss.item()/n
        print(f"at epoch {epoch+1}, G loss: {g}, D loss {d}")
        fake=fake_samples.detach().cpu().numpy()
        plt.figure(dpi=200)
        plt.plot(fake[:,0],fake[:,1],"*",c="g",
            label="generated samples")    #C
        plt.plot(train_data[:,0],train_data[:,1],".",c="r",
            alpha=0.1,label="real samples")    #D
        plt.title(f"epoch {epoch+1}")
        plt.xlim(0,50)
        plt.ylim(0,50)
        plt.legend()
        plt.savefig(f"files/p{epoch+1}.png")
        plt.show()
```

After every 25 epochs, the function prints out the average losses for the generator and the discriminator in the epoch. Further, it plots a batch of fake data points generated by the generator (in asterisks) and compares them to the data points in the training set (in dots). The plot is saved as an image in your local folder /files/.

Now, we are ready to train the model. We iterate through all batches in the training dataset. For each batch of data, we first train the discriminator using the real samples. After that, the generator creates a batch of fake samples, and we use them to train the discriminator again. Finally, we let the generator create a batch of fake samples again, but this time, we use them to train the generator instead. We train the model until the early stopping condition is satisfied, like so:

Listing 3.10 Training GANs to generate an exponential growth curve

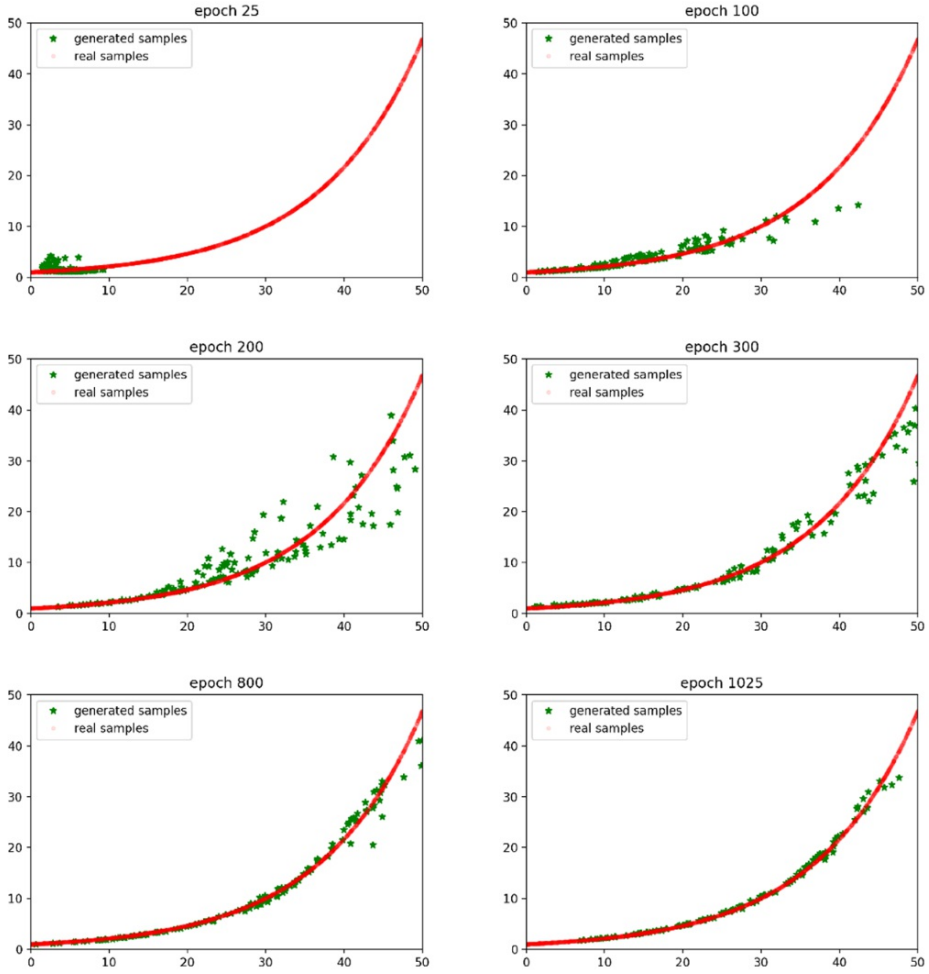
```
for epoch in range(10000):    #A
    gloss=0
    dloss=0
    for n, real_samples in enumerate(train_loader):    #B
        loss_D=train_D_on_real(real_samples)
        dloss+=loss_D
        loss_D=train_D_on_fake()
        dloss+=loss_D
        loss_G,fake_samples=train_G()
        gloss+=loss_G
    test_epoch(epoch,gloss,dloss,n)    #C
    gdif=performance(fake_samples).item()
    if stopper.stop(gdif)==True:    #D
        break
```

The training takes a few minutes if you are using GPU training. Otherwise, it may take 20-30 minutes, depending on the hardware configuration on your computer. Alternatively, you can download the trained model from the book’s GitHub repository.

After 25 epochs of training, the generated data are scattered around the point (0,0) and don't form any meaningful shape (an epoch is when all training data is used for training once). After 200 epochs of training, the data points start to form an exponential growth curve shape, even though many points are far away from the dotted curve, which is formed by points from the training set.

After 1025 epochs, the generated points fit closely with the exponential growth curve. Figure 3.2 provides subplots of the output from six different epochs. Our GANs work really well: the generator is able to generate data points to form the desired shape.

Figure 3.2 Subplots of the comparison of the generated shape with the true exponential growth curve shape at different stages of the training process. At epoch 25, the generated samples don't form any meaningful shape. At epoch 200, the samples start to look like an exponential growth curve shape. At epoch 1025, the generated samples align closely with the exponential growth curve.



3.4.2 Saving and Using the Trained Generator

Now that the GANs are trained, we'll discard the discriminator network, as we always do in GANs, and save the trained generator network in the local

folder, as follows:

```
scripted = torch.jit.script(G)
scripted.save('files/exponential.pt')
```

The `torch.jit.script()` method scripts a function or a `nn.Module` class as TorchScript code using the TorchScript compiler. We use the method to script our trained generator network and save it as a file, `exponential.pt`, on your computer.

To use the generator, we don't even need to define the model. We simply load up the saved file and use it to generate data points as follows:

```
new_G=torch.jit.load('files/exponential.pt',
                     map_location=device)
new_G.eval()
```

The generator is now loaded. We can use it to generate a batch of data points as follows:

```
noise=torch.randn((batch_size,2)).to(device)
new_data=new_G(noise)
```

Here, we first obtain a batch of random noise vectors from the latent space. We then feed them to the generator to produce the fake data. We can plot the generated data as follows:

```
fig=plt.figure(dpi=300)
plt.plot(new_data.detach().cpu().numpy()[:,0],
  new_data.detach().cpu().numpy()[:,1],"*",c="g",
        label="generated samples")    #A
plt.plot(train_data[:,0],train_data[:,1],".",c="r",
         alpha=0.1,label="real samples")    #B
plt.title("Inverted-U Shape Generated by GANs")
plt.xlim(0,50)
plt.ylim(0,50)
plt.legend()
plt.show()
```

You should see a plot similar to the last subplot in figure 3.2: the generated data samples closely resemble an exponential growth curve.

Congratulations, you have created and trained your very first GANs! Armed with this skill, you can easily change the code so that the generated data match other shapes such as sine, cosine, U-shape, and so on.

Exercise 3.3

Modify the programs in the first project so that the generator generates data samples to form a sine shape between 0 and 50. When you plot the data samples, set the value of y between -1.2 and 1.2.

3.5 Generating Numbers with Patterns

In this second project, you'll build and train GANs to generate a sequence of ten integers between 0 and 99, all of them multiples of five. The main steps involved are similar to those to generate an exponential growth curve, with the exception that the training set is not data points with two values (x, y). Instead, the training dataset is a sequence of integers that are all multiples of five between 0 and 99.

In this section, you'll first learn to convert the training data into a format that neural networks understand: one-hot variables. Further, you'll convert one-hot variables back to an integer between 0 and 99 so it's easy for human beings to understand. Hence you are essentially translating data between human-readable and machine-readable formats. After that, you'll create a discriminator and a generator and train the GANs. You'll also use early stopping to determine when the training is finished. You then discard the discriminator and use the trained generator to create a sequence of integers with the pattern you want.

3.5.1 What Are One-Hot Variables?

One-hot encoding is a technique used in machine learning and data preprocessing to represent categorical data as binary vectors. Categorical data consists of categories or labels, such as colors, types of animals, or cities, which are not inherently numeric. Machine learning algorithms typically work with numerical data, so converting categorical data into a numerical format is necessary.

Imagine you are working with a categorical feature, for example, the color of a house that can take values "Red," "Green," and "Blue." With one-hot encoding, each category is represented as a binary vector. You'll create three binary columns, one for each category. Color "Red" is one-hot encoded as [1, 0, 0], "Green" as [0, 1, 0], and "Blue" as [0, 0, 1]. Doing so preserves the categorical information without introducing any ordinal relationship between the categories. Each category is treated as independent.

Below, we define a `onehot_encoder()` function to convert an integer to a one-hot variable.

```
def onehot_encoder(position,depth):
    onehot=torch.zeros((depth,))
    onehot[position]=1
    return onehot
```

The function takes two arguments: the first argument, `position`, is the index at which the value is turned on as 1, and the second argument, `depth`, is the length of the one-hot variable. For example, if we print out the value of `onehot_encoder(1,5)`, like this:

```
print(onehot_encoder(1,5))
```

The result is as follows:

```
tensor([0., 1., 0., 0., 0.])
```

The result shows a five-value tensor with the second place (the index value of which is 1) turned on as 1 and the rest turned off as 0s.

Now that you understand how one-hot encoding works, you can convert any integer between 0 and 99 to a one-hot variable as follows:

```
def int_to_onehot(number):
    onehot=onehot_encoder(number,100)
    return onehot
```

Let's use the function to convert the number 75 to a 100-value tensor:

```
onehot75=int_to_onehot(75)
print(onehot75)
```

The output is:

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0
```

The result is a 100-value tensor with the 76th place (the index value of which is 75) turned on as 1 and all other positions turned off as 0s.

To function `int_to_onehot()` converts an integer into a one-hot variable. In a way, the function is translating human language into machine language.

Next, we want to translate machine language back to human language. Suppose we have a one-hot variable, how can we convert it into an integer that humans understand? The following function `onehot_to_int()` accomplishes that goal:

```
def onehot_to_int(onehot):
    num=torch.argmax(onehot)
    return num.item()
```

The function `onehot_to_int()` takes the argument `onehot` and converts it into an integer based on which position has the highest value.

Let's test the function to see what happens if we use the tensor `onehot75` we just created as the input:

```
print(onehot_to_int(onehot75))
```

The output is:

```
75
```

The result shows that the function converts the one-hot variable to an integer 75, which is the right answer. So, we know that the functions are defined properly.

Next, we'll build and train GANs to generate multiples of five.

3.5.2 GANs to Generate Numbers with Patterns

Our goal is to build and train a model so that the generator can generate a sequence of 10 integers, all multiples of five. We first prepare the training data and then convert them to machine-readable numbers in batches. Finally, we use the trained generator to generate the patterns we want.

For simplicity, we'll generate a sequence of 10 integers between 0 and 99. We'll then convert the sequence into ten machine-readable numbers.

The function below generates a sequence of 10 integers, all multiples of five:

```
def gen_sequence():
    indices = torch.randint(0, 20, (10,))
    values = indices*5
    return values
```

We first use the `randint()` method in PyTorch to generate ten numbers between 0 and 19. We then multiply them by five and convert them to PyTorch tensors. This creates ten integers that are all multiples of five.

Let's try to generate a sequence of training data as follows:

```
sequence=gen_sequence()
print(sequence)
```

The output is as follows:

```
tensor([60, 95, 50, 55, 25, 40, 70,  5,  0, 55])
```

The values in the above output are all multiples of five.

Next, we convert each number to a one-hot variable so that we can feed them to the neural network later.

```
import numpy as np

def gen_batch():
    sequence=gen_sequence()    #A
    batch=[int_to_onehot(i).numpy() for i in sequence]    #B
    batch=np.array(batch)
    return torch.tensor(batch)
batch=gen_batch()
```

The function `gen_batch()` above creates a batch of data so that we can feed them to the neural network for training purposes.

We also define a function `data_to_num()` to convert one-hot variables to a sequence of integers so that humans can understand the output:

```
def data_to_num(data):
    num=torch.argmax(data,dim=-1)    #A
    return num
numbers=data_to_num(batch)    #B
```

Next, we'll create two neural networks: one for the discriminator D and one for the generator G.

We'll build GANs to generate the desired pattern of numbers. Similar to what we did earlier in this chapter, we create a discriminator network, which is a binary classifier that tells fake samples from real samples. We also create a generator network to generate a sequence of ten numbers.

Here is the discriminator neural network:

```
D=nn.Sequential(
    nn.Linear(100,1),
    nn.Sigmoid()).to(device)
```

Since we'll convert integers into 100-value one-hot variables, we use 100 as the input size in the first `Linear` layer in the model. The last `Linear` layer has just one output feature in it, and we use the sigmoid activation function to squeeze the output to the range [0, 1] so it can be interpreted as the probability, p, that the sample is real. With the complementary probability 1-p, the sample is fake.

The generator's job is to create a sequence of numbers so that they can pass as real in front of the discriminator D. That is, G is trying to create a sequence of numbers to maximize the probability that D thinks that the numbers are from the training dataset.

We create the following neural network to represent the generator G:

```
G=nn.Sequential(
```

```
    nn.Linear(100,100),
    nn.ReLU()).to(device)
```

We'll feed random noise vectors from a 100-dimensional latent space to the generator. The generator then creates a tensor of 100 values based on the input. Note here that we use the `ReLU` activation function at the last layer so that the output is nonnegative. Since we are trying to generate 100 values of 0 or 1, nonnegative values are appropriate here.

As in the first project, we use the Adam optimizer for both the discriminator and the generator, with a learning rate of 0.0005:

```
loss_fn=nn.BCELoss()
lr=0.0005
optimD=torch.optim.Adam(D.parameters(),lr=lr)
optimG=torch.optim.Adam(G.parameters(),lr=lr)
```

Now that we have the training data and two networks, we'll train the model. After that, we'll discard the discriminator and use the generator to generate a sequence of ten integers.

3.5.3 Training the GANs to generate numbers with patterns

The training process for this project is very similar to that in our first project in which you generated an exponential growth shape.

We define a function `train_D_G()`, which is a combination of the three functions `train_D_on_real()`, `train_D_on_fake()`, and `train_G()` that we have defined for the first project. The function `train_D_G()` is in the Jupyter notebook for this chapter in the book's GitHub repository. Take a look at the function `train_D_G()` so you can see what minor changes we have made compared to the three functions we defined for the first project.

We use the same early stopping class that we defined for the first project so we know when to stop training. However, we have modified the `patience` argument to 800 when we instantiate the class, like so:

Listing 3.11 Training GANs to generate multiples of five

```
stopper=EarlyStop(800)    #A

mse=nn.MSELoss()
def distance(generated_data):    #B
    nums=data_to_num(generated_data)
    remainders=nums%5
    ten_zeros=torch.zeros((10,1)).to(device)
    mseloss=mse(remainders,ten_zeros)
    return mseloss

for i in range(10000):
    gloss=0
    dloss=0
    generated_data=train_D_G(D,G,loss_fn,optimD,optimG)    #C
    dis=distance(generated_data)
    if stopper.stop(dis)==True:
        break
    if i % 50 == 0:
        print(data_to_num(generated_data))    #D
```

We have also defined a `distance()` function to measure the difference between the training set and the generated data samples: it calculates the mean squared error of the remainder of each generated number when divided by five. The measure is zero when all generated numbers are multiples of five.

If you run the above code cell, you'll see the following output:

```
tensor([14, 34, 19, 89, 44,  5, 58,  6, 41, 87], device='cuda:0')
…
tensor([ 0, 80, 65,  0,  0, 10, 80, 75, 75, 75], device='cuda:0')
tensor([25, 30,  0,  0, 65, 20, 80, 20, 80, 20], device='cuda:0')
tensor([65, 95, 10, 65, 75, 20, 20, 20, 65, 75], device='cuda:0')
```

In each iteration, we generate a batch of ten numbers. We first train the discriminator D using real samples. After that, the generator creates a batch of fake samples, and we use them to train the discriminator D again. Finally, we let the generator create a batch of fake samples again, but we use them to train the generator G instead. We stop training if the generator network stops improving after 800 epochs since the last time the minimum loss was achieved. After every 50 epochs, we print out the sequence of ten numbers created by the generator so you can tell if they are indeed all multiples of five.

The output during the training process is shown above. In the first few hundred epochs, the generator still produces numbers that are not multiples of five. But after 900 epochs, all the numbers generated are multiples of five. The training process takes just a minute or so with GPU training. It takes less than ten minutes if you use CPU training. Alternatively, you can download the trained model from the book's GitHub repository.

3.5.4 Save and Use the Trained Model

We'll discard the discriminator and save the trained generator in the local folder, as follows:

```
scripted = torch.jit.script(G)
scripted.save('files/num_gen.pt')
```

We have now saved the generator to the local folder. To use the generator, we simply load up the model and use it to generate a sequence of integers as follows:

```
new_G=torch.jit.load('files/num_gen.pt',
                     map_location=device)    #A
new_G.eval()
noise=torch.randn((10,100)).to(device)    #B
new_data=new_G(noise)    #C
print(data_to_num(new_data))
```

The output is as follows:

```
tensor([40, 25, 65, 25, 20, 25, 95, 10, 10, 65], device='cuda:0')
```

The generated numbers are all multiples of five.

You can easily change the code to generate other patterns such as odd numbers, even numbers, multiples of three, and so on.

Exercise 3.4

Modify the programs in the second project so that the generator generates a sequence of ten integers that are all multiples of three.

Now that you know how GANs work, you'll be able to extend the idea behind GANs to other formats in later chapters, including high-resolution images and realistic-sounding music.

3.6 Summary

- GANs consist of two networks: a discriminator to distinguish fake samples from real samples, and a generator to create samples that are indistinguishable from those in the training set.
- The steps involved in GANs are preparing training data, creating a discriminator and a generator, training the model and deciding when to stop training, and finally, discarding the discriminator and using the trained generator to create new samples.
- The content generated by GANs depends on the training data. When the training dataset contains data pairs (x, y) that form an exponential growth curve, the generated samples are also data pairs that mimic such a shape. When the training dataset has sequences of numbers that are all multiples of five, the generated samples are also sequences of numbers, with multiples of five in them.
- GANs are versatile and capable of generating many different formats of content.

[1] Goodfellow et al, 2014, Generative Adversarial Nets, https://arxiv.org/abs/1406.2661

4 Image Generation with GANs

This chapter covers

- Creating an image from scratch by mirroring steps in the discriminator network.
- Understanding how a 2D convolutional operation works on an image.
- Learning how a 2D transposed convolutional operation inserts gaps between the output values and generates feature maps of a higher resolution.
- Building and training GANs to generate grayscale and color images.

You have successfully generated an exponential growth curve and a sequence of integers that are all multiples of five in Chapter 3. Now that you understand how GANs work, you are ready to apply the same skillsets to generate many other forms of content such as high-resolution color images and realistic-sounding music. However, this may be easier said than done (you know what they say, the devil is in the details). For example, exactly how can we make the generator conjure up realistic images out of thin air? That's the question we're going to tackle in this chapter.

A common approach for the generator to create images from scratch is to mirror steps in the discriminator network. In the first project in this chapter, your goal is to create grayscale images of clothing items such as coats, shirts, sandals, and so on. You learn to mirror the layers in the discriminator network when designing a generator network. In this project, only fully connected dense layers are used in both the generator and discriminator networks.

In the second project in this chapter, your goal is to create high-resolution color images of Anime faces. Like in the first project, the generator mirrors the steps in the discriminator network to conjure up images. However, high resolution color images in this project contain many more pixels than the low-resolution grayscale images in the first project. If we use fully connected dense layers only, the number of parameters in the model increases

enormously. This, in turn, makes learning slow and ineffective. We, therefore, turn to convolutional neural networks (CNNs). In CNNs, each neuron in a layer is connected only to a small region of the input. This local connectivity reduces the number of parameters, making the network more efficient. CNNs require fewer parameters than fully connected networks of similar size, leading to faster training times and lower computational costs. CNNs are also generally more effective at capturing spatial hierarchies in image data because they treat images as multi-dimensional objects instead of one-dimensional vectors.

To prepare you for the second project, we'll show you how convolutional operations work and how they downsample the input images and extract spatial features in them. You'll also learn concepts such as filter size, stride, and zero-padding and how they affect the degree of downsampling in CNNs. While the discriminator network uses convolutional layers, the discriminator mirrors these layers by using transposed convolutional layers (also known as deconvolution or upsampling Layers). You'll learn how transposed convolutional layers are used for upsampling to generate high-resolution feature maps.

To summarize, you'll learn how to mirror the steps in the discriminator network to create images from scratch in this chapter. In addition, you'll learn how convolutional layers and transposed convolutional layers work. After this chapter, you'll use convolutional layers and transposed convolutional layers to create high-resolution images in other settings later in this book (such as in feature transfers when training a CycleGAN to convert blond hair to black hair or in a Variational Autoencoder to generate high-resolution human face images).

4.1 GANs to Generate Grayscale Images of Clothing Items

Our goal in the first project is to train a model to generate gray-scale images of clothing items such as sandals, t-shirts, coats, and bags.

When you use GANs to generate images, you'll always start by obtaining training data. You'll then create a discriminator network from scratch. You'll

mirror steps in the discriminator network when creating a generator network. Finally, you'll train the GANs and use the trained model for image generation. Let's see how that works with a simple project that creates grayscale images of clothing items.

4.1.1 Training Samples and the Discriminator

The steps involved with preparing the training data are similar to what we have done in Chapter 2, with a few exceptions that I'll highlight below. To save time, I'll skip the steps you have seen before in Chapter 2 and refer you to the book's GitHub repository. Follow the steps in the Jupyter notebook for this chapter in the book's GitHub repository https://github.com/markhliu/DGAI so that you create a data iterator with batches.

There are 60,000 images in the training set. In Chapter 2, we split the training set further into a train set and a validation set. We used the loss in the validation set to determine whether the parameters had converged so that we could stop training. However, GANs are trained using a different approach compared to traditional supervised learning models (such as the classification models you have seen in Chapter 2). Since the quality of the generated samples improves throughout training, the discriminator's task becomes more and more difficult. The loss from the discriminator network is not a good indicator of the quality of the model. The usual approach to measure the performance of GANs is through visual inspection to assess the quality and realism of generated images. We can potentially compare the quality of generated samples with training samples and use methods such as the Inception Score to evaluate the performance of GANs.[1] However, researchers have documented the weaknesses of these measures.[2] In this chapter, we'll use visual inspections to check the quality of generated samples periodically and determine when to stop training.

The discriminator network is a binary classifier, which is similar to the binary classifier for clothing items we discussed in Chapter 2. Here the discriminator's job is to classify the samples into either real or fake.

We use PyTorch to create the following discriminator neural network D, like

so:

```
import torch.nn as nn

device="cuda" if torch.cuda.is_available() else "cpu"
D=nn.Sequential(
    nn.Linear(784, 1024),    #A
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(1024, 512),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(512, 256),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(256, 1),    #B
    nn.Sigmoid()).to(device)
```

The input size is 784 because each grayscale image has a size of 28 by 28 pixels in the training set. Because fully connected dense layers take only one-dimensional inputs, we flatten the images before feeding them to the model. The output layer has just one neuron in it: the output of the discriminator D is a single value. We use the sigmoid activation function to squeeze the output to the range [0, 1] so that it can be interpreted as the probability, p, that the sample is real. With complementary probability 1-p, the sample is fake.

Exercise 4.1

Modify the discriminator D so that the numbers of outputs in the first three layers are 1000, 500, and 200 instead of 1024, 512, and 256. Make sure the number of outputs in a layer matches the number of inputs in the next layer.

4.1.2 A Generator to Create Grayscale Images

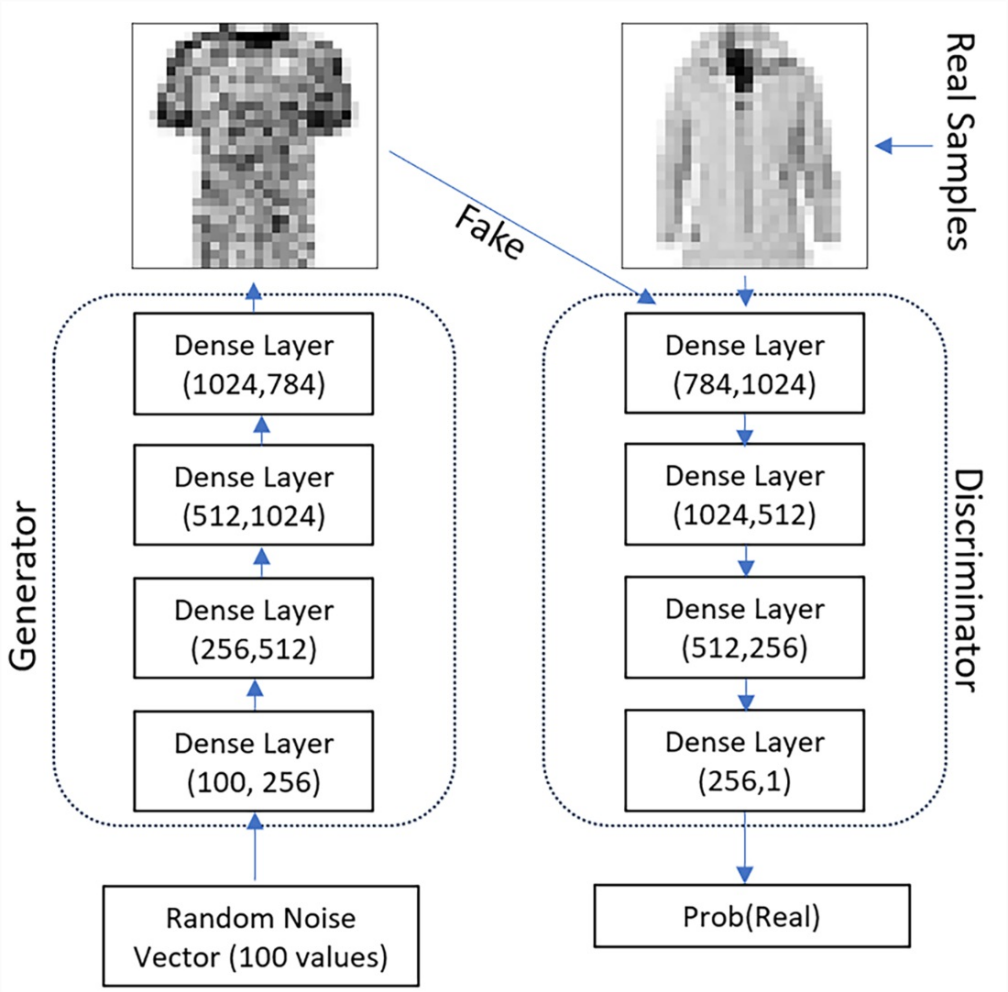
While the discriminator network is fairly easy to create, how to create a generator so that it can conjure up realistic images is a different matter. A common approach is to mirror the layers used in the discriminator network to create a generator, like so:

Listing 4.1 Designing a generator by mirroring layers in the discriminator

```
G=nn.Sequential(
    nn.Linear(100, 256),    #A
    nn.ReLU(),
    nn.Linear(256, 512),    #B
    nn.ReLU(),
    nn.Linear(512, 1024),    #C
    nn.ReLU(),
    nn.Linear(1024, 784),    #D
    nn.Tanh()).to(device)
```

Figure 4.1 provides a diagram of the architecture of generator and discriminator networks in the GAN to generate grayscale images of clothing items. As shown in the top right corner of figure 4.1, a flattened grayscale image from the training set, which contains 28*28=784 pixels, goes through four dense layers sequentially in the discriminator network and the output is the probability that the image is real. To create an image, the generator uses the same four dense layers, but in reverse order: it obtains a 100-value random noise vector from the latent space (bottom left in figure 4.1) and feeds the vector through the four dense layers. In each layer, the numbers of *inputs* and *outputs* in the discriminator are reversed and used as the numbers of *outputs* and *inputs* in the generator. Finally, the generator comes up with a 784-value tensor, which can be reshaped into a 28 by 28 grayscale image (top left).

Figure 4.1 Designing a generator network to create clothing items by mirroring the layers in the discriminator network. The right side of the diagram shows the discriminator network, which contains four fully connected dense layers. To design a generator that can conjure up clothing items from thin air, we mirror the layers in the discriminator network. Specifically, as shown on the left half of the figure, the generator has four similar dense layers in it, but in reverse order: the first layer in the generator mirrors the last layer in the discriminator, the second layer in the generator mirrors the second to last layer in the discriminator, and so on. Further, in each of the top three layers, the numbers of inputs and outputs in the discriminator are reversed and used as the numbers of outputs and inputs in the generator.



The left side of figure 4.1 is the generator network while the right side is the discriminator network. If you compare the two networks, you'll notice how the generator mirrors the layers used in the discriminator. Specifically, the generator has four similar dense layers in it, but in reverse order: the first layer in the generator mirrors the last layer in the discriminator, the second layer in the generator mirrors the second to last layer in the discriminator, and so on. The number of outputs of the generator is 784, with values between -1 and 1, and this matches the input to the discriminator network.

Exercise 4.2

Modify the generator G so that the numbers of outputs in the first three layers are 1000, 500, and 200 instead of 1024, 512, and 256. Make sure that the modified generator mirrors the layers used in the modified discriminator in Exercise 4.1.

As in GAN models we have seen in Chapter 3, the loss function is the binary cross-entropy loss since the discriminator D is performing a binary classification problem. We'll use the Adam optimizer for both the discriminator and the generator, with a learning rate of 0.0001, as follows:

```
loss_fn=nn.BCELoss()
lr=0.0001
optimD=torch.optim.Adam(D.parameters(),lr=lr)
optimG=torch.optim.Adam(G.parameters(),lr=lr)
```

Next, we'll train the GANs we just created by using the clothing item images in the training dataset.

4.1.3 Train GANs to Generate Images of Clothing Items

The training process is similar to what we have done in Chapter 3 when training GANs to generate an exponential growth curve or to generate a sequence of numbers that are all multiples of five.

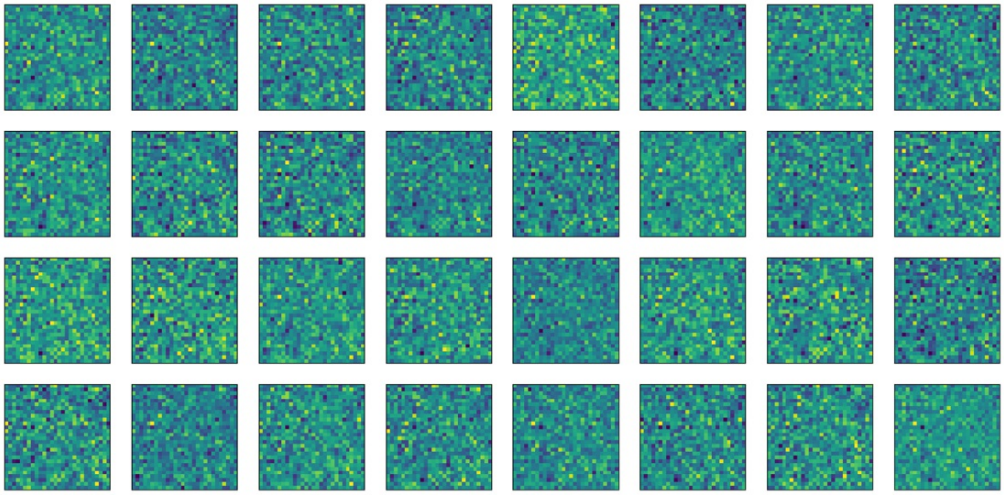
Unlike in Chapter 3, we'll solely rely on visual inspections to determine whether the model is well-trained.[3] For that purpose, we define a `see_output()` function to visualize the fake images created by the generator periodically:

Listing 4.2 Defining a function to visualize the generated clothing items

```
import matplotlib.pyplot as plt

def see_output():
    noise=torch.randn(32,100).to(device=device)
    fake_samples=G(noise).cpu().detach()    #A
    plt.figure(dpi=100,figsize=(20,10))
    for i in range(32):
        ax=plt.subplot(4, 8, i + 1)    #B
        img=(fake_samples[i]/2+0.5).reshape(28, 28)
        plt.imshow(img)    #C
        plt.xticks([])
        plt.yticks([])
    plt.show()

see_output()    #D
```

If you run the above code cell, you'll see 32 images that look like snowflake statics on a TV screen, as shown in figure 4.2. They don't look like clothing items at all because we haven't trained the generator yet.

Figure 4.2 Output from the GAN model to generate clothing items before training. Since the model is not trained, the generated images are nothing like the images in the training set.



To train the GAN model, we define a few functions: `train_D_on_real()`, `train_D_on_fake()`, and `train_G()`. They are similar to those defined in Chapter 3. Go to the Jupypter notebook for this chapter in the book's GitHub repository and see what minor modifications we have made.

Now, we are ready to train the model. We iterate through all batches in the training dataset. For each batch of data, we first train the discriminator using the real samples. After that, the generator creates a batch of fake samples, and we use them to train the discriminator again. Finally, we let the generator create a batch of fake samples again, but this time, we use them to train the generator instead. We train the model for 50 epochs, like so:

Listing 4.3 Training GANs for clothing item generation

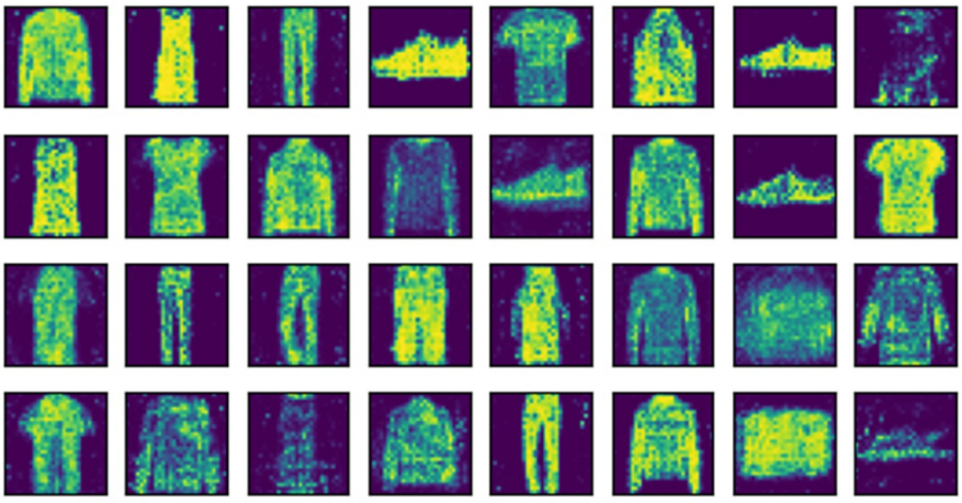
```
for i in range(50):
    gloss=0
    dloss=0
    for n, (real_samples,_) in enumerate(train_loader):
        loss_D=train_D_on_real(real_samples)    #A
        dloss+=loss_D
```

```
        loss_D=train_D_on_fake()    #B
        dloss+=loss_D
        loss_G=train_G()    #C
        gloss+=loss_G
    gloss=gloss/n
    dloss=dloss/n
    if i % 10 == 9:
        print(f"at epoch {i+1}, dloss: {dloss}, gloss {gloss}")
        see_output()    #D
```

The training takes about ten minutes if you are using GPU training. Otherwise, it may take an hour or so, depending on the hardware configuration on your computer. Or you can download the trained model from my website https://gattonweb.uky.edu/faculty/lium/gai/fashion_gen.zip.

After every ten epochs of training, you can visualize the generated clothing items, as shown in figure 4.3. After just ten epochs of training, the model can already generate clothes items that clearly can pass as real: you can tell what they are. The first three items in the first row in figure 4.3 are clearly a coat, a dress, and a pair of trousers, for example. As training progresses, the quality of the generated images becomes better and better.

Figure 4.3 Clothing items generated by an image GAN model after ten epochs of training



As we do in all GANs, we discard the discriminator and save the trained generator to generate samples later, as follows:

```
scripted = torch.jit.script(G)
scripted.save('files/fashion_gen.pt')
```

We have now saved the generator in the local folder. To use the generator, we load up the model like this:

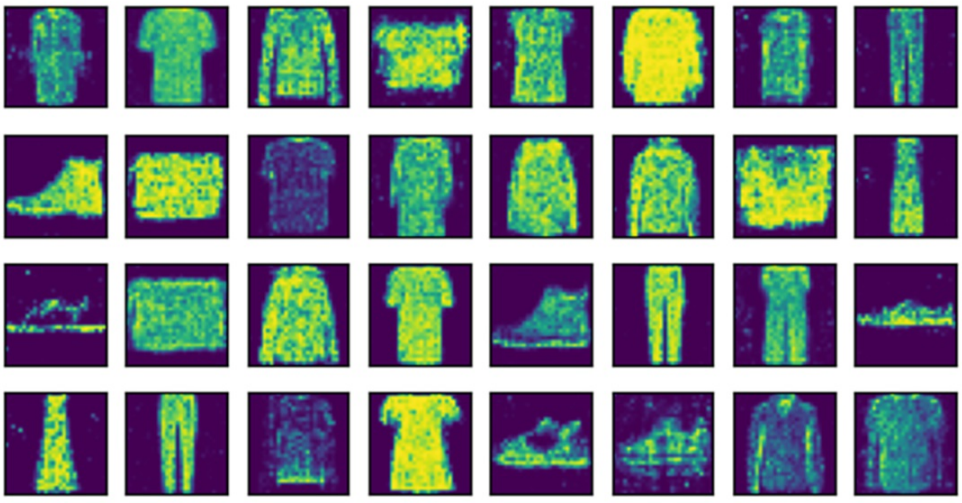
```
new_G=torch.jit.load('files/fashion_gen.pt',
                     map_location=device)
new_G.eval()
```

The generator is now loaded. We can use it to generate clothes items as follows:

```
noise=torch.randn(32,100).to(device=device)
fake_samples=new_G(noise).cpu().detach()
for i in range(32):
    ax = plt.subplot(4, 8, i + 1)
    plt.imshow((fake_samples[i]/2+0.5).reshape(28, 28))
    plt.xticks([])
    plt.yticks([])
plt.subplots_adjust(hspace=-0.6)
plt.show()
```

The generated clothing items are shown in figure 4.4.

Figure 4.4 Clothing items generated by a trained image GAN model (after 50 epochs)



The clothing items are fairly close to those in the training set, as you can see in figure 4.4.

Now that you have learned how to create grayscale images by using GANs, you'll learn how to generate high-resolution color images by using deep convolutional GAN (DCGAN) in the remaining sections of this chapter.

4.2 Convolutional Layers

To create high-resolution color images, we need more sophisticated techniques than simple fully-connected neural networks. Specifically, we'll use convolutional neural networks (CNNs), which are particularly effective for processing data with a grid-like topology, such as images. They are distinct from fully connected (dense) layers in a couple of ways. First, in CNNs, each neuron in a layer is connected only to a small region of the input. This is based on the understanding that in image data, local groups of pixels are more likely to be related to each other. This local connectivity reduces the number of parameters, making the network more efficient. Second, CNNs use the concept of shared weights — the same weights are used across different regions of the input. This is akin to sliding a filter across the entire input space. This filter detects specific features (e.g., edges or textures) regardless of their position in the input, leading to the property of translation invariance.

Due to their structure, CNNs are more efficient for image processing. They require fewer parameters than fully connected networks of similar size, leading to faster training times and lower computational costs. They are also generally more effective at capturing spatial hierarchies in image data.

Convolutional layers and transposed convolutional layers are two fundamental building blocks in CNNs, commonly used in image processing and computer vision tasks. They have different purposes and characteristics: convolutional layers are used for feature extraction. They apply a set of learnable filters (also known as kernels) to the input data to detect patterns and features at different spatial scales. These layers are essential for capturing hierarchical representations of the input data. In contrast, transposed convolutional layers are used for upsampling or generating high-resolution feature maps.

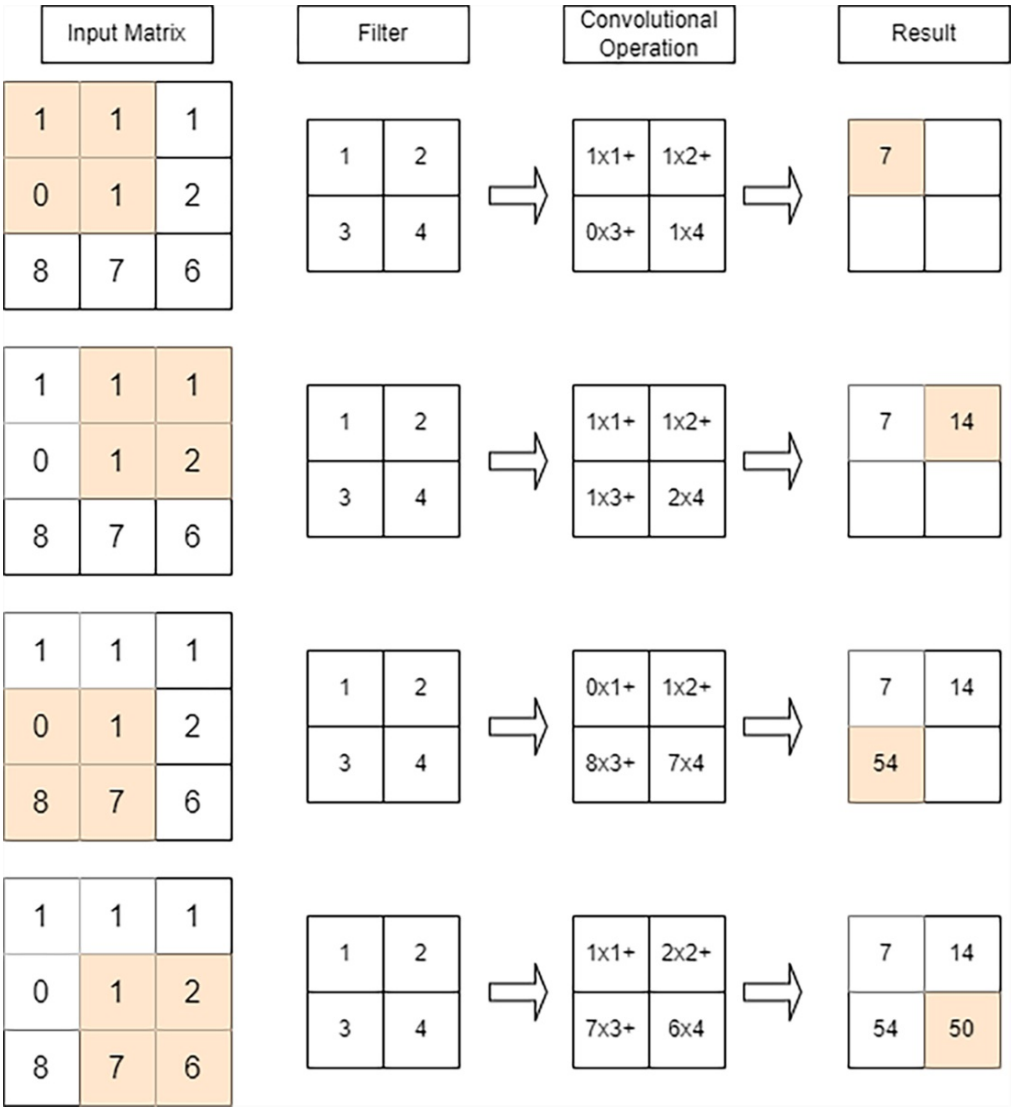
In this section, you'll learn how convolutional operations work and how kernel size, stride, and zero-padding affect convolutional operations.

4.2.1 How do convolutional operations work?

Convolutional layers use filters to extract spatial patterns on the input data. A convolutional layer is capable of automatically detecting a large number of patterns and associating them with the target label. Therefore, convolutional layers are commonly used in image classification tasks.

Convolutional operations involve applying a filter to an input image to produce a feature map. This process involves element-wise multiplication of the filter with the input image and summing the results. The weights in the filter are the same as it moves on the input image to scan different areas. Figure 4.5 shows a numerical example of how convolutional operations work. The left column in figure 4.5 is the input image, and the second column is a filter (a 2x2 matrix). Convolutional operations (the third column) involve sliding the filter over the input image, multiplying corresponding elements, and summing them up (the last column).

Figure 4.5 A numerical example of how convolutional operations work.



To have a deep understanding of exactly how convolutional operations work, let's implement the convolutional operations in PyTorch in parallel so that you can verify the numbers as shown in figure 4.5. First, let's create a PyTorch tensor to represent the input image in figure 4.5, as follows:

```
img = torch.Tensor([[1,1,1],
                    [0,1,2],
                    [8,7,6]]).reshape(1,1,3,3)    #A
```

The image is reshaped so that it has a dimension of (1, 1, 3, 3), indicating that there is just one observation in the batch, and the image has just one color channel. The height and the width of the image are both 3 pixels.

Let’s represent the 2 by 2 filter, as shown in the second column of figure 4.5, by creating a 2D convolutional layer in PyTorch, as follows:

```
conv=nn.Conv2d(in_channels=1,
            out_channels=1,
            kernel_size=2,
            stride=1)    #A
sd=conv.state_dict()    #B
print(sd)
```

A 2D convolutional layer takes several arguments. The `in_channels` argument is the number of channels in the input image. This value is 1 for grayscale images and 3 for color images since color images have three color channels (RGB: red, green, and blue). The `out_channels` is the number of channels after the convolutional layer, which can take any number, based on how many features you want to extract from the image. The `kernel_size` argument controls the size of the kernel; for example, `kernel_size=3` means the filter has a shape of 3 by 3, and `kernel_size=4` means the filter has a shape of 4 by 4. We set the kernel size to 2 so the filter has a shape of 2 by 2.

A 2D convolutional layer also has several optional arguments. The `stride` argument specifies how many pixels to move to the right or down each time the filter moves along the input image. The `stride` argument has a default value of 1. A higher value of stride leads to more downsampling of the image. The `padding` argument means how many rows of zeros to add to four sides of the input image, with a default value of 0. The `bias` argument indicates whether or not to add a learnable bias as the parameter, with a default value of `True`.

The above 2D convolutional layer has one input channel, one output channel, with a kernel size of 2 by 2, and a stride of 1. When the convolutional layer is created, the weights and the bias in it are randomly initialized. You will see the following output as the weights and the bias of the above convolutional layer:

```
OrderedDict([('weight', tensor([[[[ 0.3823,  0.4150],
          [-0.1171,  0.4593]]]])), ('bias', tensor([-0.1096]))])
```

To make our example easier to follow, we’ll replace the weights and the bias with whole numbers as follows:

```
weights={'weight':torch.tensor([[[[1,2],
   [3,4]]]]), 'bias':torch.tensor([0])}    #A
for k in sd:
    with torch.no_grad():
        sd[k].copy_(weights[k])    #B
print(conv.state_dict())    #C
```

Now the convolutional layer has weights and the bias that we have chosen. They also match the numbers in figure 4.5. The output from the above code cell is:

```
OrderedDict([('weight', tensor([[[[1., 2.],
          [3., 4.]]]])), ('bias', tensor([0.]))])
```

If we apply the above convolutional layer on the 3 by 3 image we mentioned above, what is the output? Let's find out like this:

```
output = conv(img)
print(output)
```

The output is:

```
tensor([[[[ 7., 14.],
          [54., 50.]]]], grad_fn=<ConvolutionBackward0>)
```

The output has a shape of (1, 1, 2, 2), with four values in it: 7, 14, 54, and 50. These numbers match those in figure 4.5

But how exactly does the convolutional layer generate the above output through the filter? We'll explain in detail below.

The input image is a 3 by 3 matrix, and the filter is a 2 by 2 matrix. When the filter scans over the image, it first covers the four pixels in the top left corner of the image, which has values [[1, 1], [0, 1]], as shown in the first row in figure 4.5. The filter has values [[1,2],[3,4]]. The convolution operation finds the sum of the element-wise multiplication of the two tensors (in this case, one tensor is the filter, and the other is the covered area). In other words, the convolution operation performs element-wise multiplication in each of the four cells, and then adds up the values in the four cells. Therefore, the output from scanning the top left corner is:

1*1+1*2+0*3+1*4=7.

This explains why the top left corner of the output has a value of 7. Similarly, when the filter is applied to the top right corner of the image, the covered area is [[1,1],[1,2]]. The output is therefore:

1*1+1*2+1*3+2*4=14.

This explains why the top right corner of the output has a value of 14.

Exercise 4.3

What are the values in the covered area when the filter is applied to the bottom right corner of the image? Explains why the bottom right corner of the output has a value of 50.

4.2.2 How do stride and padding affect convolutional operations?

Stride and zero padding are two important concepts in the context of convolutional operations. They play a crucial role in determining the dimensions of the output feature map and the way the filter interacts with the input data.

Stride refers to the number of pixels by which the filter moves across the input image. When the stride is 1, the filter moves one pixel at a time. A larger stride means the filter jumps over more pixels as it slides over the image. Increasing the stride reduces the spatial dimensions of the output feature map.

Zero padding involves adding layers of zeros around the border of the input image before applying the convolutional operation. Zero padding allows control over the spatial dimensions of the output feature map. Without padding, the dimensions of the output will be smaller than the input. By adding padding, you can preserve the dimensions of the input.

Let’s use an example to show how stride and padding work.

The code cell below redefines the 2D convolutional layer:

```
conv=nn.Conv2d(in_channels=1,
            out_channels=1,
            kernel_size=2,
            stride=2,    #A
            padding=1)    #B
sd=conv.state_dict()
for k in sd:
    with torch.no_grad():
        sd[k].copy_(weights[k])
output = conv(img)
print(output)
```

The output is:

```
tensor([[[[ 4.,  7.],
          [32., 50.]]]], grad_fn=<ConvolutionBackward0>)
```

The `padding=1` argument adds one row of 0s around the input image, so the padded image now has a size of five by five instead of three by three.

When the filter scans over the padded image, it first covers the top left corner, which has values [[0, 0], [0, 1]]. The filter has values [[1,2],[3,4]]. Therefore, the output from scanning the top left corner is:

0*1+0*2+0*3+1*4=4

This explains why the top left corner of the output has a value of 4. Similarly, when the filter slides two pixels down to the bottom left corner of the image, the covered area is [[0,0],[0,8]]. The output is therefore:

0*1+0*2+0*3+8*4=32

This explains why the bottom left corner of the output has a value of 32.

Exercise 4.4

What are the values in the covered area when the filter is applied to the top right corner of the padded image? Explains why the top right corner of the output has a value of 7.

4.3 Transposed Convolution and Batch Normalization

Transposed convolutional layers are also known as deconvolution or upsampling Layers. They are used for upsampling or generating high-resolution feature maps. They are often employed in generative models like GANs and Variational Autoencoders (VAEs).

Transposed convolutional layers apply a filter to the input data, but unlike standard convolution, they increase the spatial dimensions by inserting gaps between the output values, which effectively "upscales" the feature maps. This process generates feature maps of a higher resolution. Transposed convolutional layers help increase the spatial resolution, which is useful in image generation.

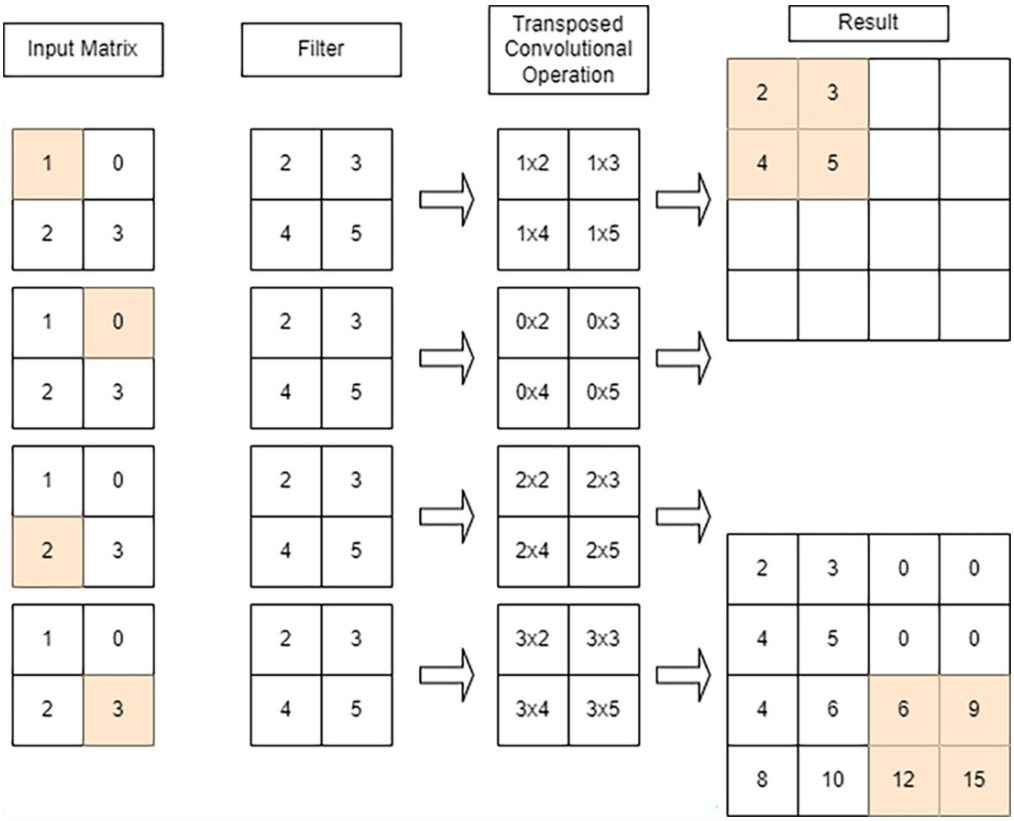
Strides can be used in transposed convolution layers to control the amount of upsampling. The greater the value of the stride, the more upsampling the transposed convolution layer has on the input data.

2D Batch Normalization is a technique used in neural networks, particularly Convolutional Neural Networks (CNNs), to stabilize and speed up the training process. It addresses several issues, including saturation, vanishing gradients, and exploding gradients, which are common challenges in deep learning. In this section, you'll look at some examples, so you have a deeper understanding of how it works. You'll use it when creating GANs to generate high-resolution color images in the next section.

4.3.1 How do transposed convolutional layers work?

Contrary to convolutional layers, transposed convolutional layers upsample and fill in gaps in an image to generate features and increase resolution by using kernels (i.e., filters). The output is usually larger than the input in a transposed convolutional layer. Therefore, transposed convolutional layers are essential tools when it comes to generating high-resolution images.

Figure 4.6 A numerical example of how transposed convolutional operations work.



To show you exactly how 2D transposed convolutional operations work, let's use a simple example and a figure to demonstrate. Suppose you have a very small 2 by 2 input image, as shown in the left column in figure 4.6. The input image has the following values in it:

```
img = torch.Tensor([[1,0],
                    [2,3]]).reshape(1,1,2,2)
```

You want to upsample the image so that it has higher resolutions. You can create a 2D transposed convolutional layer in PyTorch, as follows:

```
transconv=nn.ConvTranspose2d(in_channels=1,
            out_channels=1,
            kernel_size=2,
            stride=2)    #A
sd=transconv.state_dict()
weights={'weight':torch.tensor([[[[2,3],
   [4,5]]]]), 'bias':torch.tensor([0])}
for k in sd:
    with torch.no_grad():
        sd[k].copy_(weights[k])    #B
```

The above 2D transposed convolutional layer has one input channel, one output channel, with a kernel size of 2 by 2, and a stride of 2. The 2 by 2 filter is shown in the second column in figure 4.6. We replaced the randomly initialized weights and the bias in the layer with our handpicked whole numbers so it's easy to follow the calculations.

When the transposed convolutional layer is applied to the 2 by 2 image we mentioned above, what is the output? Let's find out like this:

```
transoutput = transconv(img)
print(transoutput)
```

The output is:

```
tensor([[[[ 2.,  3.,  0.,  0.],
          [ 4.,  5.,  0.,  0.],
          [ 4.,  6.,  6.,  9.],
          [ 8., 10., 12., 15.]]]], grad_fn=<ConvolutionBackward0>
```

The output has a shape of (1, 1, 4, 4), meaning we have upsampled a 2 by 2 image to a 4 by 4 image. How does the transposed convolutional layer generate the above output through the filter? We'll explain in detail below.

The image is a 2 by 2 matrix, and the filter is also a 2 by 2 matrix. When the filter is applied to the image, each element in the image multiplies with the filter and goes to the output. The top left value in the image is 1, and we multiply it with the values in the filter, [[2, 3], [4, 5]], and this leads to the four values in the top left block of the output matrix `transoutput` above, with values [[2, 3], [4, 5]], as shown at the top right corner in figure 4.6. Similarly, the bottom left value in the image is 2, and we multiply it with the values in the filter, [[2, 3], [4, 5]], and this leads to the four values in the bottom left block of the output matrix `transoutput` above, [[4, 6], [8, 10]].

Exercise 4.5

If the image has values [[10, 10], [15, 20]] in it, what is the output after you apply the 2D transposed convolutional layer `transconv` to the image? Assume `transconv` has values [[2, 3], [4, 5]] in it.

4.3.2 Batch Normalization

2D Batch Normalization is a standard technique in modern deep learning frameworks and has become a crucial component for training deep neural networks effectively. You'll see it quite often later in this book.

In 2D Batch Normalization, normalization is performed independently for each feature channel by adjusting and scaling values in the channel so they have a mean of zero and a variance of one. It ensures that the distribution of the inputs to layers deep in the network remains more stable during training. This stability arises because the normalization process reduces the internal covariate shift, which is the change in the distribution of network activations due to the update of weights in lower layers. It also helps in addressing the vanishing or exploding gradient problems by keeping the inputs in an appropriate range to prevent gradients from becoming too small (vanishing) or too large (exploding).

Here's how 2D batch normalization works: For each feature channel, we first calculate the mean and variance of all observations within the channel. We then normalize the values for each feature channel using the mean and variance obtained above (by subtracting the mean from each observation and then dividing the difference by the standard deviation). This ensures that the values in each channel have a mean of 0 and a standard deviation of 1 after normalization, which helps stabilize and speed up training. It also helps maintain stable gradients during backpropagation, which further aids in training deep neural networks.

Let's use a concrete example to show how the 2D batch normalization works.

Suppose that you have a 3-channel input with a size of 64 by 64. You pass the input through a 2D convolutional layer with 3 output channels as follows:

```
torch.manual_seed(42)    #A
img = torch.rand(1,3,64,64)    #B
conv = nn.Conv2d(in_channels=3,
            out_channels=3,
            kernel_size=3,
            stride=1,
            padding=1)    #C
```

```
out=conv(img)    #D
print(out.shape)
```

The output from the above code cell is:

```
torch.Size([1, 3, 64, 64])
```

We have created a three-channel input and passed it through a 2D convolutional layer with three output channels. The processed input has three channels with a size of 64 by 64 pixels.

Let’s look at the mean and standard deviation of the pixels in each of the three output channels:

```
for i in range(3):
    print(f"mean in channel {i} is", out[:,i,:,:].mean().item())
    print(f"std in channel {i} is", out[:,i,:,:].std().item())
```

The output is:

```
mean in channel 0 is -0.3766776919364929
std in channel 0 is 0.17841289937496185
mean in channel 1 is -0.3910464942455292
std in channel 1 is 0.16061744093894958
mean in channel 2 is 0.39275866746902466
std in channel 2 is 0.18207983672618866
```

The average values of the pixels in each output channel are not 0; the standard deviations of pixels in each output channel are not 1. Now, if we perform a 2D batch normalization, like so:

```
norm=nn.BatchNorm2d(3)
out2=norm(out)
print(out2.shape)
for i in range(3):
    print(f"mean in channel {i} is", out2[:,i,:,:].mean().item())
    print(f"std in channel {i} is", out2[:,i,:,:].std().item())
```

We have the following output:

```
torch.Size([1, 3, 64, 64])
mean in channel 0 is 6.984919309616089e-09
std in channel 0 is 0.9999650120735168
```

```
mean in channel 1 is -5.3085386753082275e-08
std in channel 1 is 0.9999282956123352
mean in channel 2 is 9.872019290924072e-08
std in channel 2 is 0.9999712705612183
```

The average values of pixels in each output channel are now practically 0 (or a very small number that is close to 0); the standard deviations of pixels in each output channel is now a number close to 1. That's what batch normalization does: it normalizes observations in each feature channel so that values in each feature channel have zero mean and unit standard deviation.

4.4 GANs to Generate Color Images of Anime Faces

In this second project, you'll learn how to create high-resolution color images.

The training steps in this project are similar to the first project, with the exception that the training data are color images of Anime faces. Further, the discriminator and generator neural networks are more sophisticated. We'll use 2D convolutional and 2D transposed convolutional layers in the two networks.

4.4.1 Download Anime Faces

You can download the training data from Kaggle https://www.kaggle.com/datasets/splcher/animefacedataset, which contains 63,632 color images of Anime faces. Extract the data from the zip file and put the images in a folder on your computer. For example, I placed the images in /files/anime/ on my computer.

Define the path name so you can use it later to load the images in Pytorch:

```
anime_path = r"files/anime"
```

Change the name of the path depending on where you have saved the images on your computer.

Next, we use the *ImageFolder* class in Torchvision *datasets* package to load the dataset, like so:

```
from torchvision import transforms as T
from torchvision.datasets import ImageFolder

transform = T.Compose([T.Resize((64, 64)),    #A
    T.ToTensor(),    #B
    T.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])])    #C
train_data = ImageFolder(root=anime_path,
                         transform=transform)    #D
```

We perform three different transformations when loading up the images from the local folder. First, we resize all images to 64 pixels in height and 64 pixels in width. Second, we convert the images to PyTorch tensors with values in the range [0, 1] by using the `ToTensor()` class. Finally, we use the `Normalize()` class to deduct 0.5 from the value and divide the difference by 0.5. As a result, the image data are now between -1 and 1.

We can now put the training data in batches:

```
from torch.utils.data import DataLoader

batch_size = 128
train_loader = DataLoader(dataset=train_data,
               batch_size=batch_size, shuffle=True)
```

The training dataset is now in batches, with a batch size of 128.

4.4.2 Channels-First Color Images in PyTorch

PyTorch uses a so-called channels-first approach when handling color images. This means the shape of images in PyTorch are (number_channels, height, width). In contrast, in other Python libraries such as TensorFlow or matplotlib, a channels-last approach is used: a color image has a shape of (height, width, number_channels) instead.

Let's look at an example image in our dataset and print out the shape of the image:

```
image0, _ = train_data[0]
print(image0.shape)
```

The output is:

```
torch.Size([3, 64, 64])
```

The shape of the first image is 3 by 64 by 64. This means the image has three color channels (RGB: red, green, and blue). The height and width of the image are both 64 pixels.

When we plot the images in *matplotlib*, we need to convert them to channels-last by using the `permute()` method in PyTorch, like so:

```
import matplotlib.pyplot as plt

plt.imshow(image0.permute(1,2,0)*0.5+0.5)
plt.show()
```

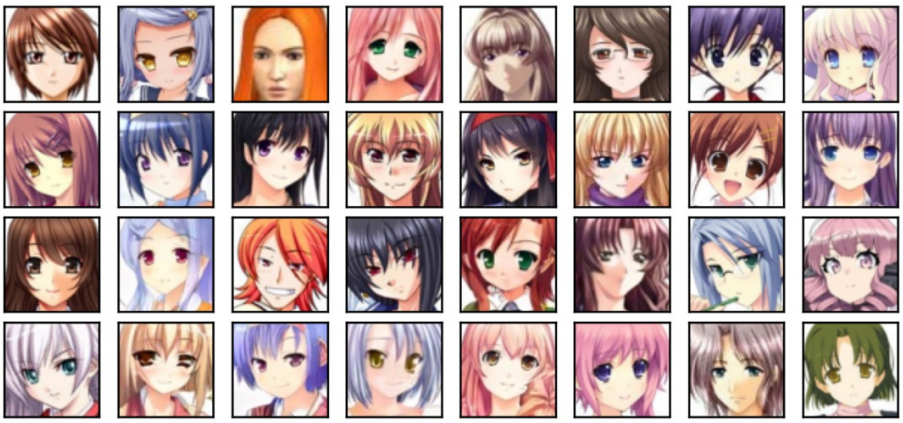
Note that we need to multiply the PyTorch tensor representing the image by 0.5 and then add 0.5 to it in order to convert the values from the range [-1, 1] to the range [0, 1]. You'll see a plot of an Anime face after running the above code cell.

Below, we define a function `plot_images()` to visualize 32 images in four rows and eight columns:

```
def plot_images(imgs):    #A
    for i in range(32):
        ax = plt.subplot(4, 8, i + 1)    #B
        plt.imshow(imgs[i].permute(1,2,0)/2+0.5)
        plt.xticks([])
        plt.yticks([])
    plt.subplots_adjust(hspace=-0.6)
    plt.show()

imgs, _ = next(iter(train_loader))    #C
plot_images(imgs)    #D
```

You'll see a plot of 32 Anime faces in a 4 by 8 grid after running the above code cell, as shown in figure 4.7.

Figure 4.7 Examples from the Anime faces training dataset.



4.5 Building a Deep Convolutional GAN (DCGAN)

In this section, you'll create a deep convolutional GAN (DCGAN) model so that we can train it to generate Anime face images. As usual, the GAN model consists of a discriminator network and a generator network. However, the networks are more sophisticated than the ones we have seen before: we'll use convolutional layers, transposed convolutional layers, and batch normalization layers in these networks.

We'll start with the discriminator network. After that, I'll explain how the generator network mirrors the layers in the discriminator network to conjure up realistic color images.

4.5.1 A Discriminator in DCGAN

As in previous GAN models we have seen, the discriminator is a binary classifier to classify samples into real or fake. However, different from the networks we have used so far, we'll use convolutional layers and batch normalizations. The high-resolution color images in this project have too many parameters, and if we use dense layers only, it's difficult to train the model effectively.

The structure of the discriminator neural network is as follows:

Listing 4.4 A discriminator in DCGAN

```
import torch.nn as nn
import torch

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

D = nn.Sequential(
    nn.Conv2d(3, 64, 4, 2, 1, bias=False),    #A
    nn.LeakyReLU(0.2, inplace=True),    #B
    nn.Conv2d(64, 128, 4, 2, 1, bias=False),
    nn.BatchNorm2d(128),    #C
    nn.LeakyReLU(0.2, inplace=True),
    nn.Conv2d(128, 256, 4, 2, 1, bias=False),
    nn.BatchNorm2d(256),
    nn.LeakyReLU(0.2, inplace=True),
    nn.Conv2d(256, 512, 4, 2, 1, bias=False),
    nn.BatchNorm2d(512),
    nn.LeakyReLU(0.2, inplace=True),
    nn.Conv2d(512, 1, 4, 1, 0, bias=False),
    nn.Sigmoid(),
    nn.Flatten()).to(device)    #D
```

The input to the discriminator network is a color image with a shape of 3 by 64 by 64. Each of the 2D convolutional layers in the network takes an image and applies filters on it to extract spatial features. Starting from the second 2D convolutional layer, we apply 2D batch normalization (which we have explained in the last section) and LeakyReLU activation (which I'll explain below) on the output. The LeakyReLU activation function is a modified version of ReLU. It allows the output to have a slope for values below zero. Specifically, the LeakyReLU function is defined as follows:

$$\text{LeakyReLU(x)} = \begin{cases} x, & for\ x > 0 \\ -\beta x, & for\ x \leq 0 \end{cases}$$

Where β is a constant between 0 and 1. The LeakyReLU activation function is commonly used to address the sparse gradients problem (when most gradients become zero or near-zero). Training DCGANs is one such case.

4.5.2 Creating A Generator in DCGAN

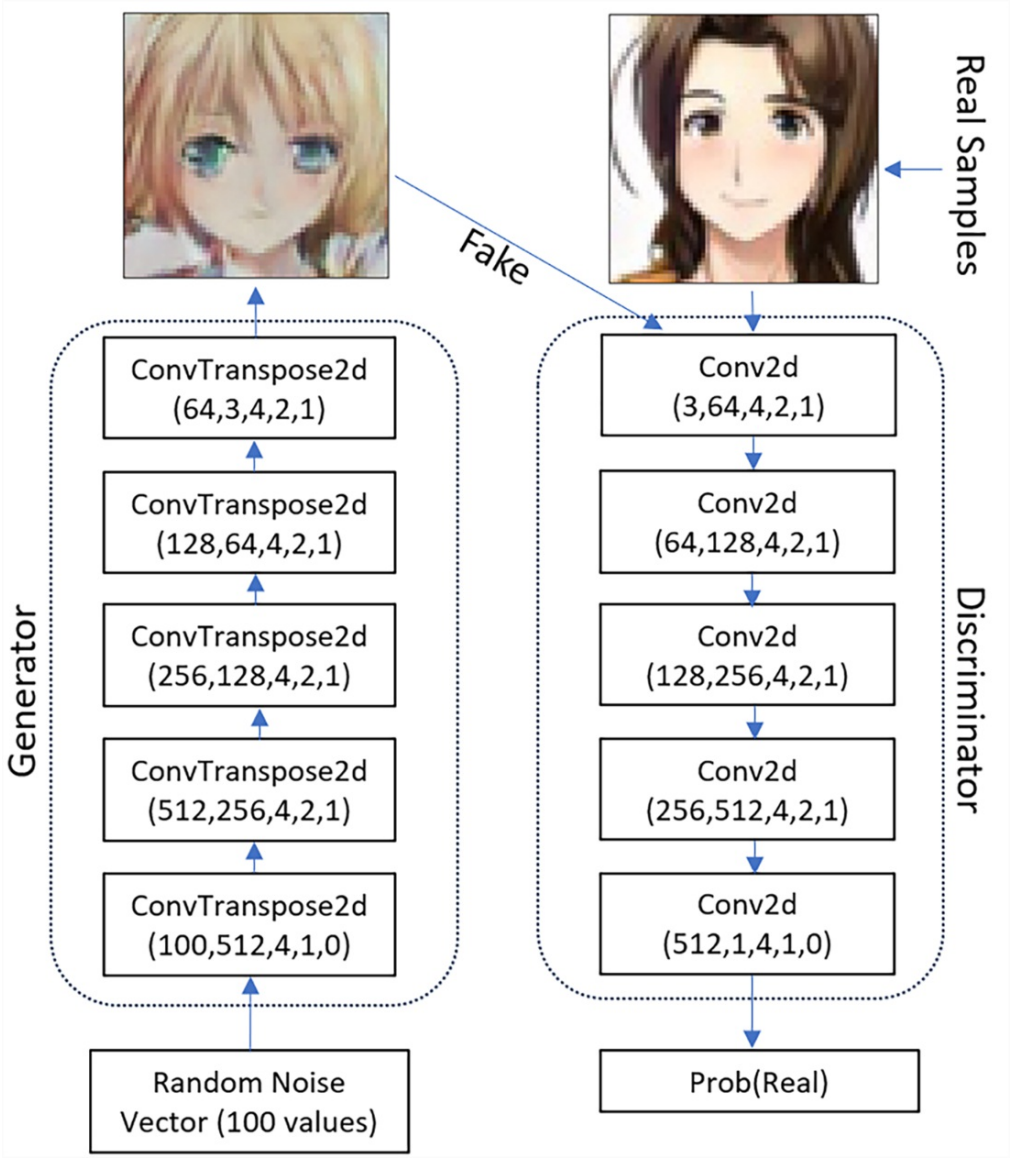
The generator's job is to create an image so that it can pass as real in front of the discriminator. That is, the generator is trying to create an image to maximize the probability that D thinks the image is from the training dataset.

We'll use the same approach when building the generator for clothing item generation. We'll mirror the layers used in the discriminator in DCGAN to create a generator, like so:

Listing 4.5 Designing a generator in DCGAN

```
G=nn.Sequential(
    nn.ConvTranspose2d(100, 512, 4, 1, 0, bias=False),    #A
    nn.BatchNorm2d(512),
    nn.ReLU(inplace=True),
    nn.ConvTranspose2d(512, 256, 4, 2, 1, bias=False),    #B
    nn.BatchNorm2d(256),
    nn.ReLU(inplace=True),
    nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False),
    nn.BatchNorm2d(128),
    nn.ReLU(inplace=True),
    nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),
    nn.BatchNorm2d(64),
    nn.ReLU(inplace=True),
    nn.ConvTranspose2d(64, 3, 4, 2, 1, bias=False),    #C
    nn.Tanh()).to(device)    #D
```

Figure 4.8 Designing a generator network in DCGAN to create Anime faces by mirroring the layers in the discriminator network. The right side of the diagram shows the discriminator network, which contains five 2D convolutional layers. To design a generator that can conjure up Anime faces out of thin air, we mirror the layers in the discriminator network. Specifically, as shown on the left half of the figure, the generator has five 2D transposed convolutional layers in it, symmetric to the 2D convolutional layers in the discriminator. Further, in each of the top four layers, the numbers of *input and output* channels in the discriminator are reversed and used as the numbers of *output and input* channels in the generator.



As shown in figure 4.8, to create an image, the generator uses five 2D transposed convolutional layers: they are symmetric to the five 2D convolutional layers in the discriminator. For example, the last layer, `ConvTranspose2d(64, 3, 4, 2, 1, bias=False)`, is modeled after the first layer in the discriminator, `Conv2d(3, 64, 4, 2, 1, bias=False)`. The numbers of *input and output* channels in `Conv2d` are reversed and used as the numbers of *output and input* channels in `ConvTranspose2d`.

The number of input channels in the first 2D transposed convolutional layer is 100. This is because the generator obtains a 100-value random noise vector from the latent space (bottom left in figure 4.8) and feeds it to the generator.

The number of output channels in the last 2D transposed convolutional layer in the generator is 3 because the output is an image with three color channels (RGB). We apply the Tanh activation function to the output of the generator to squeeze all values to the range [-1, 1] because the training images all have values between -1 and 1.

As usual, the loss function is binary cross-entropy loss. The discriminator is trying to maximize the accuracy of the binary classification: identify a real sample as real and a fake sample as fake. The generator, on the other hand, is trying to minimize the probability that the fake sample is being identified as fake.

We'll use the Adam optimizer for both the discriminator and the generator and set the learning rate to 0.0002:

```
loss_fn=nn.BCELoss()
lr = 0.0002
optimG = torch.optim.Adam(G.parameters(),
                          lr = lr, betas=(0.5, 0.999))
optimD = torch.optim.Adam(D.parameters(),
                          lr = lr, betas=(0.5, 0.999))
```

You have seen the Adam optimizer in Chapter 2, but with default values of betas. Here we select betas that are different from the default values. The betas in the Adam optimizer play crucial roles in stabilizing and speeding up the convergence of the training process. They do this by controlling how much emphasis is placed on recent versus past gradient information (beta1) and by adapting the learning rate based on the certainty of the gradient information (beta2). These parameters are typically fine-tuned based on the specific characteristics of the problem being solved.

4.6 Train and Use DCGAN

The training process for DCGAN is similar to what we have done for other GAN models such as those in Chapter 3 and earlier in this chapter.

Since we don't know the true distribution of Anime face images, we'll rely on visualization techniques to determine when the training is complete. Specifically, we define a `test_epoch()` function to visualize the Anime faces

created by the generator after each epoch of training:

```
def test_epoch():
    noise=torch.randn(32,100,1,1).\
        to(device=device)    #A
    fake_samples=G(noise).cpu().detach()    #B
    for i in range(32):    #C
        ax = plt.subplot(4, 8, i + 1)
        img=(fake_samples.cpu().detach()[i]/2+0.5).\
            permute(1,2,0)
        plt.imshow(img)
        plt.xticks([])
        plt.yticks([])
    plt.subplots_adjust(hspace=-0.6)
    plt.show()
test_epoch()    #D
```

If you run the above code cell, you'll see 32 images that look like snowflake statics on a TV screen. They don't look like Anime faces at all because we haven't trained the generator yet.

We define three functions, `train_D_on_real()`, `train_D_on_fake()`, and `train_G()`, similar to those we used to train the GANs to generate grayscale images of clothing items earlier in this chapter. Go to the Jupypter notebook for this chapter in the book's GitHub repository and familiarize yourself with the functions. They train the discriminator with real images; they then train the discriminator with fake images; finally, they train the generator.

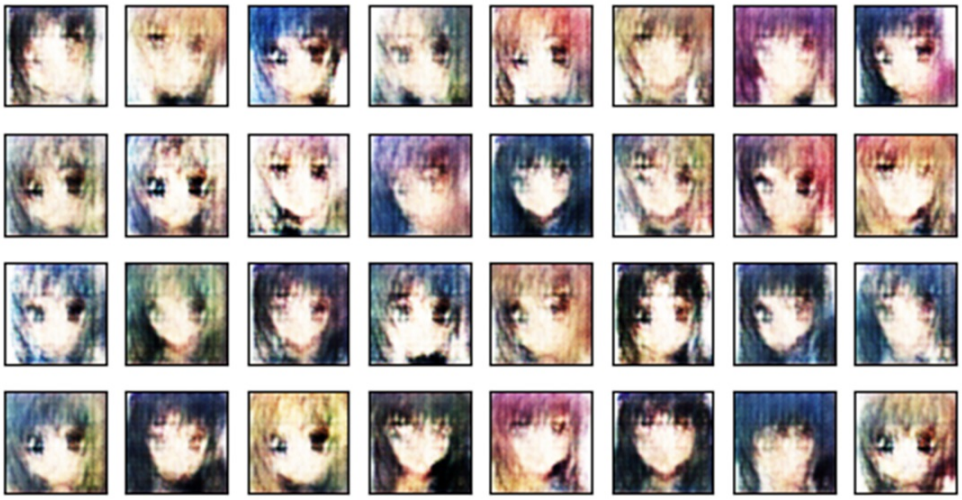
Next, we train the model for 20 epochs, like so:

```
for i in range(20):
    gloss=0
    dloss=0
    for n, (real_samples,_) in enumerate(train_loader):
        loss_D=train_D_on_real(real_samples)
        dloss+=loss_D
        loss_D=train_D_on_fake()
        dloss+=loss_D
        loss_G=train_G()
        gloss+=loss_G
    gloss=gloss/n
    dloss=dloss/n
    print(f"epoch {i+1}, dloss: {dloss}, gloss {gloss}")
    test_epoch()
```

The training takes about 20 minutes if you are using GPU training. Otherwise, it may take two to three hours, depending on the hardware configuration on your computer. Alternatively, you can download the trained model from my website https://gattonweb.uky.edu/faculty/lium/gai/anime_gen.zip.

After every epoch of training, you can visualize the generated Anime faces. After just one epoch of training, the model can already generate color images that look like Anime faces, as shown in figure 4.9. As training progresses, the quality of the generated images becomes better and better.

Figure 4.9 Generated images in DCGAN after one epoch of training.



We'll discard the discriminator and save the trained generator in the local folder, as follows:

```
scripted = torch.jit.script(G)
scripted.save('files/anime_gen.pt')
```

To use the trained generator, we load up the model and use it to generate 32 images:

```
new_G=torch.jit.load('files/anime_gen.pt',
                     map_location=device)
new_G.eval()
noise=torch.randn(32,100,1,1).to(device)
fake_samples=new_G(noise).cpu().detach()
```

```
for i in range(32):
    ax = plt.subplot(4, 8, i + 1)
    img=(fake_samples.cpu().detach()[i]/2+0.5).permute(1,2,0)
    plt.imshow(img)
    plt.xticks([])
    plt.yticks([])
plt.subplots_adjust(hspace=-0.6)
plt.show()
```

The generated Anime faces are shown in figure 4.10,. The generated images bear a close resemblance to the ones in the training set, as shown in figure 4.7.

Figure 4.10 Generated Anime face images by the trained generator in DCGAN.

You may have noticed that the hair colors of the generated images are different: some are black, some are red, and some are blond. You may wonder: can we tell the generator to create images with a certain characteristic, such as black hair or red hair? The answer is yes. You'll learn a couple of different methods to select characteristics in generated images in GANs in Chapter 5.

4.7 Summary

- To conjure up realistic-looking images out of thin air, the generator mirrors layers used in the discriminator network.
- While it's feasible to generate grayscale images by using just fully connected dense layers, to generate high-resolution color images, we

need to use convolutional neural networks (CNNs).

- 2D convolutional layers are used for feature extraction. They apply a set of learnable filters (also known as kernels) to the input data to detect patterns and features at different spatial scales. These layers are essential for capturing hierarchical representations of the input data.
- 2D Transposed Convolutional Layers (also known as Deconvolution or Upsampling Layers) are used for upsampling or generating high-resolution feature maps. They apply a filter to the input data. However, unlike standard convolution, they increase the spatial dimensions by inserting gaps between the output values, which effectively "upscales" the feature maps. This process generates feature maps of a higher resolution.
- 2D Batch Normalization is a technique commonly used in deep learning and neural networks to improve the training and performance of convolutional neural networks (CNNs) and other models that work with 2D data, such as images. It normalizes the values for each feature channel, so they have a mean of 0 and a standard deviation of 1, which helps stabilize and speed up training.

[1] See, for example, Pros and Cons of GAN Evaluation Measures, by Ali Borji, 2018, for a survey on various GAN evaluation methods; https://arxiv.org/abs/1802.03446.

[2] A Note on the Inception Score, by Shane Barratt and Rishi Sharma, 2018, demonstrates that the inception score fails to provide useful guidance when comparing models; https://arxiv.org/abs/1801.01973.

[3] Interested readers can check this GitHub repository on how to implement the inception score in PyTorch to evaluate GANs: https://github.com/sbarratt/inception-score-pytorch. However, the repository doesn’t recommend using the inception score to evaluate generative models due to its ineffectiveness.

5 Selecting Characteristics in Generated Images

This chapter covers

- Building a conditional GAN to generate images with certain attributes (human faces with or without eyeglasses, for example)
- Implementing Wasserstein distance and gradient penalty to improve image quality
- Selecting vectors associated with different features so that the trained GAN model generates images with certain characteristics (male or female faces, for example)
- Combining conditional GAN with vector selection to specify two attributes simultaneously (female faces without glasses or male faces with glasses, for example)

The Anime faces we generated with deep convolutional GAN (DCGAN) in Chapter 4 look realistic. However, you may have noticed that each generated image has different attributes such as hair color, eye color, and whether the head tilts toward left or right. You may be wondering if there is a way to tweak the model so that the generated images have certain characteristics (such as with black hair and tilting toward the left)?

It turns out you can.

In this chapter, you'll learn two different ways of selecting characteristics in the generated images and their respective advantages and disadvantages. The first method involves selecting specific vectors in the latent space. Different vectors correspond to different characteristics - for example, one vector might result in a male face, another in a female face. The second method is the use of a conditional GAN, which involves training the model on labeled data. This allows us to prompt the model to generate images with a specified label, each representing a distinct characteristic - like faces with or without eyeglasses.

In addition, you'll learn to combine the two methods so that you can select two independent attributes of the images at the same time. As a result, you can generate four different groups of images: males with glasses, males without glasses, females with glasses, and females without glasses. To make things more interesting, you can use a weighted average of the labels or a weighted average of the input vectors to generate images that transition from one attribute to another. For example, you can generate a series of images so that the eyeglasses gradually fade out on the same person's face (label arithmetic). Or you can generate a series of images so that the male features gradually fade out and a male face changes to a female face (vector arithmetic).

Being able to conduct either vector arithmetic or label arithmetic alone feels like science fiction, let alone performing the two simultaneously. The whole experience reminds us of the quote by Arthur C. Clarke (author of *2001: A Space Odyssey*), "Technology advanced enough is indistinguishable from magic."

Despite the realism of the Anime faces generated in Chapter 4, they were limited by low resolution. Training GAN models can be tricky, often hampered by issues like small sample sizes or low-quality images. These challenges can prevent models from converging, resulting in poor image quality. To address this, we'll discuss and implement an improved training technique using the Wasserstein distance with gradient penalty in our conditional GAN. This enhancement results in more realistic human faces and noticeably better image quality compared to the previous chapter.

5.1 The Eyeglasses Dataset

We'll use the eyeglasses dataset in this chapter to train a conditional GAN model. Further, in the next chapter, we'll also use this dataset to train a CycleGAN model in one of the exercises: to convert an image with eyeglasses to an image without eyeglasses and vice versa. In this section, you'll learn to download the dataset and preprocess images in it.

The Python programs in this chapter and the next are adapted from two excellent online open-source projects: the Kaggle project by Yashika Jain

https://www.kaggle.com/code/yashikajain/eye-glass-removal and a GitHub repository by Aladdin Persson https://github.com/aladdinpersson/Machine-Learning-Collection/blob/master/ML/Pytorch/GANs/CycleGAN. I encourage you to look into these two projects while going through this chapter and the next.

5.1.1 Download the eyeglasses dataset

The eyeglasses dataset we use is from Kaggle. Log into Kaggle and go to the link https://www.kaggle.com/datasets/jeffheaton/glasses-or-no-glasses/data to download the image folder and the two CSV files on the right: `train.csv` and `test.csv`. There are 5000 images in the folder /faces-spring-2020/. Once you have the data, place both the image folder and the two CSV files inside the folder /files/ on your computer.

Next, we'll sort the photos into two subfolders: one containing only images with eyeglasses and another one with images without eyeglasses.

First, let's look at the file train.csv, like so:

```
import pandas as pd

train=pd.read_csv('files/train.csv')    #A
train.set_index('id', inplace=True)    #B
```

The above code cell imports the file `train.csv` and sets the variable `id` as the index of each observation. The column `glasses` in the file has two values: 0 or 1, indicating whether the image has eyeglasses in it or not.

Below, we separate the images into two different folders: one containing images with eyeglasses and one containing images without eyeglasses.

Listing 5.1 Sorting images with and without eyeglasses into two separate folders

```
import os, shutil

G='files/glasses/G/'
NoG='files/glasses/NoG/'
os.makedirs(G, exist_ok=True)    #A
os.makedirs(NoG, exist_ok=True)    #B
```

```
folder='files/faces-spring-2020/faces-spring-2020/'
for i in range(1,4501):
    if train.loc[i]['glasses']==0:    #C
        oldpath=f"{folder}face-{i}.png"
        newpath=f"{NoG}face-{i}.png"
        shutil.move(oldpath, newpath)
    if train.loc[i]['glasses']==1:    #D
        oldpath=f"{folder}face-{i}.png"
        newpath=f"{G}face-{i}.png"
        shutil.move(oldpath, newpath)
```

In the code cell above, we first use the `os` library to create two subfolders /glasses/G/ and /glasses/NoG/ inside the folder /files/ on your computer. We then use the `shutil` library to move images to the two folders based on the label `glasses` in the file `train.csv`. Those labeled 1 are moved to folder G and those with label 0 to folder NoG.

5.1.2 Visualize images in the eyeglasses dataset

The classification column `glasses` in the file `train.csv` is not perfect. If you go to the subfolder G on your computer, for example, you'll see that most images have glasses, but about 10% of them have no glasses. Similarly, if you go to the subfolder NoG, you'll see that about 10% of them actually have glasses. You need to manually correct this by moving images from one folder to the other. This is important for our training later so you should do the same: manually move images in the two folders so that one contains only images with glasses and the other images without glasses. Welcome to the life of a data scientist: fixing data problems is part of daily routine!

Let’s first visualize some examples of images with eyeglasses:

Listing 5.2 Visualizing images with eyeglasses

```
import random
import matplotlib.pyplot as plt
from PIL import Image

imgs=os.listdir(G)
random.seed(42)
samples=random.sample(imgs,16)    #A
fig=plt.figure(dpi=200, figsize=(8,2))
```

```
for i in range(16):    #B
    ax = plt.subplot(2, 8, i + 1)
    img=Image.open(f"{G}{samples[i]}")
    plt.imshow(img)
    plt.xticks([])
    plt.yticks([])
plt.subplots_adjust(wspace=-0.01,hspace=-0.01)
plt.show()
```

If you have manually corrected the mislabeling of images in folder G, you'll see 16 images with eyeglasses after running the code in listing 5.2. The output is in figure 5.1.

Figure 5.1 Sample images with eyeglasses in the training dataset.

You can change G to NoG in listing 5.2 to visualize 16 sample images without eyeglasses in the dataset. The complete code is in the book's GitHub repository. The output is shown in figure 5.2.

Figure 5.2 Sample images without eyeglasses in the training dataset.

5.2 Conditional GAN with Wasserstein Distance

A conditional GAN is similar to the GAN models you have seen in chapters 3 and 4, with the exception that you attach a label to the input data. The labels

correspond to different characteristics in the input data. Once the trained GAN model “learns” to associate a certain label with a characteristic, you can feed a random noise vector with a label to the model to generate output with the desired characteristic.

GAN models often suffer from problems like mode collapse (the generator finds a certain type of output that is good at fooling the discriminator and then collapses its outputs to these few modes, ignoring other variations), vanishing gradients, and unstable training. Wasserstein GAN introduces the Earth Mover's (or Wasserstein-1) distance as the loss function, offering a smoother gradient flow and more stable training. It mitigates issues like mode collapse. We’ll implement it in conditional GAN training in this chapter. Note that Wasserstein GAN is a concept independent of Conditional GAN: it uses the Wasserstein distance to improve the training process and can be applied to any GAN model (such as the ones we created in Chapters 3 and 4). We'll combine both concepts in one setting to save space.

Other ways to stabilize GAN training

The problems with training GAN models are most common when generating high-resolution images. The model architecture is usually complex with many neural layers. Other than Wasserstein GAN, progressive GAN is another way to stabilize training. Progressive GANs enhance the stability of GAN training by breaking down the complex task of high-resolution image generation into manageable steps, allowing for more controlled and effective learning. For details, see Progressive Growing of GANs for Improved Quality, Stability, and Variation, by Karas et al: https://arxiv.org/abs/1710.10196.

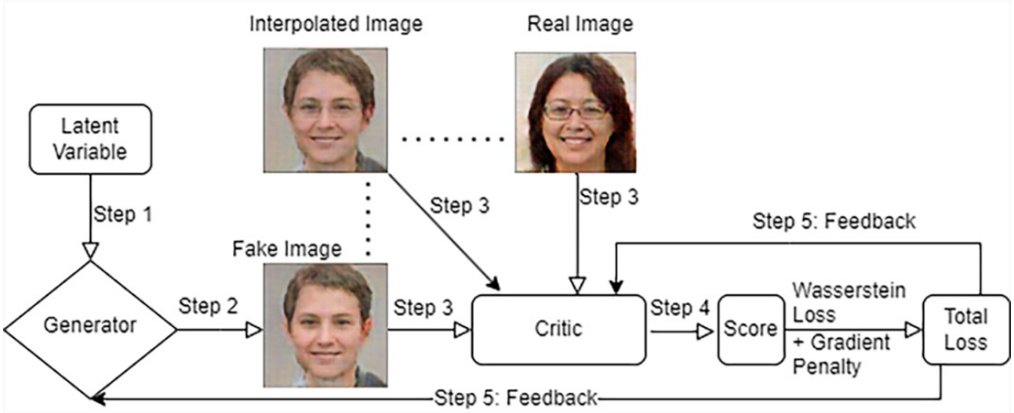
5.2.1 Wasserstein GAN with Gradient Penalty

Wasserstein GAN (WGAN) is a technique used to improve the training stability and performance of GAN models. Regular GANs (such as the ones you have seen in Chapters 3 and 4) have two components – a generator and a discriminator. The generator creates fake data, while the discriminator evaluates whether the data is real or fake. Training involves a competitive zero-sum game in which the generator tries to fool the discriminator and the

discriminator tries to accurately classify real and fake data instances.

Researchers have proposed to use Wasserstein distance (a measure of dissimilarity between two distributions) instead of the binary cross-entropy as the loss function to stabilize training, with a gradient penalty term.[1] The technique offers a smoother gradient flow and mitigates issues like mode collapse. Figure 5.3 provides a diagram of Wasserstein GAN (WGAN). As you can see on the right side of the figure, the losses associated with the real and fake images are Wasserstein loss instead of the regular binary cross-entropy loss.

Figure 5.3 A diagram of Wasserstein GAN with gradient penalty. The discriminator network in Wasserstein GAN (which we call the critic) rates input images: it tries to assign a score of -∞ to a fake image (bottom left) and a score of ∞ to the real image (top middle). Further, an interpolated image of the real and fake images (top left) is presented to the critic, and the gradient penalty with respect to the critic's rating on the interpolated image is added to the total loss in the training process.



Further, for the Wasserstein distance to work correctly, the discriminator (called the critic in WGANs) must be 1-Lipschitz continuous, meaning the gradient norms of the critic's function must be at most 1 everywhere. The original WGAN paper proposed weight clipping to enforce the Lipschitz constraint. However, this approach had its problems, like capacity underuse and exploding or vanishing gradients.

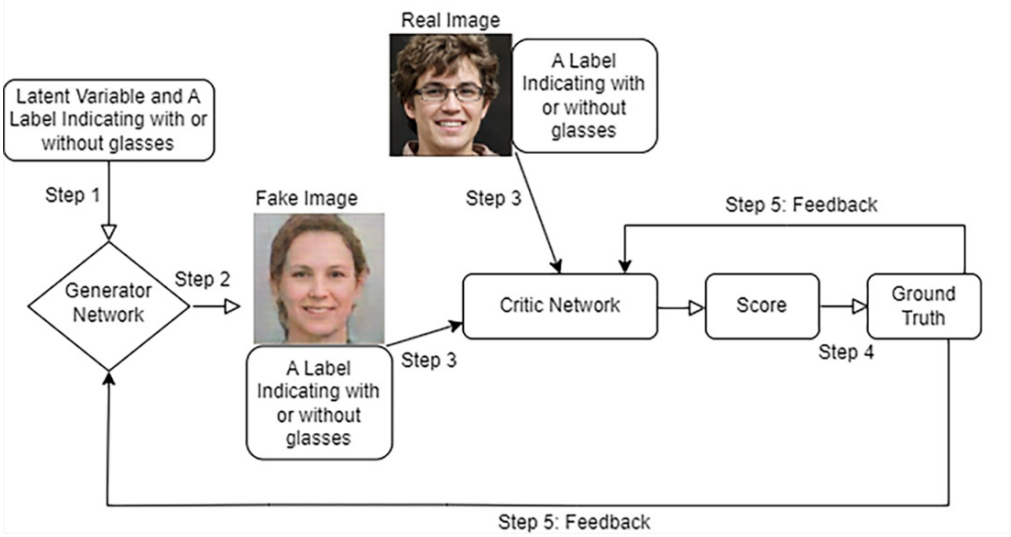
To address weight clipping issues, the gradient penalty is added to the loss function to enforce the Lipschitz constraint more effectively. To implement WGAN with gradient penalty, we first randomly sample points along the straight line between real and generated data points (as indicated by the

interpolated image in the top left of figure 5.3). We then compute the gradient of the critic's output with respect to these sampled points. Finally, we add a penalty to the loss function proportional to the deviation of these gradient norms from 1 (the penalty term is called gradient penalty). That is, gradient penalty in WGANs is a technique to improve training stability and sample quality by enforcing the Lipschitz constraint more effectively, addressing the limitations of the original WGAN model.

5.2.2 Conditional GANs

Conditional GAN (or cGAN) is an extension of the basic GAN framework. In a Conditional GAN, both the generator and the discriminator (or the critic since we are implanting WGAN and cGAN in the same setting) are conditioned on some additional information. This could be anything such as class labels, data from other modalities, or even textual descriptions. This conditioning is typically achieved by feeding this additional information into both the Generator and Discriminator. In our setting, we'll add class labels to the inputs to both the generator and the critic: we attach one label to images with eyeglasses and another label to images without eyeglasses. Figure 5.4 provides a diagram of the training process for cGANs.

Figure 5.4 A diagram of the training process for conditional GANs.



As you see at the top left of figure 5.4, in a cGAN, the generator receives

both a random noise vector and the conditional information (a label indicating whether the image has eyeglasses or not) as input. It uses this information to generate data that not only looks real but also aligns with the conditional input.

The critic receives either real data from the training set or fake data generated by the generator, along with the conditional information (a label indicating whether the image has eyeglasses or not in our setting). Its task is to determine whether the given data is real or fake, taking the conditional information into account (does the generated image have eyeglasses in it?). In figure 5.4, we use the critic network instead of the discriminator network since we implement both cGAN and WGAN simultaneously, but the concept of cGAN applies to traditional GANs as well.

The main advantage of cGANs is their ability to select aspects of the generated data, making them more versatile and applicable in scenarios where the output needs to be directed or conditioned on certain input parameters. In our setting, we'll train the cGAN so that we have the ability to select whether the generated images have eyeglasses or not.

In summary, Conditional GANs are a powerful extension of the basic GAN architecture, enabling targeted generation of synthetic data based on conditional inputs.

5.3 Create a conditional GAN

In this section, you'll learn to create a conditional GAN to generate human faces with or without eyeglasses. You'll also learn to implement the Wasserstein GAN with gradient penalty to stabilize training.

The generator in cGANs uses not only random noise vectors, but also conditional information such as labels as inputs, to create images either with or without eyeglasses. Further, a critic network in WGANs is different from the discriminator network in traditional GANs. You'll also learn how to calculate the Wasserstein distance and the gradient penalty in this section.

5.3.1 A critic in conditional GAN

In traditional GANs, the discriminator is a binary classifier to identify the input as either real or fake, conditional on the label. In Wasserstein GAN, we call the discriminator network the critic. The critic evaluates the input and gives a score between $-\infty$ and ∞. The higher the score, the more likely that the input is from the training set (that is, real).

Listing 5.3 creates the critic network. The architecture is somewhat similar to the discriminator network we used in Chapter 4 when generating color images of Anime faces. In particular, we use seven `Conv2d` layers in PyTorch to gradually downsample the input so that the output is a single value between $-\infty$ and ∞, as follows:

Listing 5.3 A critic network in conditional GAN with Wasserstein distance

```
class Critic(nn.Module):
    def __init__(self, img_channels, features):
        super().__init__()
        self.net = nn.Sequential(    #A
            nn.Conv2d(img_channels, features,
                      kernel_size=4, stride=2, padding=1),
            nn.LeakyReLU(0.2),
            self.block(features, features * 2, 4, 2, 1),
            self.block(features * 2, features * 4, 4, 2, 1),
            self.block(features * 4, features * 8, 4, 2, 1),
            self.block(features * 8, features * 16, 4, 2, 1),
            self.block(features * 16, features * 32, 4, 2, 1),
            nn.Conv2d(features * 32, 1, kernel_size=4,
                      stride=2, padding=0))    #B
    def block(self, in_channels, out_channels,
              kernel_size, stride, padding):
        return nn.Sequential(    #C
            nn.Conv2d(in_channels,out_channels,
                kernel_size,stride,padding,bias=False,),
            nn.InstanceNorm2d(out_channels, affine=True),
            nn.LeakyReLU(0.2))
    def forward(self, x):
        return self.net(x)
```

The input to the critic network is a color image with a shape of 5 by 256 by 256. The first three channels are the color channels (colors red, green, and blue; RGB). The last two channels are labeling channels to tell the critic whether the image is with glasses or without glasses. We'll discuss the exact

mechanism to accomplish this in the next section.

The critic network consists of seven `Conv2d` layers. If you recall, in Chapter 4, we have discussed in depth how these layers work. They are used for feature extraction by applying a set of learnable filters on the input images to detect patterns and features at different spatial scales, effectively capturing hierarchical representations of the input data. The critic then evaluates the input images based on these representations. The five `Conv2d` layers in the middle are all followed by an `InstanceNorm2d` layer and a `LeakyReLU` activation, we hence define a `block()` method in the critic network to simplify the critic network architecture. The `InstanceNorm2d` layer is similar to the `BatchNorm2d` layer we discussed in Chapter 4, except that we normalize each individual instance in the batch independently.

Another key point to keep in mind is that the output is no longer a value between 0 and 1 since we don't use the sigmoid activation in the last layer in the critic network. Instead, the output is a value between $-\infty$ and ∞.

5.3.2 A generator in conditional GAN

In Wasserstein GANs, the generator's job is to create data instances so that they can be evaluated at a high score by the critic. In conditional GANs, the generator must generate data instances with conditional information (with or without eyeglasses in our setting). Since we are implementing a conditional GAN with Wasserstein distance, we'll tell the generator what type of images we want it to generate by attaching a label to the random noise vector. We'll discuss the exact mechanism in the next section.

We create the following neural network to represent the generator:

Listing 5.4 A generator in conditional GAN

```
class Generator(nn.Module):
    def __init__(self, noise_channels, img_channels, features):
        super(Generator, self).__init__()
        self.net = nn.Sequential(    #A
            self.block(noise_channels, features *64, 4, 1, 0),
            self.block(features * 64, features * 32, 4, 2, 1),
            self.block(features * 32, features * 16, 4, 2, 1),
```

```
            self.block(features * 16, features * 8, 4, 2, 1),
            self.block(features * 8, features * 4, 4, 2, 1),
            self.block(features * 4, features * 2, 4, 2, 1),
            nn.ConvTranspose2d(
                features * 2, img_channels, kernel_size=4,
                stride=2, padding=1),
            nn.Tanh())    #B
    def block(self, in_channels, out_channels,
              kernel_size, stride, padding):
        return nn.Sequential(    #C
            nn.ConvTranspose2d(in_channels,out_channels,
                kernel_size,stride,padding,bias=False,),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),)
    def forward(self, x):
        return self.net(x)
```

We'll feed a random noise vector from a 100-dimensional latent space to the generator as input. We'll also feed a two-dimensional one-hot encoded image label to the generator to tell it to generate an image either with or without eyeglasses. For simplicity, we'll concatenate the two pieces of information together to form a 102-dimensional input variable to the generator. The generator then generates a color image based on the input from the latent space and the labeling information.

The generator network consists of seven `ConvTranspose2d` layers, and the idea is to mirror the steps in the critic network to conjure up images, as we discussed in Chapter 4. The first six `ConvTranspose2d` layers are all followed by a `BatchNorm2d` layer and a `ReLU` activation, we hence define a `block()` method in the generator network to simplify the architecture. As we have done in Chapter 4, we use the Tanh activation function at the output layer so the output pixels are all in the range of -1 and 1, the same as the images in the training set.

5.3.3 Weight initialization and the gradient penalty function

In deep learning, the weights in neural networks are randomly initialized. When the network architecture is complicated and there are many hidden layers in it (which is the case in our setting), how weights are initialized is crucial.

We, therefore, define the following `weights_init()` function to initialize weights in both the generator and the critic networks:

```
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

The function initializes weights in `Conv2d` and `ConvTranspose2d` layers with values drawn from a normal distribution with a mean of 0 and a standard deviation of 0.02. It also initializes weights in `BatchNorm2d` layers with values drawn from a normal distribution with a mean of 1 and a standard deviation of 0.02. We choose a small standard deviation in weight initializations to avoid vanishing gradients.

Next, we create a generator and a critic based on the `Generator()` and `Critic()` classes we defined in the last subsection. We then initialize the weights in them based on the `weights_init()` function defined above, like so:

```
z_dim=100
img_channels=3
features=16
gen=Generator(z_dim+2,img_channels,features).to(device)
critic=Critic(img_channels+2,features).to(device)
weights_init(gen)
weights_init(critic)
```

As usual, we'll use the Adam optimizer for both the critic and the generator:

```
lr = 0.0001
opt_gen = torch.optim.Adam(gen.parameters(),
                           lr = lr, betas=(0.0, 0.9))
opt_critic = torch.optim.Adam(critic.parameters(),
                           lr = lr, betas=(0.0, 0.9))
```

The generator tries to create images that are indistinguishable from those in the training set, with the given label. It presents the images to the critic to obtain high ratings on the generated images. The critic, on the other hand,

tries to assign high ratings to real images and low ratings to fake images, conditional on the given label. Specifically, the loss function for the critic has three components in it:

$$critic\_value(fake) - critic\_value(real) + weight * GradientPenalty$$

The first term, $critic\_value(fake)$, says that if an image is fake, the critic's objective is to identify it as fake and give it a low evaluation. The second term, $-critic\_value(real)$, indicates that if the image is real, the critic's objective is to identify it as real and give it a high evaluation. Further, the critic wants to minimize the gradient penalty term, $weight * GradientPenalty$, where $weight$ is a constant to determine how much penalty we want to assign to deviations of the gradient norms from the value 1. The gradient penalty is calculated as follows:

Listing 5.5 How to calculate gradient penalty

```
def GP(critic, real, fake):
    B, C, H, W = real.shape
    alpha=torch.rand((B,1,1,1)).repeat(1,C,H,W).to(device)
    interpolated_images = real*alpha+fake*(1-alpha)    #A
    critic_scores = critic(interpolated_images)    #B
    gradient = torch.autograd.grad(
        inputs=interpolated_images,
        outputs=critic_scores,
        grad_outputs=torch.ones_like(critic_scores),
        create_graph=True,
        retain_graph=True)[0]    #C
    gradient = gradient.view(gradient.shape[0], -1)
    gradient_norm = gradient.norm(2, dim=1)
    gp = torch.mean((gradient_norm - 1) ** 2)    #D
    return gp
```

In the function `GP()`, we first create interpolated images of real ones and fake ones. This is done by randomly sampling points along the straight line between real and generated images. Imagine a slider: at one end is the real image, and at the other is the fake image. As you move the slider, you see a continuous blend from the real to the fake, with the interpolated images representing the stages in between.

We then present interpolated images to the critic network to obtain ratings on

them and calculate the gradient of the critic's output with respect to the interpolated images. Finally, the gradient penalty is calculated as the squared deviation of the gradient norms from the target value of 1.

5.4 Train the conditional GAN

As we mentioned in the last section, we need to find a way to tell both the critic and the generator what the image label is, so they know if the image has eyeglasses in it or not.

In this section, you'll first learn how to add labels to the inputs to the critic network and the inputs to the generator network so the generator knows what type of images to create while the critic can evaluate the images conditional on the labels. After that, you'll learn how to train the conditional GAN with Wasserstein distance.

5.4.1 Add labels to inputs

We first preprocess the data and convert the images to torch tensors, as follows:

```
import torchvision.transforms as T
import torchvision

batch_size=16
imgsz=256
transform=T.Compose([
    T.Resize((imgsz,imgsz)),
    T.ToTensor(),
    T.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])])
data_set=torchvision.datasets.ImageFolder(
    root=r"files/glasses",
    transform=transform)
```

We set the batch size to 16 and the image size to 256 by 256 pixels. The pixel values are chosen so the generated images have higher resolutions than those in the last chapter (64 by 64 pixels). We choose a batch size of 16, smaller than the batch size in Chapter 3, due to the larger image size. If the batch size is too large, your GPU (or even CPU) will run out of memory.

tip

If you are using GPU-training and your GPU has limited memory, consider reducing the batch size to a smaller number than 16, such as 10 or 8, so that your GPU doesn't run out of memory. Alternatively, you can keep the batch size at 16, but switch to CPU training to address the GPU memory problem.

Next, we'll add labels to the training data. Since there are two types of images: images with eyeglasses and images without glasses, we'll create two one-hot image labels. Images with glasses will have a one-hot label of [1, 0], and images without glasses will have a one-hot label of [0, 1].

The input to the generator is a 100-value random noise vector. We concatenate the one-hot label with the random noise vector and feed the 102-value input to the generator. The input to the critic network is a three-channel color image with a shape of 3 by 256 by 256 (PyTorch uses channel-first tensors to represent images). How to attach a label with a shape of 1 by 2 to an image with a shape of 3 by 256 by 256? The solution is to add two channels to the input image so that the image shape changes from (3, 256, 256) to (5, 256, 256): the two additional channels are the one-hot labels. Specifically, if an image has eyeglasses in it, the fourth channel is filled with 1s and the fifth channel 0s; if the image has no eyeglasses in it, the fourth channel is filled with 0s and the fifth channel 1s.

How to create labels if there are more than two values in a characteristic?

You can easily extend the conditional GAN model to characteristics with more than two values. For example, if you create a model to generate images with different hair colors: black, blond, and white, the image label you feed to the generator can have values [1, 0, 0], [0, 1, 0], and [0, 0, 1], respectively. You can attach three channels to the input image before you feed it to the discriminator or critic. For example, if an image has black hair, the fourth channel is filled with 1s and the fifth and sixth channels 0s.

Additionally, in this particular example, since there are only two values in the label, you can potentially use values 0 and 1 to indicate images with and without glasses when you feed the label to the generator. You can attach one channel to the input image before you feed it to the critic: if an image has

eyeglasses in it, the fourth channel is filled with 1s; if the image has no eyeglasses in it, the fourth channel is filled with 0s. I'll leave that as an exercise for you. The solution is provided in the book's GitHub repository.

We implement the above change as follows:

Listing 5.6 Attaching labels to input images

```
newdata=[]
for i,(img,label) in enumerate(data_set):
    onehot=torch.zeros((2))
    onehot[label]=1
    channels=torch.zeros((2,imgsz,imgsz))    #A
    if label==0:
        channels[0,:,:]=1    #B
    else:
        channels[1,:,:]=1    #C
    img_and_label=torch.cat([img,channels],dim=0)    #D
    newdata.append((img,label,onehot,img_and_label))
```

tip

Earlier when we load the images by using the torchvision.datasets.ImageFolder() method from the folder /files/glasses, PyTorch assigns labels to images in each subfolder in alphabetical order. Therefore, images in /files/glasses/G/ are assigned a label of 0 and those in /files/glasses/NoG/ a label of 1.

We first create an empty list `newdata` to hold images with labels. We create a PyTorch tensor with a shape (2, 256, 256), to be attached to the original input image to form a new image with a shape of (5, 256, 256). If the original image label is 0 (this means images are from the folder /files/glasses/G/), we fill the fourth channel with 1s and the fifth channel with 0s so that the critic knows it's an image with glasses. On the other hand, if the original image label is 1 (this means images are from the folder /files/glasses/NoG/), we fill the fourth channel with 0s and the fifth channel with 1s so that the critic knows it's an image without glasses.

tip

If you are using GPU-training and your GPU has limited memory, consider reducing the batch size to a smaller number than 16, such as 10 or 8, so that your GPU doesn’t run out of memory. Alternatively, you can keep the batch size at 16, but switch to CPU training to address the GPU memory problem.

We create a data iterator with batches (to improve computational efficiency, memory usage, and optimization dynamics in the training process) as follows:

```
data_loader=torch.utils.data.DataLoader(
    newdata,batch_size=batch_size,shuffle=True)
```

5.4.2 Train the Conditional GAN

Now that we have the training data and two networks, we'll train the conditional GAN. We’ll use visual inspections to determine when the training should stop.

Once the model is trained, we'll discard the critic network and use the generator to create images with a certain characteristic (with or without glasses in our case).

We'll create a function to test periodically what the generated images look like:

Listing 5.7 Inspect generated images

```
def test_epoch(epoch):
    noise = torch.randn(32, z_dim, 1, 1)
    labels = torch.zeros(32, 2, 1, 1)
    labels[:,0,:,:]=1    #A
    noise_and_labels=torch.cat([noise,labels],dim=1).to(device)
    fake=gen(noise_and_labels).cpu().detach()    #B
    fig=plt.figure(figsize=(20,10),dpi=100)
    for i in range(32):    #C
        ax = plt.subplot(4, 8, i + 1)
        img=(fake.cpu().detach()[i]/2+0.5).permute(1,2,0)
        plt.imshow(img)
        plt.xticks([])
        plt.yticks([])
    plt.subplots_adjust(hspace=-0.6)
```

```
    plt.savefig(f"files/glasses/G{epoch}.png")
    plt.show()
    noise = torch.randn(32, z_dim, 1, 1)
    labels = torch.zeros(32, 2, 1, 1)
    labels[:,1,:,:]=1    #D
    … (code omitted)
```

After each epoch of training, we'll ask the generator to create a set of images with glasses and a set of images without glasses. We then plot the images so that we can inspect them visually. To create images with glasses, we first create one-hot labels [1, 0] and attach them to the random noise vectors before feeding the concatenated vector to the generator network. The generator creates images with glasses since the label is [1, 0] instead of [0, 1]. We then plot the generated images in four rows and eight columns and save the subplots on your computer. The process of creating images without glasses is similar except that we use the one-hot label [0, 1] instead of [1, 0]. I skipped part of the code in listing 5.7 but you can find it in the book's GitHub repository https://github.com/markhliu/DGAI.

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

Listing 5.8 Train the model with a batch of data

```
def train_batch(onehots,img_and_labels,epoch):
    real = img_and_labels.to(device)    #A
    B = real.shape[0]
    for _ in range(5):
        noise = torch.randn(B, z_dim, 1, 1)
        onehots=onehots.reshape(B,2,1,1)
        noise_and_labels=torch.cat([noise,onehots],dim=1).to(devi
        fake_img = gen(noise_and_labels).to(device)
        fakelabels=img_and_labels[:,3:,:,:].to(device)
        fake=torch.cat([fake_img,fakelabels],dim=1).to(device)
        critic_real = critic(real).reshape(-1)
        critic_fake = critic(fake).reshape(-1)
        gp = GP(critic, real, fake)
        loss_critic=(-(torch.mean(critic_real) -
           torch.mean(critic_fake)) + 10 * gp)    #C
        critic.zero_grad()
        loss_critic.backward(retain_graph=True)
        opt_critic.step()
    gen_fake = critic(fake).reshape(-1)
```

```
    loss_gen = -torch.mean(gen_fake)    #D
    gen.zero_grad()
    loss_gen.backward()
    opt_gen.step()
    return loss_critic, loss_gen
```

In the `train_batch()` function, we train the critic and the generator with a batch of data from the training set. To train the critic, we ask the generator to create a batch of data with the given label and train the critic network five times with the real and fake images. In the `train_batch()` function, we also train the generator with a batch of fake data.

NOTE

The loss for the critic has three components: loss from evaluating real images, loss from evaluating fake images, and the gradient penalty loss.

We now train the model for 100 epochs, as follows:

```
for epoch in range(1,101):
    closs=0
    gloss=0
    for _,_,onehots,img_and_labels in data_loader:    #A
        loss_critic, loss_gen = train_batch(onehots,\
                                img_and_labels,epoch)    #B
        closs+=loss_critic
        gloss+=loss_gen
    print(f"at epoch {epoch},\
    critic loss: {closs}, generator loss {gloss}")
    test_epoch(epoch)
torch.save(gen.state_dict(),'files/cgan.pth')    #C
```

After each epoch of training, we print out the critic loss and the generator loss to ensure that the losses are in a reasonable range. We also generate 32 images of faces with glasses as well as 32 images without glasses, by using the `test_epoch()` function we defined earlier. We also save the weights in the trained generator in the local folder after training is done so that later we can generate images using the trained model.

The above training takes about 30 minutes if you are using GPU training. Otherwise, it may take several hours, depending on the hardware

configuration on your computer. Alternatively, you can download the trained model from my website https://gattonweb.uky.edu/faculty/lium/gai/cgan.zip.

5.5 Select characteristics in generated images

There are at least two ways to generate images with a certain characteristic. The first is to attach a label to a random noise vector before feeding it to the trained conditional GAN model. Different labels lead to different characteristics in the generated image (in our case, whether the image has eyeglasses in it or not). The second way is to select the noise vector you feed to the trained model: while one vector leads to an image with a male face, another leads to an image with a female face. Note that the second way works even in a traditional GAN such as the ones we trained in Chapter 4. It works in a conditional GAN as well.

Better yet, in this section, you'll learn to combine these two methods so you can select two characteristics simultaneously: an image of a male face with eyeglasses, or a female face without eyeglasses, and so on.

There are pros and cons for each one of these two methods to select a certain characteristic in generated images. The first way, the conditional GAN, requires labeled data to train the model. Sometimes, labeled data is costly to curate. However, once you have successfully trained a conditional GAN, you can generate a wide range of images with a certain characteristic. In our case, you can generate many different images with eyeglasses (or without eyeglasses), each one is different from another. The second way, handpicking a noise vector, doesn't need labeled data to train the model. However, each handpicked noise vector can only generate one image. If you want to generate many different images with the same characteristic as the conditional GAN, you'll need to handpick many different noise vectors ex ante.

5.5.1 Select images with or without eyeglasses

By attaching a label of either [1, 0] or [0, 1] to a random noise vector before you feed it to the trained conditional GAN model, you can select whether the generated image has eyeglasses in it or not.

First, we'll use the trained model to generate 32 images with glasses and plot them in a four-by-eight grid. To make results reproducible, we'll fix the random state in PyTorch. Further, we'll use the same set of random noise vectors so that we look at the same set of faces.

We fix the random state at seed 0 and generate 32 images of faces with eyeglasses, like so:

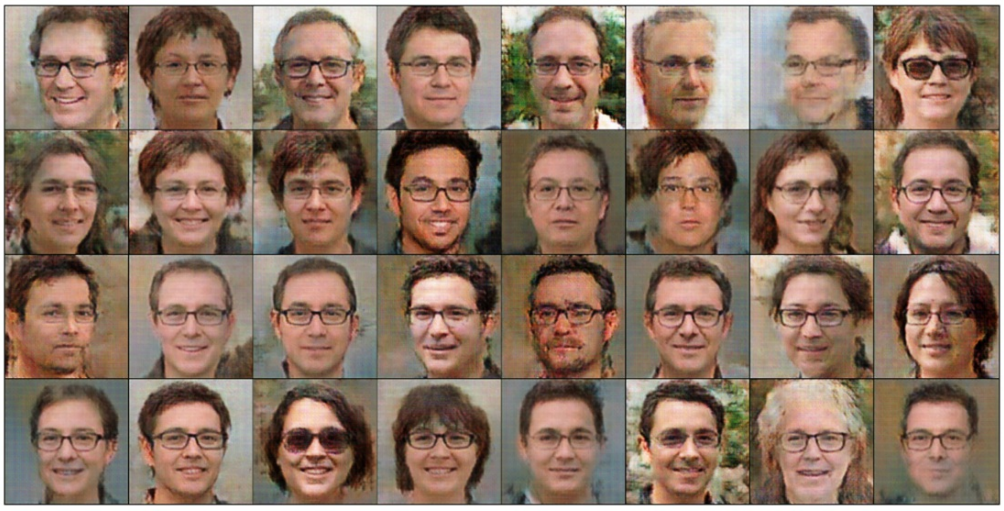
Listing 5.9 Generate images of human faces with eyeglasses

```
torch.manual_seed(0)    #A

generator=Generator(z_dim+2,img_channels,features).to(device)
generator.load_state_dict(torch.load("files/cgan.pth"))    #B
generator.eval()

noise_g=torch.randn(32, z_dim, 1, 1)    #C
labels_g=torch.zeros(32, 2, 1, 1)
labels_g[:,0,:,:]=1    #D
noise_and_labels=torch.cat([noise_g,labels_g],dim=1).to(device)
fake=generator(noise_and_labels)
plt.figure(figsize=(20,10),dpi=50)
for i in range(32):
    ax = plt.subplot(4, 8, i + 1)
    img=(fake.cpu().detach()[i]/2+0.5).permute(1,2,0)
    plt.imshow(img.numpy())
    plt.xticks([])
    plt.yticks([])
plt.subplots_adjust(wspace=-0.08,hspace=-0.01)
plt.show()
```

We create another instance of the `Generator()` class and name it `generator`. We then load up the trained weights that we saved in the local folder in the last section (or you can download the weights in the book's GitHub repository). To generate 32 images of human faces with eyeglasses, we first draw 32 random noise vectors in the latent space. We'll also create a set of labels and name them `labels_g`, and they tell the generator to produce 32 images with eyeglasses.

If you run the program in listing 5.9, you'll see 32 images as shown in figure 5.5.

Figure 5.5 Images of human faces with eyeglasses that are generated by the trained conditional GAN model.



First of all, all 32 images do have eyeglasses in them. This indicates that the trained conditional GAN model is able to generate images conditional on the provided labels. You may have noticed that some images have male features while others have female features. To prepare us for vector arithmetic in the next subsection, we'll select one random noise vector that leads to an image with male features and one that leads to female features. After inspecting the 32 images in figure 5.5, we select images with index values 0 and 14, like so:

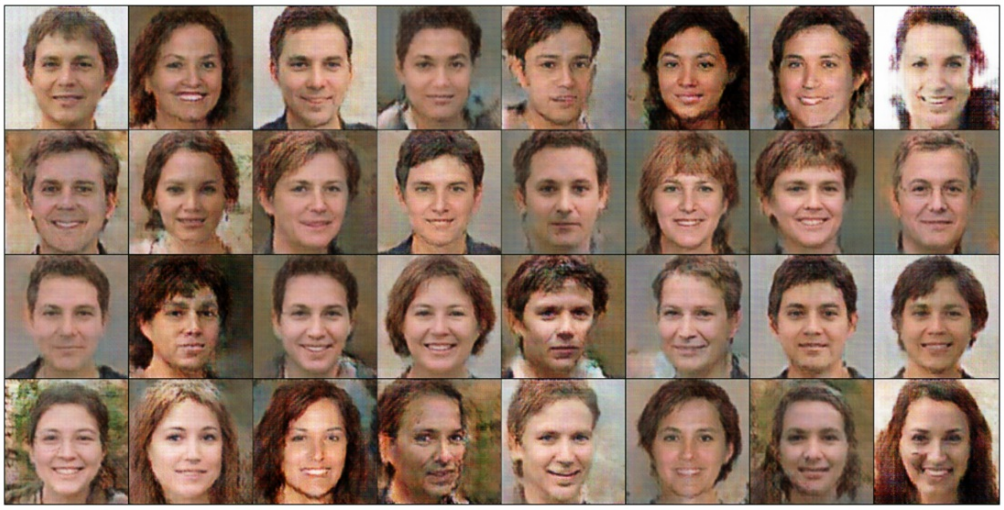
```
z_male_g=noise_g[0]
z_female_g=noise_g[14]
```

To generate 32 images without eyeglasses, we first produce another set of random noise vectors and labels:

```
noise_ng = torch.randn(32, z_dim, 1, 1)
labels_ng = torch.zeros(32, 2, 1, 1)
labels_ng[:,1,:,:]=1
```

The new set of random noise vectors is named `noise_ng`, and the new set of labels `labels_ng`. Feed them to the generator and you should see 32 images without eyeglasses, as shown in figure 5.6.

Figure 5.6 Images of human faces without eyeglasses that are generated by the trained conditional GAN model.



None of the 32 faces in figure 5.6 has eyeglasses in it: the trained conditional GAN model can generate images contingent upon the given label. We select images with indexes 8 (male) and 31 (female) to prepare for vector arithmetic in the next subsection:

```
z_male_ng=noise_ng[8]
z_female_ng=noise_ng[31]
```

Next, we'll use label interpolation to perform label arithmetic. Recall that the two labels, `noise_g` and `noise_ng`, instruct the trained conditional GAN model to create images with and without eyeglasses, respectively. What if we feed an interpolated label (a weighted average of the two labels [1, 0] and [0, 1]) to the model? What type of images will the trained generator produce? Let's find out:

Listing 5.10 Label arithmetic in conditional GAN

```
weights=[0,0.25,0.5,0.75,1]    #A
plt.figure(figsize=(20,4),dpi=300)
for i in range(5):
    ax = plt.subplot(1, 5, i + 1)
    # change the value of z
    label=weights[i]*labels_ng[0]+(1-weights[i])*labels_g[0]    #
    noise_and_labels=torch.cat(
        [z_female_g.reshape(1, z_dim, 1, 1),
         label.reshape(1, 2, 1, 1)],dim=1).to(device)
    fake=generator(noise_and_labels).cpu().detach()    #C
    img=(fake[0]/2+0.5).permute(1,2,0)
```

```
    plt.imshow(img)
    plt.xticks([])
    plt.yticks([])
plt.subplots_adjust(wspace=-0.08,hspace=-0.01)
plt.show()
```

We first create five weights (w): 0, 0.25, 0.5, 0.75, and 1, equally spaced between 0 and 1. Each of these five values of w is the weight we put on the no eyeglasses label `labels_ng`. The complementary weight is put on the eyeglasses label `labels_g`. The interpolated label therefore has a value of `w*labels_ng +(1-w)*labels_g`. We then feed the interpolated label to the trained model, along with the random noise vector `z_female_g`, that we saved earlier. The five generated images, based on the five values of w, are plotted in a one-by-five grid, as shown in figure 5.7.

Figure 5.7 Label arithmetic in conditional GAN. We first create two labels: the no eyeglasses label `labels_ng` and the eyeglasses label `labels_g`. These two labels instruct the trained generator to produce images with and without eyeglasses in them, respectively. We then create five interpolated labels, each as a weighted average of the original two labels: `w*labels_ng +(1-w)*labels_g`, where the weight `w` takes five different values, 0, 0.25, 0.5, 0.75, and 1. The five generated images based on the five interpolated labels are shown above. The image on the far left has eyeglasses. As we move from the left to the right, the eyeglasses gradually fade away, until the image on the far right has no eyeglasses in it.

When you look at the five generated images in figure 5.7 from the left to the right, you'll notice that the eyeglasses gradually fade away. The image on the left has eyeglasses in it while the image on the right has no eyeglasses in it. The three images in the middle show some signs of eyeglasses, but the eyeglasses are not as conspicuous as those in the first image.

Exercise 5.1

Since we used the random noise vector `z_female_g` in listing 5.10, the images in figure 5.7 have a female face in them. Change the noise vector to `z_male_g` in listing 5.10 and rerun the program; see what the images look

like.

5.5.2 Vector arithmetic in latent space

You may have noticed that some generated human face images have male features in them while others have female features in them. You may wonder: can we select male or female features in generated images? The answer is yes. We can achieve this by selecting noise vectors in the latent space.

In the last subsection, we have saved two random noise vectors, `z_male_ng` and `z_female_ng`, that lead to images of a male face and a female face, respectively. Below, we feed a weighted average of the two vectors (i.e., an interpolated vector) to the trained model and see what the generated images look like:

Listing 5.11 Vector arithmetic to select image characteristics

```
weights=[0,0.25,0.5,0.75,1]    #A
plt.figure(figsize=(20,4),dpi=50)
for i in range(5):
    ax = plt.subplot(1, 5, i + 1)
    # change the value of z
    z=weights[i]*z_female_ng+(1-weights[i])*z_male_ng    #B
    noise_and_labels=torch.cat(
        [z.reshape(1, z_dim, 1, 1),
         labels_ng[0].reshape(1, 2, 1, 1)],dim=1).to(device)
    fake=generator(noise_and_labels).cpu().detach()    #C
    img=(fake[0]/2+0.5).permute(1,2,0)
    plt.imshow(img)
    plt.xticks([])
    plt.yticks([])
plt.subplots_adjust(wspace=-0.08,hspace=-0.01)
plt.show()
```

We have created five weights, 0, 0.25, 0.5, 0.75, and 1. We iterate through the five weights and create five weighted averages of the two random noise vectors, `w*z_female_ng+(1-w)*z_male_ng`. We then feed the five vectors, along with the label, `labels_ng`, to the trained model to obtain five images, as shown in figure 5.8.

Figure 5.8 Vector arithmetic in GAN. We first save two random noise vectors `z_female_ng` and

`z_male_ng`. The two vectors lead to images of female and male faces, respectively. We then create five interpolated vectors, each as a weighted average of the original two vectors: `w*z_female_ng+(1-w)*z_male_ng`, where the weight `w` takes five different values, 0, 0.25, 0.5, 0.75, and 1. The five generated images based on the five interpolated vectors are shown above. The image on the far left has male features. As we move from the left to the right, the male features gradually fade away and the female features gradually appear, until the image on the far right shows a female face in it.

Vector arithmetic can transition from one instance of an image to another instance. Since we happen to have selected a male and a female image, when you look at the five generated images in figure 5.8 from the left to the right, you'll notice that male features gradually fade away and female features gradually appear. The first image shows an image with a male face while the last image shows an image with a female face in it.

Exercise 5.2

Since we used the label `labels_ng` in listing 5.11, the images in figure 5.8 have no eyeglasses in them. Change the label to `labels_g` in listing 5.11 and rerun the program to see what the images look like.

5.5.3 Select two characteristics simultaneously

So far, we have selected one characteristic at a time. By selecting the label, you have learned how to generate images with or without eyeglasses in them. By selecting a specific noise vector, you have learned how to select a specific instance of the generated image.

What if you want to select two characteristics (glasses and gender, for example) at the same time? There are four possible combinations of the two independent characteristics: male faces with glasses, male faces without glasses, female faces with glasses, and female faces without glasses. Below, we'll generate an image of each type:

Listing 5.12 Select two characteristics simultaneously in generated images

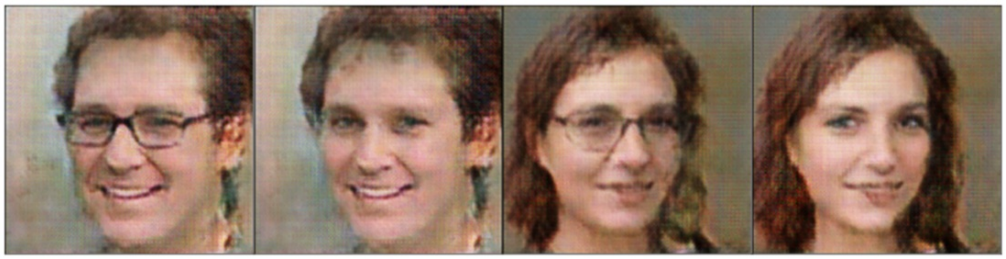
```
plt.figure(figsize=(20,5),dpi=50)
for i in range(4):    #A
    ax = plt.subplot(1, 4, i + 1)
    p=i//2
    q=i%2
    z=z_female_g*p+z_male_g*(1-p)    #B
    label=labels_ng[0]*q+labels_g[0]*(1-q)    #C
    noise_and_labels=torch.cat(
        [z.reshape(1, z_dim, 1, 1),
         label.reshape(1, 2, 1, 1)],dim=1).to(device)    #D
    fake=generator(noise_and_labels)
    img=(fake.cpu().detach()[0]/2+0.5).permute(1,2,0)
    plt.imshow(img.numpy())
    plt.xticks([])
    plt.yticks([])
plt.subplots_adjust(wspace=-0.08,hspace=-0.01)
plt.show()
```

To generate four images to cover the four different cases, we need to use one of the noise vectors as the input: `z_female_g` or `z_male_g`. We also need to attach to the input a label, which can be either `labels_ng` or `labels_g`. To use one single program to cover all four cases, we iterate through four values of i, 0 to 3, and create two values, p and q, which are the integer quotient and the remainder of the value i divided by 2. Therefore, the values of p and q can be either 0 or 1. By setting the value of the random noise vector to `z_female_g*p+z_male_g*(1-p)`, we can select a random noise vector to generate either a male or female face. Similarly, by setting the value of the label to `labels_ng[0]*q+labels_g[0]*(1-q)`, we can select a label to determine whether the generated image has eyeglasses in it or not. Once we combine the random noise vector with the label and feed them to the trained model, we can select two characteristics simultaneously.

If you run the program in listing 5.12, you'll see four images as shown in figure 5.9.

Figure 5.9 Select two characteristics simultaneously in the generated image. We select a noise vector from the following two choices `z_female_ng` and `z_male_ng`. We also select a label from the following two choices: `labels_ng` and `labels_g`. We then feed the noise vector and the label to the trained generator to create an image. Based on the values of the noise vector and the label, the trained model can create four types of images. By doing this, we effectively select two independent characteristics in the generated image: a male or a female face, and whether the

image has eyeglasses in it or not.



The four generated images in figure 5.9 have two independent characteristics: a male or a female face, and whether the image has eyeglasses in it or not. The first image shows an image with a male face with glasses, the second image is a male face without glasses. The third image is a female face with glasses while the last image shows a female face without glasses.

Exercise 5.3

We used the two random noise vectors `z_female_g` and `z_male_g` in listing 5.12. Change the two random noise vectors to `z_female_ng` and `z_male_ng` instead and rerun the program to see what the images look like.

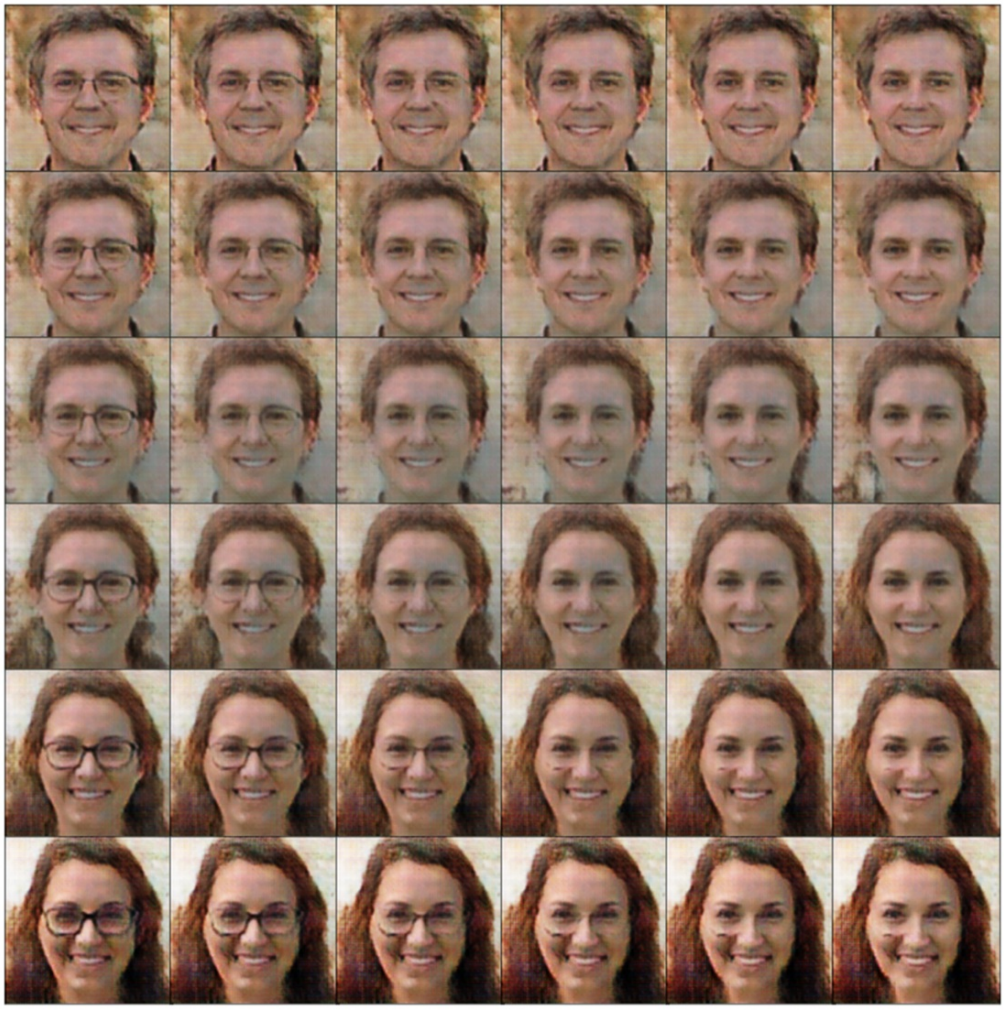
Finally, we can conduct label arithmetic and vector arithmetic simultaneously. That is, we can feed an interpolated noise vector and an interpolated label to the trained conditional GAN model and see what the generated image looks like. You can achieve that by running the following code block:

```
plt.figure(figsize=(20,20),dpi=50)
for i in range(36):
    ax = plt.subplot(6,6, i + 1)
    p=i//6
    q=i%6
    z=z_female_ng*p/5+z_male_ng*(1-p/5)
    label=labels_ng[0]*q/5+labels_g[0]*(1-q/5)
    noise_and_labels=torch.cat(
        [z.reshape(1, z_dim, 1, 1),
         label.reshape(1, 2, 1, 1)],dim=1).to(device)
    fake=generator(noise_and_labels)
    img=(fake.cpu().detach()[0]/2+0.5).permute(1,2,0)
    plt.imshow(img.numpy())
    plt.xticks([])
```

```
    plt.yticks([])
plt.subplots_adjust(wspace=-0.08,hspace=-0.01)
plt.show()
```

The code is similar to that in listing 5.12, except that p and q each can take six different values, 0, 1, 2, 3, 4, and 5. The random noise vector, `z_female_ng*p/5+z_male_ng*(1-p/5)`, takes six different values based on the value of p. The label, `labels_ng[0]*q/5+labels_g[0]*(1-q/5)`, takes six different values based on the value of q. We, therefore, have 36 different combinations of images based on the interpolated noise vector and the interpolated label. If you run the program above, you'll see 36 images as shown in figure 5.10.

Figure 5.10 Conduct vector arithmetic and label arithmetic simultaneously. The value of i changes from 0 to 35; p and q are the integer quotient and remainder, respectively, when i is divided by 6. Therefore, p and q each can take six different values, 0, 1, 2, 3, 4, and 5. The interpolated noise vector, `z_female_ng*p/5+z_male_ng*(1-p/5)`, and the interpolated label, `labels_ng[0]*q/5+labels_g[0]*(1-q/5)`, can each take six different values. In each row, when you go from left to right, the eyeglasses gradually fade away. In each column, when you go from top to bottom, the image changes gradually from a male face to a female face.



The are 36 images in figure 5.10. The interpolated noise vector is a weighted average of the two random noise vectors, `z_female_ng` and `z_male_ng`, which generate a female face and a male face, respectively. The label is a weighted average of the two labels, `labels_ng` and `labels_g`, which determine whether the generated image has eyeglasses in it or not. The trained model generates 36 different images based on the interpolated noise vector and the interpolated label. In each row, when you go from the left to the right, the eyeglasses gradually fade away. That is, we conduct label arithmetic in each row. In each column, when you go from the top to the bottom, the image changes gradually from a male face to a female face. That is, we conduct vector arithmetic in each column.

Exercise 5.4

In this project, there are two values in the label: one indicates eyeglasses and one indicates no eyeglasses. Therefore, we can use a binary value instead of onehot variables as labels. Change the programs in this chapter and use values 1 and 0 (instead of [1, 0] and [0, 1]) to represent images with and without glasses. Attach 1 or 0 to the random noise vector so that you feed a 101-value vector to the generator. Attach one channel to the input image before you feed it to the critic: if an image has eyeglasses in it, the fourth channel is filled with 0s; if the image has no eyeglasses in it, the fourth channel is filled with 1s. Then create a generator and a critic; use the training dataset to train them. The solution is provided in the book's GitHub repository, along with solutions to the other three exercises in this chapter.

Now that you have witnessed what GAN models are capable of, you'll explore deeper in the next chapter by conducting style transfers with GANs. For example, you'll learn how to build a CycleGAN model and train it using celebrity face images so that you can convert blond hair to black hair or black hair to blond hair in these images. The exact same model can be trained on other datasets: for example, you can train it on the human face dataset you used in this chapter so that you can add to or remove eyeglasses from human face images.

5.6 Summary

- By selecting a certain noise vector in the latent space and feeding it to the trained GAN model, we can select a certain characteristic in the generated image, such as whether the generated image has a male or female face in it.
- A conditional GAN is different from a traditional GAN. We train the model on labeled data and ask the trained model to generate data with a specific attribute. For example, one label tells the model to generate images of human faces with eyeglasses while another tells the model to create human faces without eyeglasses.
- After a conditional GAN is trained, we can use a series of weighted averages of the labels to generate images that transition from an image represented by one label to an image represented by another label. For example, a series of images in which the eyeglasses gradually fade away on the same person's face. We call this label arithmetic.

- We can also use a series of weighted averages of two different noise vectors to create images that transition from one attribute to another. For example, a series of images in which the male features gradually fade away and female features gradually appear. We call this vector arithmetic.
- Wasserstein GAN (WGAN) is a technique used to improve the training stability and performance of GAN models by using Wasserstein distance instead of the binary cross-entropy as the loss function. Further, for the Wasserstein distance to work correctly, the critic in WGANs must be 1-Lipschitz continuous, meaning the gradient norms of the critic's function must be at most 1 everywhere. The gradient penalty in WGANs adds a regularization term to the loss function to enforce the Lipschitz constraint more effectively.

[1] Martin Arjovsky, Soumith Chintala, and Leon Bottou, 2017, Wasserstein GAN, https://arxiv.org/abs/1701.07875 and Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville, 2017, Improved Training of Wasserstein GANs, https://arxiv.org/abs/1704.00028.

6 CycleGAN: Converting Blond Hair to Black Hair

This chapter covers

- The idea behind CycleGAN and cycle consistency loss
- Building a CycleGAN model to translate images from one domain to another
- Training a CycleGAN by using any dataset with two domains of images
- Training a CycleGAN to convert black hair to blond hair and vice versa

The GAN models we have discussed in the last three chapters are all trying to produce images that are indistinguishable from those in the training set.

You may be wondering: Can we translate images from one domain to another, such as transforming horses into zebras, converting black hair to blond hair or blond hair to black, adding or removing eyeglasses in images, turning photographs into paintings, or converting winter scenes to summer scenes?

It turns out you can, and you'll acquire such skills in this chapter through CycleGAN!

CycleGAN was introduced in a paper by Jun-Yan Zhu et al in 2017.[1] The key innovation of CycleGAN is its ability to learn to translate between domains without paired examples. CycleGAN has a variety of interesting and useful applications, such as simulating the aging or rejuvenation process on faces to assist digital identity verification or visualizing clothing in different colors or patterns without physically creating each variant to streamline the design process.

CycleGAN uses a cycle consistency loss function to ensure the original image can be reconstructed from the transformed image, encouraging the preservation of key features. The idea behind cycle consistency loss is truly

ingenious and deserves some highlight here. The CycleGAN in this chapter has two generators: let's call them the black hair generator and the blond hair generator, respectively. The black hair generator takes in an image with blond hair (instead of a random noise vector as you have seen before) and converts it to one with black hair, while the blond hair generator takes in an image with black hair and converts it to one with blond hair.

To train the model, we'll give a real image with black hair to the blond hair generator to produce a fake image with blond hair. We'll then give the fake blond hair image to the black hair generator to convert it back to an image with black hair. If both generators work well, there is little difference between the original image with black hair and the fake one after a roundtrip conversion. To train the CycleGAN, we adjust the model parameters to minimize the sum of adversarial losses and cycle consistency losses. As in Chapters 3 and 4, adversarial losses are used to quantify how well the generator can fool the discriminator and how well the discriminator can differentiate between real and fake samples. Cycle consistency loss, a unique concept in CycleGANs, measures the difference between the original image and the fake image after a roundtrip conversion. The inclusion of the cycle consistency loss in the total loss function is the key innovation in CycleGANs.

You'll use two new Python libraries in this chapter: `pandas` and `albumentations`. To install these libraries, execute the following line of code in a new cell in your Jupyter Notebook application on your computer:

```
!pip install pandas albumentations
```

Follow the instructions to finish the installation.

6.1 CycleGAN and Cycle Consistency Loss

CycleGAN extends the basic GAN architecture to include two generators and two discriminators. Each generator-discriminator pair is responsible for learning the mapping between two distinct domains. It aims to translate images from one domain to another (e.g., horses to zebras, summer to winter scenes, and so on) while retaining the key characteristics of the original

images. It uses a cycle consistency loss that ensures the original image can be reconstructed from the transformed image, encouraging the preservation of key features.

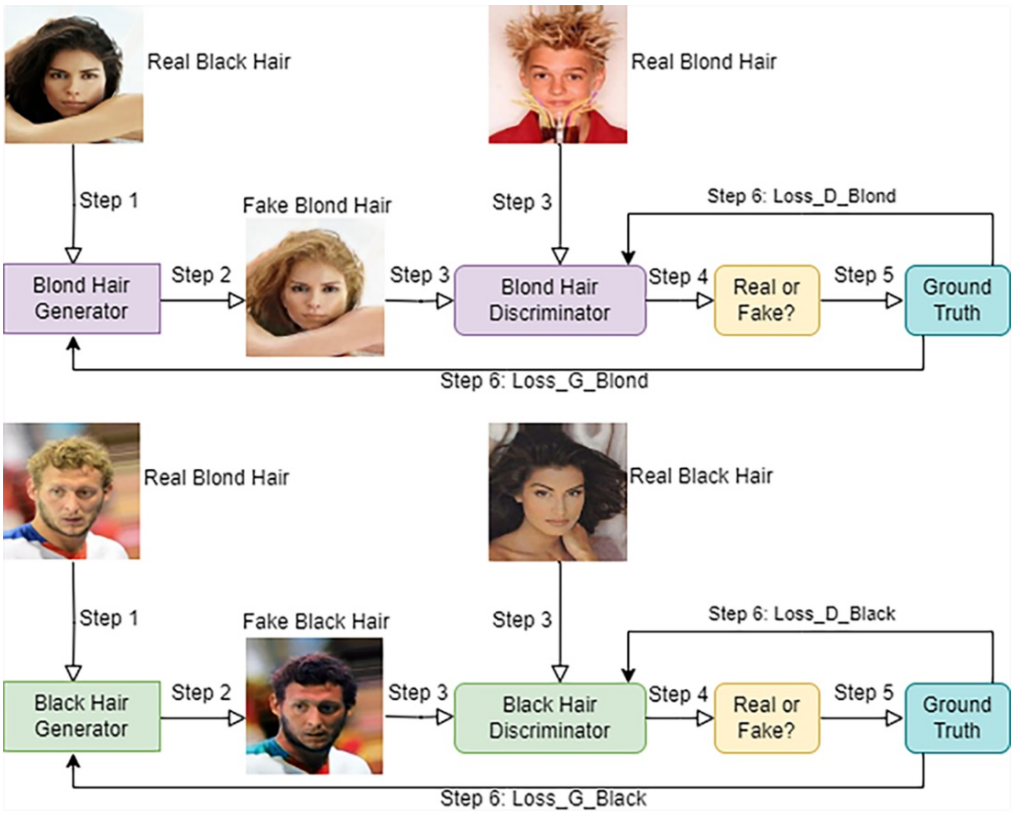
In this section, we'll first discuss the architecture of CycleGAN. We'll emphasize the key innovation of CycleGANs: cycle consistency loss.

6.1.1 What is CycleGAN?

CycleGAN consists of two Generators and two Discriminators. The generators translate images from one domain to another, while the discriminators determine the authenticity of the images in their respective domains. These networks are capable of transforming photographs into artworks mimicking the style of famous painters or specific art movements, thereby bridging the gap between art and technology. They can also be used in healthcare for tasks like converting MRI images to CT scans or vice versa, which can be helpful in situations where one type of imaging is unavailable or too costly.

For our project in this chapter, we'll convert between images with black hair and blond hair. We, therefore, use them as an example when explaining how CycleGAN works. Figure 6.1 is a diagram of the CycleGAN architecture.

Figure 6.1 The architecture of A CycleGAN to convert images with black hair to ones with blond hair and to convert images with blond hair to ones with black hair. The diagram also outlines the training steps to minimize adversarial losses. How the model minimizes cycle consistency losses is explained in figure 6.2.



To train CycleGAN, we use unpaired datasets from the two domains we wish to translate between. We'll use 48,472 celebrity face images with black hair and 29,980 ones with blond hair. We adjust the model parameters to minimize the sum of adversarial losses and cycle consistency losses. For ease of explanation, we'll explain only adversarial losses in figure 6.1. We'll explain how the model minimizes cycle consistency losses in the next subsection.

In each iteration of training, we feed real black hair images (top left in figure 6.1) to the blond hair generator to obtain fake blond hair images. We then feed the fake blond hair images, along with real blond hair images, to the blond hair discriminator (top middle). The blond hair discriminator produces a probability that each one is a real blond hair image. We then compare the predictions with the ground truth (whether an image is a true image with blond hair) and calculate the loss to the discriminator (Loss_D_Blond) as well as the loss to the generator (Loss_G_Blond).

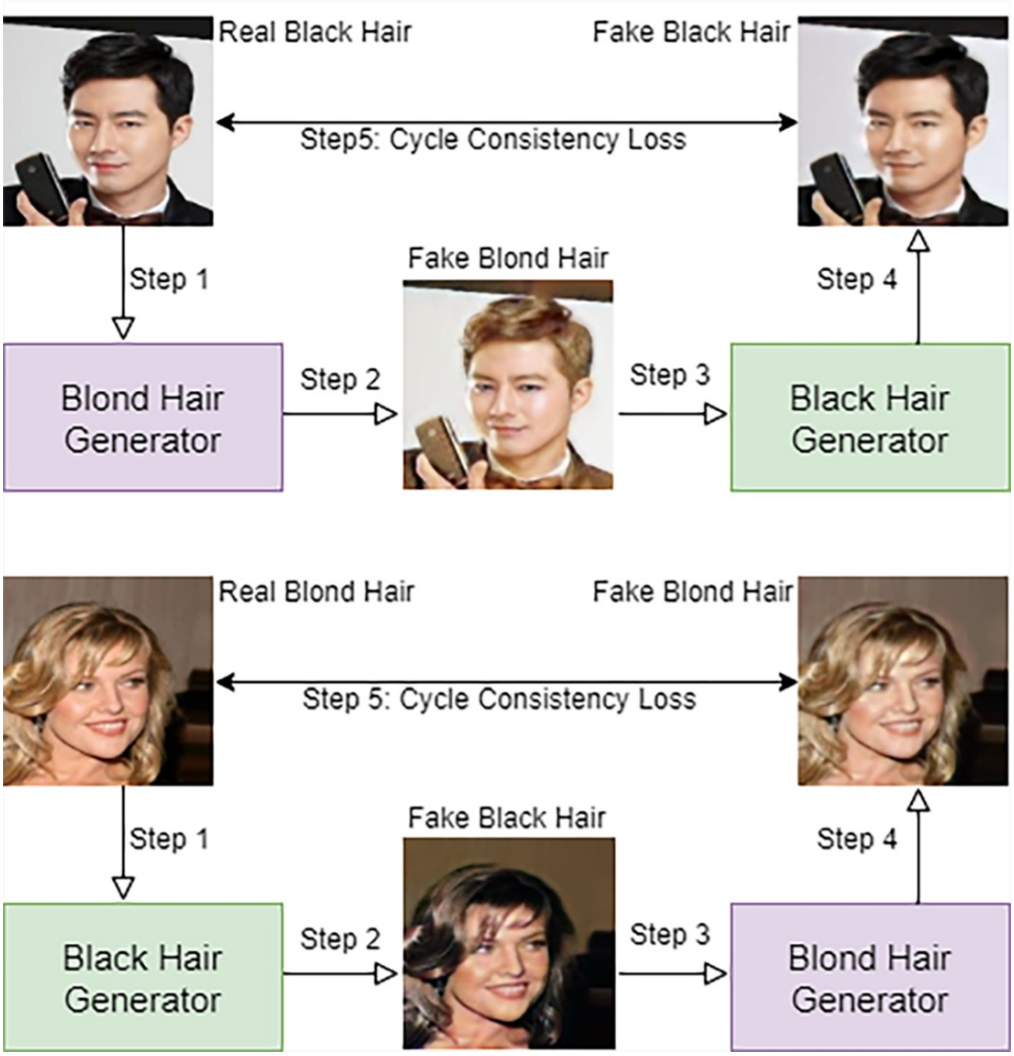
At the same time, in each iteration of training, we feed real blond hair images (middle left) to the black hair generator (bottom left) to create fake black hair

images. We present the fake black hair images, along with real ones, to the black hair discriminator (middle bottom) to obtain predictions that they are real. We compare the predictions from the black hair discriminator with the ground truth and calculate the loss to the discriminator (Loss_D_Black) and the loss to the generator (Loss_G_Black). We train the generators and discriminators simultaneously. To train the two discriminators, we adjust the model parameters to minimize the discriminator loss, which is the sum of Loss_D_Black and Loss_D_Blond.

6.1.2 Cycle consistency loss

To train the two generators, we adjust the model parameters to minimize the sum of the adversarial loss and cycle consistency loss. The adversarial loss is the sum of Loss_G_Black and Loss_G_Blond that we discussed in the previous subsection. To explain cycle consistency loss, let’s look at figure 6.2.

Figure 6.2 How CycleGAN minimizes cycle consistency losses between original black hair images and fake ones after roundtrips and cycle consistency losses between original blond hair images and fake ones after roundtrips.



The loss function for the generators in CycleGAN consists of two parts. The first part, the adversarial loss, ensures that generated images are indistinguishable from real images in the target domain. For example, Loss_G_Blond (defined in the previous subsection) ensures that fake blond images produced by the blond hair generator resemble real images with blond hair in the training set. The second part, the cycle consistency loss, ensures that an image translated from one domain to another can be translated back to the original domain.

The cycle consistency loss is a crucial component of CycleGANs, ensuring that the original input image can be recovered after a round-trip translation. The idea is that if you translate a real black hair image (top left in figure 6.2) to a fake blond hair image and convert it back to a fake black hair image

(bottom left), you should end up with an image close to the original black hair image. The cycle consistency loss for black hair images is the mean absolute error, at the pixel level, between the fake image and the original real one. Let's call this loss Loss_Cycle_Black. The same applies to translating blond hair to black hair and then back to blond hair, and we call this loss Loss_Cycle_Blond. The total cycle consistency loss is the sum of Loss_Cycle_Black and Loss_Cycle_Blond.

We used black and blond hair images as examples to explain CycleGAN. However, the model can be applied to any two domains. To drive home the message, I'll ask you to train the same CycleGAN model by using images with and without eyeglasses that you used in Chapter 5. The solution is provided in the book's GitHub repository, and you'll see that the trained model can indeed add or remove eyeglasses from human face images.

6.2 The celebrity faces dataset

We'll use celebrity face images with black hair and blond hair as the two domains. You'll first download the data in this section. You'll then process the images to get them ready for training later in this chapter.

6.2.1 Download the celebrity faces dataset

To download the celebrity faces dataset, log into Kaggle and go to the link https://www.kaggle.com/datasets/jessicali9530/celeba-dataset. Unzip the dataset after downloading and place all image files inside the folder /files/img_align_celeba/img_align_celeba/ on your computer (note there is a subfolder with the same name in the folder itself). There are about 200,000 images in the folder. Also download the file `list_attr_celeba.csv` from Kaggle and place it in the /files/ folder on your computer. The CSV file specifies various attributes of each image.

The celebrity faces dataset contains images with many different hair colors: brown, gray, black, blond, and so on. We'll select images with black or blond hair as our training set because these two types are the most abundant in the celebrity faces dataset. Run the code in listing 6.1 to select all images with black or blond hair:

Listing 6.1 Select images with black or blond hair

```
import pandas as pd
import os, shutil

df=pd.read_csv("files/list_attr_celeba.csv")    #A
os.makedirs("files/black", exist_ok=True)
os.makedirs("files/blond", exist_ok=True)    #B
folder="files/img_align_celeba/img_align_celeba"
for i in range(len(df)):
    dfi=df.iloc[i]
    if dfi['Black_Hair']==1:    #C
        try:
            oldpath=f"{folder}/{dfi['image_id']}"
            newpath=f"files/black/{dfi['image_id']}"
            shutil.move(oldpath, newpath)
        except:
            pass
    elif dfi['Blond_Hair']==1:    #D
        try:
            oldpath=f"{folder}/{dfi['image_id']}"
            newpath=f"files/blond/{dfi['image_id']}"
            shutil.move(oldpath, newpath)
        except:
            pass
```

We first use the `pandas` library to load the file `list_attr_celeba.csv` so that we know whether each image has black or blond hair in it. We then create two folders locally, /files/black/ and /files/blond/, to store images with black and blond hair, respectively. Listing 6.1 then iterates through all images in the dataset. If an image's attribute `Black_Hair` is 1, we move it to the folder /files/black/; if an image's attribute `Blond_Hair` is 1, we move it to the folder /files/blond/. You'll see 48,472 images with black hair and 29,980 images with blond hair. Figure 6.3 shows some examples of the images.

Figure 6.3 Sample images of celebrity faces with black or blond hair.

Images in the top row of figure 6.3 have black hair while images in the bottom row have blond hair. Further, the image quality is high: all faces are front and center, and hair colors are easy to identify. The quantity and quality of the training data will help the training of the CycleGAN model.

6.2.2 Process the black and blond hair image data

We'll generalize the CycleGAN model so that it can be trained on any dataset with two domains of images. We'll also define a `LoadData()` class to process the training dataset for the CycleGAN model. The function can be applied to any dataset with two domains, be that human face images with different hair colors, images with or without eyeglasses, or images with summer and winter scenes.

To that end, we have created a local module `ch06util`. Download the files `ch06util.py` and `__init__.py` from the book's GitHub repository and place them in the folder /utils/ on your computer. In the local module, we have defined the following `LoadData()` class:

Listing 6.2 The LoadData() class to process the training data in CycleGAN

```
class LoadData(Dataset):
    def __init__(self, root_A, root_B, transform=None):    #A
        super().__init__()
        self.root_A = root_A
        self.root_B = root_B
        self.transform = transform
        self.A_images = []
        for r in root_A:
            files=os.listdir(r)
            self.A_images += [r+i for i in files]
        self.B_images = []
```

```
        for r in root_B:    #B
            files=os.listdir(r)
            self.B_images += [r+i for i in files]
        self.len_data = max(len(self.A_images),
                            len(self.B_images))
        self.A_len = len(self.A_images)
        self.B_len = len(self.B_images)
    def __len__(self):    #C
        return self.len_data
    def __getitem__(self, index):    #D
        A_img = self.A_images[index % self.A_len]
        B_img = self.B_images[index % self.B_len]
        A_img = np.array(Image.open(A_img).convert("RGB"))
        B_img = np.array(Image.open(B_img).convert("RGB"))
        if self.transform:
            augmentations = self.transform(image=B_img,
                                           image0=A_img)
            B_img = augmentations["image"]
            A_img = augmentations["image0"]
        return A_img, B_img
```

The `LoadData()` class is inherited from the `Dataset` class in PyTorch. The two lists `root_A` and `root_B` contain folders of images in domains A and B, respectively. The class loads up images in the two domains and produces a pair of images, one from domain A and one from domain B so that we can use the pair to train the CycleGAN model later.

As we did in previous chapters, we create a data iterator with batches to improve computational efficiency, memory usage, and optimization dynamics in the training process, as follows:

Listing 6.3 Processing the black and blond hair images for training

```
transforms = albumentations.Compose(
    [albumentations.Resize(width=256, height=256),    #A
        albumentations.HorizontalFlip(p=0.5),
        albumentations.Normalize(mean=[0.5, 0.5, 0.5],
        std=[0.5, 0.5, 0.5],max_pixel_value=255),    #B
        ToTensorV2()],
    additional_targets={"image0": "image"})
dataset = LoadData(root_A=["files/black/"],
    root_B=["files/blond/"],
    transform=transforms)    #C
loader=DataLoader(dataset,batch_size=1,
```

```
    shuffle=True, pin_memory=True)    #D
```

We first define an instance of the `Compose()` class in the `albumentations` library (which is famous for fast and flexible image augmentations), and call it `transforms`. The class transforms the images in several ways: it resizes images to 256 by 256 pixels and normalizes the values to the range -1 to 1. The augmentations and increase in size boost the performance of the CycleGAN model and make the generated images realistic. We then apply the `LoadData()` class to the black and blond hair images. We set the batch size to 1 since the images have a large file size and we use a pair of images to train the model in each iteration. Setting the batch size to more than 1 may result in your machine running out of memory.

6.3 Build a CycleGAN model

We'll build a CyelGAN model from scratch in this section. We'll take great care to make our CycleGAN model general so that it can be trained using any dataset with two domains of images. As a result, we'll use A and B to denote the two domains, instead of, for example, black and blond hair images. As an exercise, you'll train the same CycleGAN model by using the eyeglasses dataset that you used in Chapter 5. This helps you apply the skills you learned in this chapter to other real-world applications by using a different dataset.

6.3.1 Create two discriminators

Even though CycleGAN has two discriminators, they are identical *ex ante*. Therefore, we'll create one single `Discriminator()` class and then instantiate the class twice: one instance is discriminator A and the other discriminator B. The two domains in CycleGAN are symmetric and it doesn't matter which domain we call domain A: images with black hair or images with blond hair.

Open the file `ch06util.py` you just downloaded. In it, we have defined the `Discriminator()` class as follows:

Listing 6.4 Define the Discriminator() class in CycleGAN

```
class Discriminator(nn.Module):
```

```
    def __init__(self, in_channels=3, features=[64,128,256,512]):
        super().__init__()
        self.initial = nn.Sequential(
            nn.Conv2d(in_channels,features[0],    #A
                kernel_size=4,stride=2,padding=1,
                padding_mode="reflect"),
            nn.LeakyReLU(0.2, inplace=True))
        layers = []
        in_channels = features[0]
        for feature in features[1:]:    #B
            layers.append(Block(in_channels, feature,
                stride=1 if feature == features[-1] else 2))
            in_channels = feature
        layers.append(nn.Conv2d(in_channels,1,kernel_size=4,    #
                stride=1,padding=1,padding_mode="reflect"))
        self.model = nn.Sequential(*layers)
    def forward(self, x):
        out = self.model(self.initial(x))
        return torch.sigmoid(out)    #D
```

The above code listing defines the discriminator network. The architecture is similar to the discriminator network in Chapter 4 and the critic network in Chapter 5. The main components are five `Conv2d` layers. We apply the sigmoid activation function on the last layer because the discriminator performs a binary classification problem. The discriminator takes a 3-channel color image as input and produces a single number between 0 and 1, which can be interpreted as the probability that the input image is a real image in the domain.

We then create two instances of the class and call them `disc_A` and `disc_B`, respectively, as follows:

```
from utils.ch06util import Discriminator, weights_init    #A
import torch

device = "cuda" if torch.cuda.is_available() else "cpu"
disc_A = Discriminator().to(device)
disc_B = Discriminator().to(device)    #B
weights_init(disc_A)
weights_init(disc_B)    #C
```

In the local module `ch06util`, we also defined a `weights_init()` function to initialize model weights. The function is defined similarly to the one in

Chapter 5. We then initialize weights in the two newly created discriminators, `disc_A` and `disc_B`.

Now that we have two discriminators, we'll create two generators next.

6.3.2 Create two generators

Similarly, we define a single `Generator()` class in the local module and instantiate the class twice: one instance is generator A, and the other is generator B. In the file `ch06util.py` you just downloaded, we have defined the `Generator()` class as follows:

Listing 6.5 The Generator() class in CycleGAN

```
class Generator(nn.Module):
    def __init__(self, img_channels, num_features=64,
                 num_residuals=9):
        super().__init__()
        self.initial = nn.Sequential(
            nn.Conv2d(img_channels,num_features,kernel_size=7,
                stride=1,padding=3,padding_mode="reflect",),
            nn.InstanceNorm2d(num_features),
            nn.ReLU(inplace=True))
        self.down_blocks = nn.ModuleList(
            [ConvBlock(num_features,num_features*2,kernel_size=3,
                       stride=2, padding=1),
            ConvBlock(num_features*2,num_features*4,kernel_size=3
                stride=2,padding=1)])
        self.res_blocks = nn.Sequential(    #B
            *[ResidualBlock(num_features * 4)
            for _ in range(num_residuals)])
        self.up_blocks = nn.ModuleList(
            [ConvBlock(num_features * 4, num_features * 2,
                    down=False, kernel_size=3, stride=2,
                    padding=1, output_padding=1),
                ConvBlock(num_features * 2, num_features * 1,
                    down=False,kernel_size=3, stride=2,
                    padding=1, output_padding=1)])
        self.last = nn.Conv2d(num_features * 1, img_channels,
            kernel_size=7, stride=1,
            padding=3, padding_mode="reflect")

    def forward(self, x):
        x = self.initial(x)
```

```
        for layer in self.down_blocks:
            x = layer(x)
        x = self.res_blocks(x)
        for layer in self.up_blocks:
            x = layer(x)
        return torch.tanh(self.last(x))    #D
```

The generator network consists of several `Conv2d` layers, followed by nine residual blocks (which I'll explain in detail below). After that, the network has two upsampling blocks that consist of a `ConvTranspose2d` layer, an `InstanceNorm2d` layer, and a `ReLU` activation. As we have done in previous chapters, we use the tanh activation function at the output layer, so the output pixels are all in the range of -1 to 1, the same as the images in the training set.

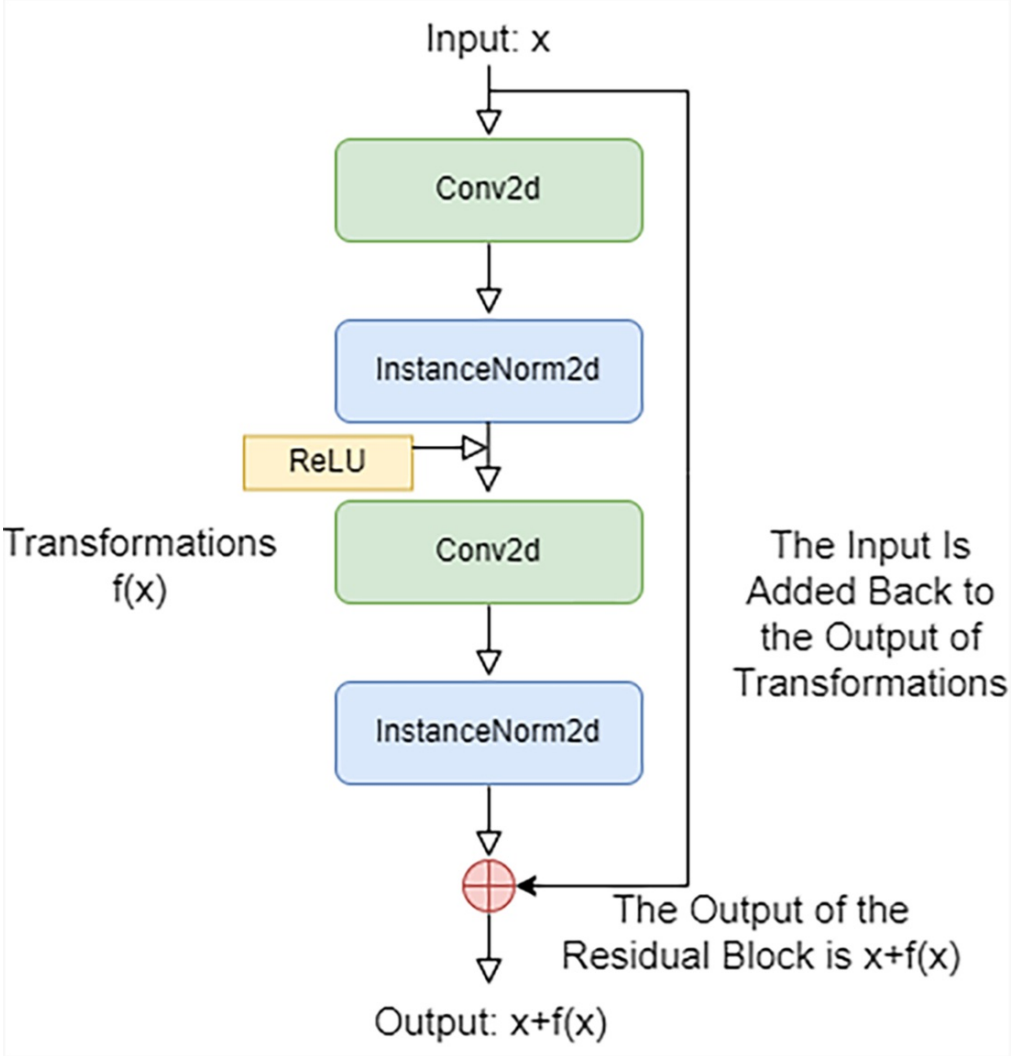
The residual block in the generator is defined in the local module as follows:

```
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels,
                 down=True, use_act=True, **kwargs):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels,
                      padding_mode="reflect", **kwargs)
            if down
            else nn.ConvTranspose2d(in_channels,
                                    out_channels, **kwargs),
            nn.InstanceNorm2d(out_channels),
            nn.ReLU(inplace=True) if use_act else nn.Identity())
    def forward(self, x):
        return self.conv(x)

class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super().__init__()
        self.block = nn.Sequential(
            ConvBlock(channels,channels,kernel_size=3,padding=1),
            ConvBlock(channels,channels,
                      use_act=False, kernel_size=3, padding=1))
    def forward(self, x):
        return x + self.block(x)
```

A residual connection is a concept in deep learning, particularly in the design of deep neural networks. You'll see it quite often later in this book. It's a

technique used to address the problem of vanishing gradients, which often occurs in very deep networks. It's often used when the neural network is deep and there is a concern about vanishing gradients. In a residual block, which is the basic unit of a network with residual connections, the input is passed through a series of transformations (like convolution, activation, and batch or instance normalization) and then added back to the output of these transformations. Figure 6.4 provides a diagram of the architecture of the residual block defined above.

Figure 6.4 The architecture of a residual block. The input x is passed through a series of transformations (two sets of Conv2d layer & InstanceNorm2d layer, and a ReLU activation). The input x is then added back to the output of these transformations, f(x). The output of the residual block is therefore x+f(x).



The transformations in each residual block are different. In this example, the input x is passed through two sets of `Conv2d` layer and `InstanceNorm2d` layer, and a ReLU activation in between. The input x is then added back to the output of these transformations, f(x), to form the final output, x+f(x); hence the name residual connection.

Next, we create two instances of the *Generator()* class and call one of them *gen_A* and the other *gen_B*, like so:

```
from utils.ch06util import Generator

gen_A = Generator(img_channels=3, num_residuals=9).to(device)
gen_B = Generator(img_channels=3, num_residuals=9).to(device)
weights_init(gen_A)
weights_init(gen_B)
```

When training the model, we'll use the mean absolute error (i.e., L1 loss) to measure the cycle consistency loss. We'll use the mean squared error (i.e., L2 loss) to gauge the adversarial loss. L1 loss is often used if the data are noisy and have many outliers since it punishes extreme values less than the L2 loss. Therefore, we import the following loss functions:

```
import torch.nn as nn

l1 = nn.L1Loss()
mse = nn.MSELoss()
g_scaler = torch.cuda.amp.GradScaler()
d_scaler = torch.cuda.amp.GradScaler()
```

Both L1 and L2 losses are calculated at the pixel level. The original image has a shape of (3, 256, 256) and so is the fake image. To calculate the losses, we first calculate the difference (absolute value of this difference for L1 loss and the squared value of this difference for L2 loss) between the corresponding pixel values between two images at each of the 3x256x256=196608 positions and average them over the positions.

We'll use PyTorch's automatic mixed precision package `torch.cuda.amp` to speed up training. The default data type in PyTorch tensors is `float32`, a 32-bit floating-point number, which takes up twice as much memory as a 16-bit

floating number, `float16`. Operations on the former are slower than those on the latter. There is a trade-off between precision and computational costs. Which data type to use depends on the task at hand. `torch.cuda.amp` provides an automatic mixed precision, where some operations use `float32` and others `float16`. Mixed precision tries to match each operation to its appropriate data type to speed up training.

As we have done in Chapter 4, we'll use the Adam optimizer for both the discriminators and the generators:

```
lr = 0.00001
opt_disc = torch.optim.Adam(list(disc_A.parameters()) +
  list(disc_B.parameters()),lr=lr,betas=(0.5, 0.999))
opt_gen = torch.optim.Adam(list(gen_A.parameters()) +
  list(gen_B.parameters()),lr=lr,betas=(0.5, 0.999))
```

Next, we'll train the CycleGAN model by using images with black or blond hair.

6.4 CycleGAN to translate between black and blond hair

Now that we have the training data and the CycleGAN model, we'll train the model by using images with black or blond hair. As with all GAN models, we'll discard the discriminators after training. We'll use the two trained generators to convert black hair images to blond hair ones and convert blond hair images to black hair ones.

6.4.1 Train a CycleGAN to translate between black and blond hair

As we explained in Chapter 4, we'll use visual inspections to determine when to stop training. To that end, we create a function to test what the real images look like and what the corresponding generated images look like so that we can compare the two to visually inspect the effectiveness of the model. In the local module `ch06util`, we define a `test()` function as follows:

```
def test(i,A,B,fake_A,fake_B):
    save_image(A*0.5+0.5,f"files/A{i}.png")
    save_image(B*0.5+0.5,f"files/B{i}.png")    #A
    save_image(fake_A*0.5+0.5,f"files/fakeA{i}.png")
    save_image(fake_B*0.5+0.5,f"files/fakeB{i}.png")    #B
```

We save four images after every 100 batches of training. We save real images and the corresponding fake images in the two domains in the local folder so we can periodically check the generated images and compare them with the real ones to assess the progress of training. We made the function general so that it can be applied to images from any two domains.

Further, we define a `train_epoch()` function in the local module `ch06util` to train the discriminators and the generators for an epoch. Listing 6.6 highlights the code we use to train the two discriminators:

Listing 6.6 Train the two discriminators in CycleGAN

```
def train_epoch(disc_A, disc_B, gen_A, gen_B, loader, opt_disc,
        opt_gen, l1, mse, d_scaler, g_scaler,device):
    loop = tqdm(loader, leave=True)
    for i, (A,B) in enumerate(loop):    #A
        A=A.to(device)
        B=B.to(device)
        with torch.cuda.amp.autocast():    #B
            fake_A = gen_A(B)
            D_A_real = disc_A(A)
            D_A_fake = disc_A(fake_A.detach())
            D_A_real_loss = mse(D_A_real,
                                torch.ones_like(D_A_real))
            D_A_fake_loss = mse(D_A_fake,
                                torch.zeros_like(D_A_fake))
            D_A_loss = D_A_real_loss + D_A_fake_loss
            fake_B = gen_B(A)
            D_B_real = disc_B(B)
            D_B_fake = disc_B(fake_B.detach())
            D_B_real_loss = mse(D_B_real,
                                torch.ones_like(D_B_real))
            D_B_fake_loss = mse(D_B_fake,
                                torch.zeros_like(D_B_fake))
            D_B_loss = D_B_real_loss + D_B_fake_loss
            D_loss = (D_A_loss + D_B_loss) / 2    #C
        opt_disc.zero_grad()
        d_scaler.scale(D_loss).backward()
```

```
        d_scaler.step(opt_disc)
        d_scaler.update()
        …
```

The training for the two discriminators is similar to what we have done in Chapter 4, with a couple of differences. First, instead of having just one discriminator, we have two discriminators here, one for images in domain A and one for images in domain B. The total loss for the two discriminators is the simple average of the adversarial losses of the two discriminators. Second, we use the PyTorch automatic mixed precision package to speed up training, reducing the training time by more than 50%.

We simultaneously train the two generators in the same iteration. Listing 6.7 highlights the code we use to train the two generators:

Listing 6.7 Train the two generators in CycleGAN

```
def train_epoch(disc_A, disc_B, gen_A, gen_B, loader, opt_disc,
        opt_gen, l1, mse, d_scaler, g_scaler,device):
        …
        with torch.cuda.amp.autocast():
            D_A_fake = disc_A(fake_A)
            D_B_fake = disc_B(fake_B)
            loss_G_A = mse(D_A_fake, torch.ones_like(D_A_fake))
            loss_G_B = mse(D_B_fake, torch.ones_like(D_B_fake))
            cycle_B = gen_B(fake_A)
            cycle_A = gen_A(fake_B)
            cycle_B_loss = l1(B, cycle_B)
            cycle_A_loss = l1(A, cycle_A)    #B
            G_loss=loss_G_A+loss_G_B+cycle_A_loss*10+cycle_B_loss
        opt_gen.zero_grad()
        g_scaler.scale(G_loss).backward()
        g_scaler.step(opt_gen)
        g_scaler.update()
        if i % 100 == 0:
            test(i,A,B,fake_A,fake_B)    #D
        loop.set_postfix(D_loss=D_loss.item(),G_loss=G_loss.item(
```

The training for the two generators is different from what we have done in Chapter 4 in two important ways. First, instead of having just one generator, we train two generators simultaneously here. Second, the total loss for the two generators is the weighted sum of adversarial losses and cycle consistency losses, as we have explained in the first section in this chapter.

The cycle consistency loss is the mean absolute error between the original image and the fake image that’s translated back to the original domain.

Now that we have everything ready, we'll start the training loop:

```
from utils.ch06util import train_epoch

for epoch in range(1):
    train_epoch(disc_A, disc_B, gen_A, gen_B, loader, opt_disc,
    opt_gen, l1, mse, d_scaler, g_scaler, device)    #A
torch.save(gen_A.state_dict(), "files/gen_black.pth")
torch.save(gen_B.state_dict(), "files/gen_blond.pth")    #B
```

The above training takes a couple of hours if you use GPU training. It may take a whole day otherwise. If you don't have the computing resources to train the model, download the pre-trained generators from my website https://gattonweb.uky.edu/faculty/lium/ml/hair.zip. Unzip the file and place the files `gen_black.pth` and `gen_blond.pth` in the folder /files/ on your computer. You'll be able to convert between black hair images and blond hair ones in the next subsection.

Exercise 6.1

When training the CycleGAN model, we assume that domain A contains images with black hair and domain B contains images with blond hair. Modify the code in listing 6.2 so that domain A contains images with blond hair and domain B contains images with black hair.

6.4.2 Roundtrip conversions of black hair images and blond hair images

Due to the high quality and the abundant quantity of the training dataset, we have trained the CycleGAN with great success. We’ll not only convert between images with black hair and images with blond hair, but we’ll also conduct roundtrip conversions. For example, we’ll convert images with black hair to images with blond hair and then convert them back to images with black hair. That way, we can compare the original images with the generated images in the same domain after a roundtrip and see the difference.

Listing 6.8 performs conversions of images between the two domains as well as roundtrip conversions of images in each domain:

Listing 6.8 Roundtrip conversions of images with black or blond hair

```
gen_A.load_state_dict(torch.load("files/gen_black.pth"))
gen_B.load_state_dict(torch.load("files/gen_blond.pth"))
i=1
for black,blond in loader:
    fake_blond=gen_B(black.to(device))
    save_image(black*0.5+0.5,f"files/black{i}.png")    #A
    save_image(fake_blond*0.5+0.5,f"files/fakeblond{i}.png")
    fake2black=gen_A(fake_blond)
    save_image(fake2black*0.5+0.5,f"files/fake2black{i}.png")
    fake_black=gen_A(blond.to(device))
    save_image(blond*0.5+0.5,f"files/blond{i}.png")    #C
    save_image(fake_black*0.5+0.5,f"files/fakeblack{i}.png")
    fake2blond=gen_B(fake_black)
    save_image(fake2blond*0.5+0.5,f"files/fake2blond{i}.png")
    i=i+1
    if i>10:
        break
```

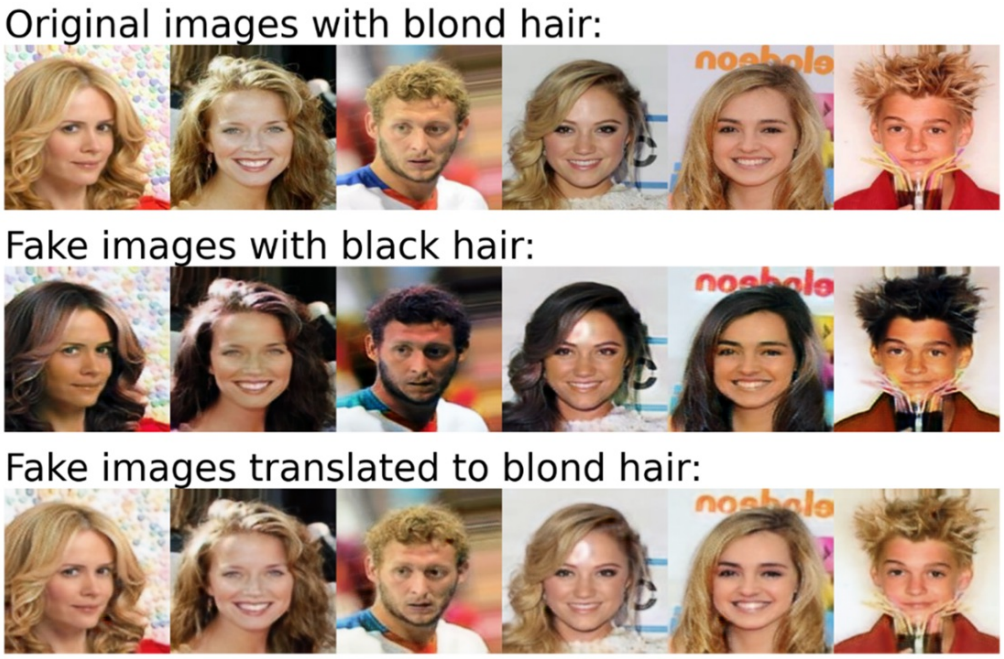
We have saved six sets of images in your local folder /files/. The first set is the original images with black hair. The second set is the fake blond images produced by the trained blond hair generator: the images are saved as `fakeblond0.png`, `fakeblond1.png`, and so on. The third set is the fake images with black hair after a roundtrip: we feed the fake images we just created to the trained black hair generator to obtain fake images with black hair: they are saved as `fake2black0.png`, `fake2black1.png`, and so on. Figure 6.5 shows the three sets of images.

Figure 6.5 A roundtrip conversion of images with black hair. Images in the top row are the original images with black hair from the training set. Images in the middle row are the corresponding fake images with blond hair, produced by the trained blond hair generator. Images in the bottom row are fake images with black hair after a roundtrip: we feed the images in the middle row to the trained black hair generator to create fake images with black hair.

There are three rows of images in figure 6.5. The top row displays original images with black hair from the training set. The middle row displays fake blond hair images produced by the trained blond hair. The bottom row contains fake black hair images after a roundtrip conversion: the images look almost identical to the ones in the top row! Our trained CycleGAN model works extremely well.

The fourth set of images in the local folder /files/ are the original images with blond hair. The fifth set is the fake image produced by the trained black hair generator. Finally, the sixth set contains fake images with blond hair after a roundtrip. Figure 6.6 compares these three sets of images.

Figure 6.6 A roundtrip conversion of images with blond hair. Images in the top row are the original images with blond hair from the training set. Images in the middle row are the corresponding fake images with black hair, produced by the trained black hair generator. Images in the bottom row are fake images with blond hair after a roundtrip conversion: we feed the images in the middle row to the trained blond hair generator to create fake images with blond hair.



In figure 6.6, fake black hair images produced by the trained black hair generator are shown in the middle row: they have black hair on the same human faces as the top row. Fake blond hair images after a roundtrip are shown in the bottom row: they look almost identical to the original blond hair images in the top row.

Exercise 6.2

The CycleGAN model is general and can be applied to any training dataset with two domains of images. Train the CycleGAN model using the eyeglasses images that you downloaded in Chapter 5. Use images with glasses as domain A and images without glasses as domain B. Then use the trained CycleGAN to add and remove eyeglasses from images (i.e., translating images between the two domains). An example implementation and results are in the book's GitHub repository.

So far, we have focused on one type of generative model, GANs. In the next chapter, you'll learn to use another type of generative model, variational autoencoders (VAEs), to generate high-resolution images. You'll learn the advantages and disadvantages of VAEs compared to GANs. More importantly, you'll learn the encoder-decoder architecture in VAEs. The architecture is widely used in generative models, including Transformers,

which we'll study later in the book.

6.5 Summary

- CycleGAN can translate images between two domains without paired examples. It consists of two discriminators and two generators. One generator converts images in domain A to domain B while the other generator converts images in domain B to domain A. The two discriminators classify if a given image is from a specific domain.
- CycleGAN uses a cycle consistency loss function to ensure the original image can be reconstructed from the transformed image, encouraging the preservation of key features.
- A properly constructed CycleGAN model can be applied to any dataset with images from two domains. The same model can be trained with different datasets and be used to translate images in different domains.
- When we have abundant high-quality training data, the trained CycleGAN can convert images in one domain to another and convert them back to the original domain. The images after a roundtrip conversion can potentially look almost identical to the original images.

[1] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Axexie Efros, 2017, Unpaired Image-to-Image Translation using Cycle Consistent Adversarial Networks. https://arxiv.org/abs/1703.10593.

7 Image Generation with Variational Autoencoders (VAEs)

This chapter covers

- Autoencoders (AEs) and variational autoencoders (VAEs) and their differences
- Building and training an autoencoder from scratch to generate handwritten digits
- Building and training a VAE from scratch to generate human face images
- Performing encoding arithmetic and encoding interpolation with the trained VAE

So far, you have learned how to generate shapes, numbers, and images, all by using generative adversarial networks (GANs). In this chapter, you'll learn to create images by using another generative model: autoencoders (AEs) and their variant, variational autoencoders (VAEs). You'll also learn the practical uses of VAEs by performing encoding arithmetic and encoding interpolation.

AEs have a dual-component structure: an encoder and a decoder. The encoder compresses the data into an abstract representation in a lower-dimensional space (the latent space), and the decoder decompresses the encoded information and reconstructs the data. The primary goal of an AE is to learn a compressed representation of the input data, focusing on minimizing the reconstruction error - the difference between the original input and its reconstruction (at the pixel level, as we have seen in Chapter 6 when calculating cycle consistency loss). The encoder-decoder architecture is a cornerstone in various generative models, including Transformers, which you'll explore in detail in the latter half of this book. For example, in Chapter 9, you'll build a Transformer for translating English to French: the encoder converts an English phrase into an abstract vector representation while the decoder constructs the French translation based on the compressed representation generated by the encoder. Text-to-image Transformers like

DALL-E 2 and Imagen also utilize an autoencoder architecture in their design. This involves first encoding an image into a compact, low-dimensional probability distribution. Then, they decode from this distribution. Of course, what constitutes an encoder and a decoder is different in different models.

Your first project in this chapter involves constructing and training an AE from scratch to generate handwritten digits. You'll use 60,000 grayscale images of handwritten digits (0 to 9), each with a size of 28x28=784 pixels, as the training data. The encoder in the AE compresses each image into a deterministic vector representation with only 20 values. The decoder in the AE reconstructs the image with the aim of minimizing the difference between the original image and the reconstructed image. This is achieved by minimizing the mean absolute error between the two images, at the pixel level. The end result is an AE capable of generating handwritten digits almost identical to those in the training set.

While AEs are good at replicating the input data, they often falter in generating new samples that are not present in the training set. More importantly, AEs are not good at input interpolation: they often fail to generate intermediate representations between two input data points. This leads us to variational autoencoders (VAEs). VAEs differ from AEs in two critical ways. Firstly, while an AE encodes each input into a specific point in the latent space, a VAE encodes it into a probability distribution within this space. Secondly, an AE focuses solely on minimizing the reconstruction error, whereas a VAE learns the parameters of the probability distribution for latent variables, minimizing a loss function that includes both reconstruction loss and a regularization term, the Kullback-Liebler (KL) divergence.

The KL-divergence encourages the latent space to approximate a normal distribution and ensures that the latent variables don't just memorize the training data but rather capture the underlying distribution. It helps in achieving a well-structured latent space where similar data points are mapped closely together, making the space continuous and interpretable. As a result, we can manipulate the encodings to achieve new outcomes, which makes encoding arithmetic and input interpolation possible in VAEs.

In the second project in this chapter, you'll build and train a VAE from the

ground up to generate human face images. Here, your training set comprises eyeglasses images that you downloaded in Chapter 5. The VAE's encoder compresses an image of size 3x256x256=196,608 pixels into a 100-value probabilistic vector, each following a normal distribution. The decoder then reconstructs the image based on this probabilistic vector. The trained VAE can not only replicate human faces from the training set but also generate novel ones.

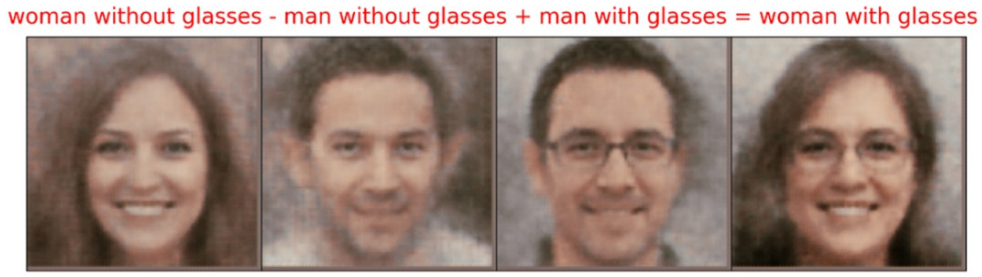
You'll learn how to conduct encoding arithmetic and input interpolation in VAEs. You'll manipulate the encoded representations (latent vectors) of different inputs to achieve specific outcomes (i.e., with or without certain characteristics in images) when decoded. The latent vectors control different characteristics in the decoded images such as gender, whether there are eyeglasses in an image, and so on. For example, you can first obtain the latent vectors for men with glasses (z1), women with glasses (z2), and women without glasses (z3). You then calculate a new latent vector, z4=z1-z2+z3. Since both z1 and z2 lead to eyeglasses in images when decoded, z1-z2 cancels out eyeglasses features in the resulting image. Similarly, since both z2 and z3 lead to a female face, z3-z2 cancels out female features in the resulting image. Therefore, if you decode z4=z1-z2+z3 with the trained VAE, you'll get an image of a man without glasses.

You'll also create a series of images transitioning from a woman with glasses to a woman without glasses by varying the weight assigned to the latent vectors z1 and z2 above. These exercises exemplify the versatility and creative potential of VAEs in the field of generative models.

Compared to generative adversarial networks (GANs) that we studied in the last few chapters, AEs and VAEs have a simple architecture and are easy to construct. Further, AEs and VAEs are generally easier and more stable to train relative to GANs. However, images generated by AEs and VAEs tend to be blurrier or less sharp compared to those generated by GANs. GANs excel in generating high-quality, realistic images but suffer from training difficulties and resource intensiveness. The choice between GANs and VAEs largely depends on the specific requirements of the task at hand, including the desired quality of the output, computational resources available, and the importance of having a stable training process.

VAEs have a wide range of practical applications in the real world. Consider, for instance, you run an eyewear store and have successfully marketed a new style of men's glasses online. Now, you wish to target the female market with the same style, but lack images of women wearing these glasses and face high costs for a professional photo shoot. Here's where VAEs come into play: you can combine existing images of men wearing the glasses with pictures of both men and women without glasses. This way, you can create realistic images of women sporting the same eyewear style, as illustrated in figure 7.1, through encoding arithmetic, a technique you'll learn in this chapter.

Figure 7.1 Generating images of women with glasses by performing encoding arithmetic.



In another scenario, suppose your store offers eyeglasses with dark and light frames, both of which are popular. You want to introduce a middle option with frames of an intermediate shade. With VAEs, through a method called encoding interpolation, you can effortlessly generate a smooth transition series of images, as shown in figure 7.2. These images would vary from dark to light-framed glasses, offering customers a visual spectrum of choices.

Figure 7.2 Generating a series of images that transition from glasses with dark frames to those with light frames.

The use of VAEs is not limited to eyeglasses; it extends to virtually any product category, be it clothing, furniture, or food. The technology provides a creative and cost-effective solution for visualizing and marketing a wide

range of products. Furthermore, although image generation is a prominent example, VAEs can be applied to many other types of data, including music and text. Their versatility opens up endless possibilities in terms of practical use!

7.1 An overview of autoencoders

This section discusses what an AE is and its basic structure. In order for you to have a deep understanding of the inner workings of AEs, you'll build and train an AE to generate handwritten digits as your first project in this chapter. This section provides an overview of an AE's architecture and a blueprint for completing the first project.

7.1.1 What is an autoencoder?

AEs are a type of neural network used in unsupervised learning, particularly effective for tasks like image generation, compression, and denoising. An AE consists of two main parts: an encoder and a decoder. The encoder compresses the input into a lower-dimensional representation (latent space), and the decoder reconstructs the input from this representation.

The compressed representation, or latent space, captures the most important features of the input data. In image generation, this space encodes crucial aspects of the images that the network has been trained on. AEs are useful for their efficiency in learning data representations and their ability to work with unlabeled data, making them suitable for tasks like dimensionality reduction and feature learning. One challenge with autoencoders is the risk of losing information in the encoding process, which can lead to less accurate reconstructions. Using deeper architectures with multiple hidden layers can help in learning more complex and abstract representations, potentially mitigating information loss in AEs. Also, training autoencoders to generate high-quality images can be computationally intensive and requires large datasets.

As we mentioned in Chapter 1, the best way to learn something is to create it from scratch. To that end, you'll learn to create an AE to generate handwritten digits in the first project in this chapter. The next subsection

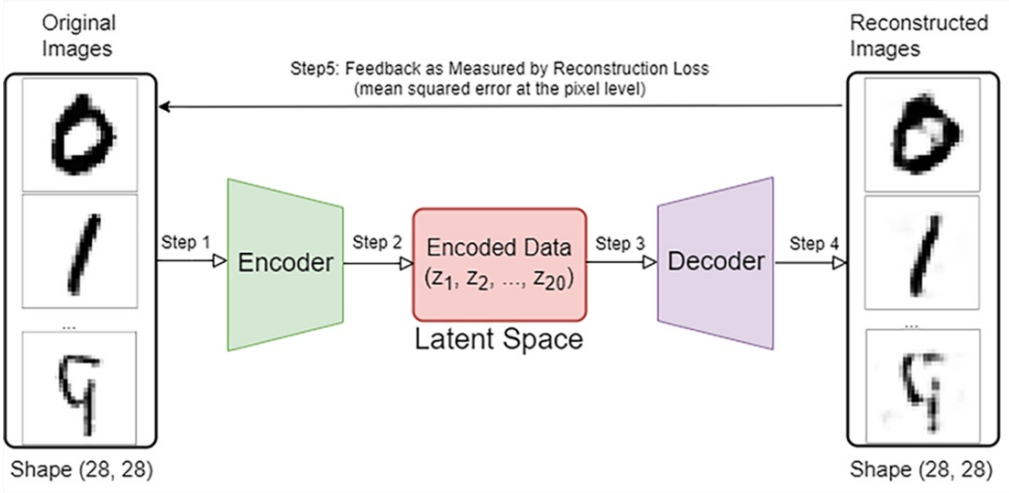
provides a blueprint for how to do that.

7.1.2 Steps in building and training an autoencoder

Imagine that you must build and train an AE from the ground up to generate grayscale images of handwritten digits so that you acquire the skills needed to use AEs for more complicated tasks such as color image generation or dimensionality reduction. How should you go about this task?

Figure 7.3 provides a diagram of the architecture of an AE and the steps involved in training an AE to generate handwritten digits.

Figure 7.3 The architecture of an autoencoder (AE) and the steps to train one to generate handwritten digits. An AE consists of an encoder (middle left) and a decoder (middle right). In each iteration of training, images of handwritten digits are fed to the encoder (step 1). The encoder compresses the images to deterministic points in the latent space (step 2). The decoder takes the encoded vectors (step 3) from the latent space and reconstructs the images (step 4). The AE adjusts its parameters to minimize the reconstruction loss: the difference between the originals and the reconstructions (step 5).



As you can see from the diagram, the AE has two main parts: an encoder (middle left) that compresses images of handwritten digits into vectors in the latent space and a decoder (middle right) that reconstructs these images based on the encoded vectors. Both the encoder and decoder are deep neural networks that can potentially include different types of layers such as dense layers, convolutional layers, transposed convolutional layers, and so on. Since our example involves grayscale images of handwritten digits, we'll use

only dense layers. However, AEs can also be used to generate higher-resolution color images; for those tasks, convolutional neural networks (CNNs) are usually included in encoders and decoders. Whether to use CNNs in AEs depends on the resolution of the images you want to generate.

When an AE is built, the parameters in it are randomly initialized. We need to obtain a training set to train the model: PyTorch provides 60,000 grayscale images of handwritten digits, evenly distributed among the ten digits, 0 to 9. The left side of figure 7.3 shows three examples, and they are images of digits 0, 1, and 9, respectively. In the first step in the training loop, we feed images in the training set to the encoder. The encoder compresses the images to 20-value vectors in the latent space (step 2). There is nothing magic about the number 20. If you use 25-value vectors in the latent space, you'll get similar results. We then feed the vector representations to the decoder (step 3) and ask it to reconstruct the images (step 4). We calculate the reconstruct loss, which is the mean squared error, over all the pixels, between the original image and the reconstructed image. We then propagate this loss back through the network to update the parameters in the encoder and decoder to minimize the reconstruction loss (step 5) so that in the next iteration, the AE can reconstruct images closer to the original ones. This process is repeated for many epochs over the dataset.

After the model is trained, you'll feed unseen images of handwritten digits to the encoder and obtain encodings. You then feed the encodings to the decoder to obtain reconstructed images. You'll notice that the reconstructed images look almost identical to the originals. The right side of figure 7.3 shows three examples of reconstructed images: they do look similar to the corresponding originals on the left side of the figure.

7.2 Build and train an autoencoder to generate digits

Now that you have a blueprint to build and train an AE to generate handwritten digits, let's dive into the project and implement the steps outlined in the last section.

Specifically, in this section, you'll learn how to first obtain a training set and

a test set of images of handwritten digits. You'll then build an encoder and decoder with dense layers. You'll train the AE with the training dataset and use the trained encoder to encode images in the test set. Finally, you'll learn to use the trained decoder to reconstruct images and compare them to the originals.

7.2.1 Gather handwritten digits

You can download grayscale images of handwritten images by using the *datasets* package in the Torchvision library, similar to how you downloaded images of clothing items in Chapter 2.

First, let's download a training set and a test set as follows:

```
import torchvision
import torchvision.transforms as T

transform=T.Compose([
    T.ToTensor()])
train_set=torchvision.datasets.MNIST(root=".",    #A
    train=True,download=True,transform=transform)    #B
test_set=torchvision.datasets.MNIST(root=".",
    train=False,download=True,transform=transform)    #C
```

Instead of using the `FashionMNIST()` class as we did in Chapter 2, we use the `MNIST()` class here. The `train` argument in the class tells PyTorch whether to download the training set (when the argument is set to `True`) or the test set (when the argument is set to `False`). Before transformation, the image pixels are integers ranging from 0 to 255. The `ToTensor()` class in the above code block converts them to PyTorch float tensors with values between 0 to 1. There are 60,000 images in the training set and 10,000 in the test set, evenly distributed among ten digits, 0 to 9, in each set.

We'll create batches of data for training and testing, with 32 images in each batch:

```
import torch

batch_size=32
train_loader=torch.utils.data.DataLoader(
    train_set,batch_size=batch_size,shuffle=True)
```

```
test_loader=torch.utils.data.DataLoader(
    test_set,batch_size=batch_size,shuffle=True)
```

Now that we have the data ready, we'll build and train an autoencoder next.

7.2.2 Build and train an autoencoder

An autoencoder consists of two parts: the encoder and the decoder. We'll define an `AE()` class as follows to represent the autoencoder:

Listing 7.1 Creating an autoencoder to generate handwritten digits

```
import torch.nn.functional as F
from torch import nn

device="cuda" if torch.cuda.is_available() else "cpu"
input_dim = 784    #A
z_dim = 20    #B
h_dim = 200
class AE(nn.Module):
    def __init__(self,input_dim,z_dim,h_dim):
        super().__init__()
        self.common = nn.Linear(input_dim, h_dim)
        self.encoded = nn.Linear(h_dim, z_dim)
        self.l1 = nn.Linear(z_dim, h_dim)
        self.decode = nn.Linear(h_dim, input_dim)
    def encoder(self, x):    #C
        common = F.relu(self.common(x))
        mu = self.encoded(common)
        return mu
    def decoder(self, z):    #D
        out=F.relu(self.l1(z))
        out=torch.sigmoid(self.decode(out))
        return out
    def forward(self, x):    #E
        mu=self.encoder(x)
        out=self.decoder(mu)
        return out, mu
```

The input size is 784 because the grayscale images of handwritten digits have a size of 28 by 28 pixels. We flatten the images to 1D tensors and feed them to the autoencoder. The images first go through the encoder: they are compressed into encodings in a lower dimensional space. Each image is now

represented by a 20-value latent variable. The decoder reconstructs the images based on the latent variables. The output from the autoencoder has two tensors: `out`, the reconstructed images, and mu, latent variables (i.e., encodings).

Next, we instantiate the `AE()` class we defined above to create an autoencoder. We also use the Adam optimizer during training, as we did in previous chapters:

```
model = AE(input_dim,z_dim,h_dim).to(device)
lr=0.00025
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

We define a function `test()` to visually inspect the reconstructed handwritten digits after each epoch of training, as follows:

Listing 7.2 Defining the test() function to visually inspect reconstructed images

```
import matplotlib.pyplot as plt

originals = []    #A
idx = 0
for img,label in test_set:
    if label == idx:
        originals.append(img)
        idx += 1
    if idx == 10:
        break
def test():
    reconstructed=[]
    for idx in range(10):
        with torch.no_grad():
            img = originals[idx].reshape((1,input_dim))
            out,mu = model(img.to(device))    #B
        reconstructed.append(out)    #C
    imgs=originals+reconstructed
    plt.figure(figsize=(10,2),dpi=50)
    for i in range(20):
        ax = plt.subplot(2,10, i + 1)
        img=(imgs[i]).detach().cpu().numpy()
        plt.imshow(img.reshape(28,28),    #D
                   cmap="binary")
        plt.xticks([])
```

```
        plt.yticks([])
    plt.show()
```

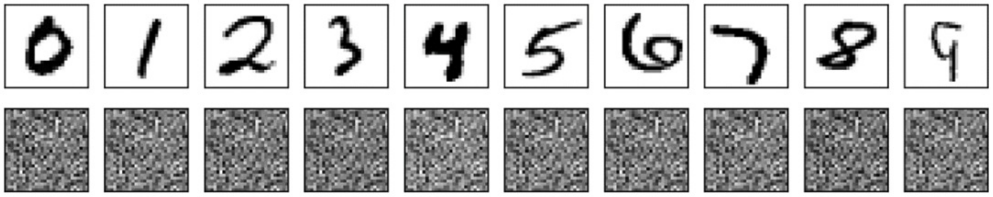
We first collect ten sample images, one representing a different digit, and place them in a list, `originals`. We feed the images to the autoencoder to obtain the reconstructed images. Finally, we plot both the originals and the reconstructed images so that we can compare them and assess the performance of the autoencoder periodically.

Before training starts, we call the function `test()` to visualize the output:

```
test()
```

You'll see the output as shown in figure 7.4.

Figure 7.4 Comparing reconstructed images by the autoencoder with the originals before training starts. The top row shows ten original images of handwritten digits in the test set. The bottom row shows the reconstructed images by the autoencoder before training. The reconstructions are nothing more than pure noise.



Though we could divide our data into training and validation sets and train the model until no further improvements are seen on the validation set (as we have done in Chapter 2), our primary aim here is to grasp how AEs work, not necessarily to achieve the best parameter tuning. Therefore, we'll train the autoencoder for ten epochs, as follows:

Listing 7.3 Training the autoencoder to generate handwritten digits

```
for epoch in range(10):
    tloss=0
    for imgs, labels in train_loader:    #A
        imgs=imgs.to(device).view(-1, input_dim)
        out, mu=model(imgs)    #B
        loss=((out-imgs)**2).sum()    #C
        optimizer.zero_grad()
```

```
        loss.backward()
        optimizer.step()
        tloss+=loss.item()
    print(f"at epoch {epoch} toal loss = {tloss}")
    test()    #D
```

In each epoch of training, we iterate through all batches of data in the training set. We feed the original images to the autoencoder to obtain the reconstructed images. We then calculate the reconstruction loss, which is the mean squared error between the original images and the reconstructed images, by calculating the difference between the two images, pixel by pixel, squaring the values, and averaging the squared difference. We adjust the model parameters to minimize the reconstruction loss, utilizing the Adam optimizer, which is a variation of the gradient descent method.

The model takes about two minutes to train if you are using GPU training. Alternatively, you can download the trained model from my website https://gattonweb.uky.edu/faculty/lium/gai/AEdigits.zip.

7.2.3 Save and use the trained autoencoder

We'll save the model in the local folder on your computer, as follows:

```
scripted = torch.jit.script(model)
scripted.save('files/AEdigits.pt')
```

To use it to reconstruct an image of handwritten digits, we load up the model as follows:

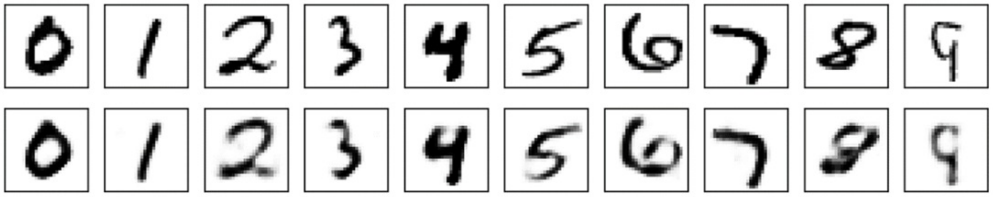
```
model=torch.jit.load('files/AEdigits.pt',map_location=device)
model.eval()
```

We can use it to generate handwritten digits by calling the `test()` function we defined earlier, as follows:

```
test()
```

The output is shown in figure 7.5.

Figure 7.5 Comparing reconstructed images by the autoencoder with the originals after training. The top row shows ten original images of handwritten digits in the test set. The bottom row shows the reconstructed images by the trained autoencoder. The reconstructed images look similar to the original ones.



The reconstructed handwritten digits do resemble the original ones, although the reconstruction is not perfect. Some information gets lost during the encoding-decoding process. However, compared to GANs, autoencoders are easy to construct and take less time to train. Further, the encoder-decoder architecture is employed by many generative models. This project will help your understanding of later chapters, especially when we explore Transformers.

7.3 What are variational autoencoders?

While autoencoders are good at reconstructing original images, they fail at generating novel images that are unseen in the training set. Further, AEs tend not to map similar inputs to nearby points in the latent space. As a result, the latent space associated with an AE is neither continuous nor easily interpretable. For example, you cannot interpolate two input data points to generate meaningful intermediate representations. For these reasons, we'll study an improvement in autoencoders: variational autoencoders (VAEs).

In this section, you'll first learn the key differences between AEs and VAEs and why these differences lead to the ability of the latter to generate realistic images that are unseen in the training set. You'll then learn the steps involved in training VAEs in general and training one to generate high-resolution human face images in particular.

7.3.1 Differences between AEs and VAEs

Variational autoencoders (VAEs) were first proposed by Diederik Kingma

and Max Welling in 2013.[1] They are a variant of AEs. Like an AE, a VAE also has two main parts: an encoder and a decoder.

However, there are two key differences between AEs and VAEs. First, the latent space in an AE is deterministic. Each input is mapped to a fixed point in the latent space. In contrast, the latent space in a VAE is probabilistic. Instead of encoding an input as a single vector in the latent space, a VAE encodes an input as a distribution over possible values. In our second project, for example, we'll encode a color image into a 100-value probabilistic vector. Additionally, we'll assume that each element in this vector adheres to an independent normal distribution. Since defining a normal distribution requires just the mean (μ) and standard deviation (σ), each element in our 100-element probabilistic vector will be characterized by these two parameters. To reconstruct the image, we sample a vector from this distribution and decode it. The uniqueness of VAEs is highlighted by the fact that each sampling from the distribution results in a slightly varied output.

In statistical terms, the encoder in a VAE is trying to learn the true distribution of the training data x, $p(x|\theta)$, where θ is the parameters defining the distribution. For tractability, we usually assume that the distribution is normal. Because we only need the mean, μ, and standard deviation, σ, to define a normal distribution, we can rewrite the true distribution as $p(x|\theta)=p(x|\mu,\sigma)$. The decoder in the VAE generates a sample based on the distribution learned by the encoder. That is, the decoder generates an instance probabilistically from the distribution $p(x|\mu,\sigma)$.

The second key difference between AEs and VAEs lies in the loss function. When training an AE, we minimize the reconstruction loss so that the reconstructed images are as close to the originals as possible. In contrast, in VAEs, the loss function consists of two parts: the reconstruction loss and the Kullback-Leibler (KL) divergence. KL divergence is a measure of how one probability distribution diverges from a second, expected probability distribution. In VAEs, KL divergence is used to regularize the encoder by penalizing deviations of the learned distribution (the encoder's output) from a prior distribution (a standard normal distribution). This encourages the encoder to learn meaningful and generalizable latent representations. By penalizing distributions that are too far from the prior, KL divergence helps

in avoiding overfitting.

The KL divergence is calculated as follows in our setting:

$$KL\ Divergence = \sum_{n=1}^{100}\left(\frac{\sigma^2}{2}+\frac{\mu^2}{2}-\log(\sigma^2)-\frac{1}{2}\right)$$

The summation is taken over all 100 dimensions of the latent space. When the encoder compresses the images into standard normal distributions in the latent space, such that μ=0 and σ=1, the KL divergence becomes 0. In any other scenario, the value exceeds 0. Thus, the KL divergence is minimized when the encoder successfully compresses the images into standard normal distributions within the latent space.

7.3.2 The blueprint to train a VAE to generate human face images

In the second project in this chapter, you'll build and train a VAE from scratch to generate color images of human faces. The trained model can generate images that are unseen in the training set. Further, you can interpolate inputs to generate novel images that are intermediate representations between two input data points. Below is a blueprint for this second project.

Figure 7.6 provides a diagram of the architecture of a VAE and the steps in training a VAE to generate human face images.

Figure 7.6 The architecture of a variational autoencoder (VAE) and the steps to train one to generate human face images. A VAE consists of an encoder (middle upper left) and a decoder (middle bottom right). In each iteration of training, human face images are fed to the encoder (step 1). The encoder compresses the images to probabilistic points in the latent space (step 2; since we assume normal distributions, each probability point is characterized by a vector of means and a vector of standard deviations). We then sample encodings from the distribution and present them to the decoder. The decoder takes sampled encodings (step 3) and reconstructs images (step 4). The VAE adjusts its parameters to minimize the sum of reconstruction loss and the Kullback-Leibler (KL) divergence. The KL divergence measures the difference between the encoder's output and a standard normal distribution.

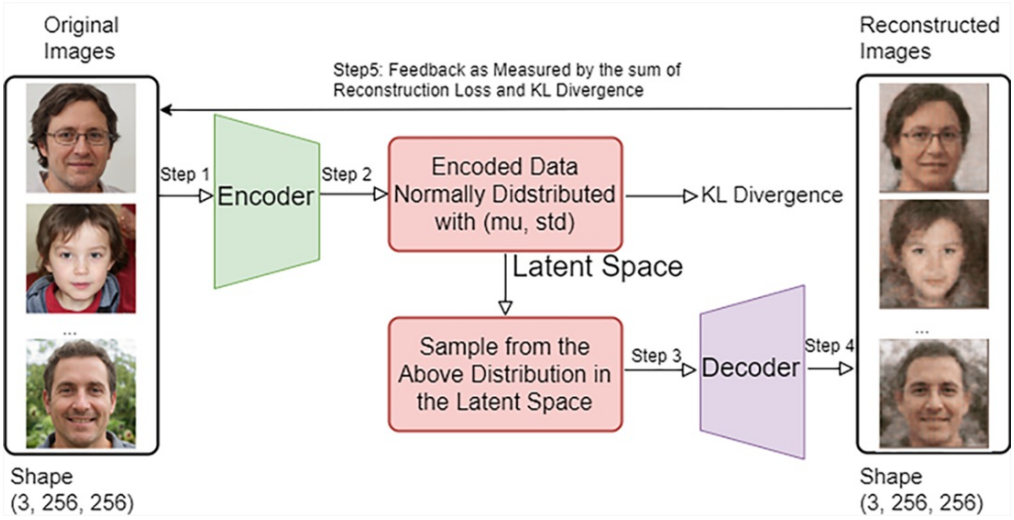


Figure 7.6 shows that a VAE also has two parts: an encoder (middle top left) and a decoder (middle bottom right). Since the second project involves high-resolution color images, we'll use convolutional neural networks (CNNs) to create the VAE. As we discussed in Chapter 4, high-resolution color images contain many more pixels than low-resolution grayscale images. If we use fully connected dense layers only, the number of parameters in the model is too large, making learning slow and ineffective. CNNs require fewer parameters than fully connected networks of similar size, leading to faster and more effective learning.

Once the VAE is created, you'll use the eyeglasses dataset that you have downloaded in Chapter 5 to train the model. The left side of figure 7.6 shows three examples of the original human face images in the training set. In the first step in the training loop, we feed images in the training set, with a size of 3x256x256=196,608 pixels, to the encoder. The encoder compresses the images to 100-value probabilistic vectors in the latent space (step 2; vectors of means and standard deviations due to the assumption of normal distribution). We then sample from the distribution and feed the sampled vector representations to the decoder (step 3) and ask it to reconstruct the images (step 4). We calculate the total loss as the sum of the reconstruction loss and the KL divergence. We propagate this loss back through the network to update the parameters in the encoder and decoder to minimize the total loss (step 5). The total loss encourages the VAE to encode the inputs into more meaningful and generalizable latent representations and to reconstruct images closer to the originals.

After the model is trained, you'll feed human face images to the encoder and obtain encodings. You then feed the encodings to the decoder to obtain reconstructed images. You'll notice that the reconstructed images look close to the originals. The right side of figure 7.6 shows three examples of reconstructed images: they look similar to the corresponding originals on the left side of the figure, though not perfectly.

More importantly, you can discard the encoder and randomly draw encodings from the latent space and feed them to the trained decoder in VAE to generate novel human face images that are unseen in the training set. Further, you can manipulate the encoded representations of different inputs to achieve specific outcomes when decoded. You can also create a series of images transitioning from one instance to another by varying the weight assigned to any two encodings.

7.4 A variational autoencoder to generate human face images

This section creates and trains a VAE from scratch to generate human face images, by following the steps outlined in the last section.

Compared to what we have done to build and train AEs, our approach for the second project incorporates several modifications. Firstly, we plan to use Convolutional Neural Networks (CNNs) in both the encoders and decoders of VAEs, particularly because high-resolution color images possess a greater number of pixels. Relying solely on fully connected dense layers would result in an excessively large number of parameters, leading to slow and inefficient learning. Secondly, as part of our process to compress images into vectors that follow a normal distribution in the latent space, we will generate both a mean vector and a standard deviation vector during the encoding of each image. This differs from the fixed value vector used in AEs. From the encoded normal distribution, we'll then sample to get encodings, which are subsequently decoded to produce images. Notably, each reconstructed image will vary slightly every time we sample from this distribution, which gives rise to VAEs' ability to generate novel images.

7.4.1 Build a variational autoencoder

If you recall, the eyeglasses dataset that you downloaded in Chapter 5 is saved in the folder /files/glasses/ on your computer after some labels are manually corrected. We'll resize the images to 256 by 256 pixels with values between 0 and 1. We then create a batch iterator with 16 images in each batch, as follows:

```
transform = T.Compose([
            T.Resize(256),    #A
            T.ToTensor(),    #B
            ])
data = torchvision.datasets.ImageFolder(
    root="files/glasses",
    transform=transform)    #C
batch_size=16
loader = torch.utils.data.DataLoader(data,    #D
     batch_size=batch_size,shuffle=True)
```

Next, we'll create a variational autoencoder that includes convolutional and transposed convolutional layers. We first define an `Encoder()` class as follows:

Listing 7.4 The encoder in the variational autoencoder

```
latent_dims=100    #A
class Encoder(nn.Module):
    def __init__(self, latent_dims=100):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 8, 3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(8, 16, 3, stride=2, padding=1)
        self.batch2 = nn.BatchNorm2d(16)
        self.conv3 = nn.Conv2d(16, 32, 3, stride=2, padding=0)
        self.linear1 = nn.Linear(31*31*32, 1024)
        self.linear2 = nn.Linear(1024, latent_dims)
        self.linear3 = nn.Linear(1024, latent_dims)
        self.N = torch.distributions.Normal(0, 1)
        self.N.loc = self.N.loc.cuda()
        self.N.scale = self.N.scale.cuda()
    def forward(self, x):
        x = x.to(device)
        x = F.relu(self.conv1(x))
        x = F.relu(self.batch2(self.conv2(x)))
        x = F.relu(self.conv3(x))
```

```
        x = torch.flatten(x, start_dim=1)
        x = F.relu(self.linear1(x))
        mu =  self.linear2(x)    #B
        std = torch.exp(self.linear3(x))    #C
        z = mu + std*self.N.sample(mu.shape)    #D
        return mu, std, z
```

The encoder network consists of several convolutional layers, which extract the spatial features of the input images. The encoder compresses the inputs into vector representations, `z`, which are normally distributed with means, `mu`, and standard deviations, `std`. The output from the encoder consists of three tensors: `mu`, `std`, and `z`. While the mu and std are the mean and standard deviation of the probabilistic vector, respectively, z is an instance sampled from this distribution.

Specifically, the input image, with a size of (3, 256, 256), first goes through a Conv2d layer with a stride value of 2. As we explained in Chapter 4, this means the filter skips two pixels each time it moves on the input image, which leads to downsampling of the image. The output has a size of (8, 128, 128). It then goes through two more Conv2d layers and the size becomes (32, 31, 31). It is flattened and passed through three linear layers.

We define a `Decoder()` class to represent the decoder in the VAE, as follows:

Listing 7.5 The decoder in the variational autoencoder

```
class Decoder(nn.Module):
    def __init__(self, latent_dims=100):
        super().__init__()
        self.decoder_lin = nn.Sequential(    #A
            nn.Linear(latent_dims, 1024),
            nn.ReLU(True),
            nn.Linear(1024, 31*31*32),    #B
            nn.ReLU(True))
        self.unflatten = nn.Unflatten(dim=1,
                  unflattened_size=(32,31,31))
        self.decoder_conv = nn.Sequential(    #C
            nn.ConvTranspose2d(32,16,3,stride=2,
                               output_padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(True),
            nn.ConvTranspose2d(16, 8, 3, stride=2,
                               padding=1, output_padding=1),
```

```
            nn.BatchNorm2d(8),
            nn.ReLU(True),
            nn.ConvTranspose2d(8, 3, 3, stride=2,
                               padding=1, output_padding=1))

    def forward(self, x):
        x = self.decoder_lin(x)
        x = self.unflatten(x)
        x = self.decoder_conv(x)
        x = torch.sigmoid(x)    #D
        return x
```

The decoder is a mirror image of the encoder: instead of performing convolutional operations, it performs transposed convolutional operations on the encodings to generate feature maps. It gradually converts encodings in the latent space back into high-resolution color images.

Specifically, the encoding first goes through two linear layers. It's then unflattened to a shape (32, 31, 31), mirroring the size of the image after the last Conv2d layer in the encoder. It then goes through three ConvTranspose2d layers, mirroring the Conv2d layers in the encoder. The output from the decoder has a shape of (3, 256, 256), the same as that of the training image.

We'll combine the encoder with the decoder to create a variational autoencoder, like so:

```
class VAE(nn.Module):
    def __init__(self, latent_dims=100):
        super().__init__()
        self.encoder = Encoder(latent_dims)    #A
        self.decoder = Decoder(latent_dims)    #B
    def forward(self, x):
        x = x.to(device)
        mu, std, z = self.encoder(x)    #C
        return mu, std, self.decoder(z)    #D
```

The VAE consists of an encoder and a decoder, as defined by the `Encoder()` and `Decoder()` classes above. When we pass images through the VAE, the output consists of three tensors: the mean and standard deviation of the encodings, and the reconstructed images.

Next, we create a variational autoencoder by instantiating the `VAE()` class and define the optimizer for the model:

```
vae=VAE().to(device)
lr=1e-4
optimizer=torch.optim.Adam(vae.parameters(),
                           lr=lr,weight_decay=1e-5)
```

We'll manually calculate the reconstruction loss and the KL-divergence loss during training. Therefore, we'll not define a loss function here.

7.4.2 Train the variational autoencoder

To train the model, we first define a `train_epoch()` function to train the model for one epoch:

Listing 7.6 Define the train_epoch() function

```
def train_epoch(epoch):
    vae.train()
    epoch_loss = 0.0
    for imgs, _ in loader:
        imgs = imgs.to(device)
        mu, std, out = vae(imgs)    #A
        reconstruction_loss = ((imgs-out)**2).sum()    #B
        kl = ((std**2)/2 + (mu**2)/2 - torch.log(std) - 0.5).sum(
        loss = reconstruction_loss + kl    #D
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_loss+=loss.item()
    print(f'at epoch {epoch}, loss is {epoch_loss}')
```

We iterate through all batches in the training set. We pass images through the VAE to obtain reconstructed images. The total loss is the sum of the reconstruction loss and the KL divergence. The model parameters are adjusted in each iteration to minimize the total loss.

We also define a `test_epoch()` function to visually inspect the generated images by the VAE, as follows:

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

def test_epoch():
    with torch.no_grad():
        noise = torch.randn(18,latent_dims).to(device)
        imgs = vae.decoder(noise).cpu()
        imgs = torchvision.utils.make_grid(imgs,6,3).numpy()
        fig, ax = plt.subplots(figsize=(6,3),dpi=100)
        plt.imshow(np.transpose(imgs, (1, 2, 0)))
        plt.axis("off")
        plt.show()
```

A well-trained VAE can map similar inputs to nearby points in the latent space, leading to a more continuous and interpretable latent space. As a result, we can randomly draw vectors from the latent space and the VAE can decode the vectors into meaningful inputs. Therefore, in the above function `test_epoch()`, we randomly draw 18 vectors from the latent space and use them to generate 18 images after each epoch of training. We plot them in a 3 by 6 grid and visually inspect them to see how the VAE is performing during the training process.

Next, we train the VAE for ten epochs, like so:

```
for epoch in range(1,11):
    train_epoch(epoch)
    test_epoch()
torch.save(vae.state_dict(),"files/VAEglasses.pth")
```

The above training takes about half an hour if you use GPU training or several hours otherwise. The trained model weights are saved on your computer. Alternatively, you can download the trained weights from my website https://gattonweb.uky.edu/faculty/lium/gai/VAEglasses.zip. Make sure you unzip the file after downloading.

7.4.3 Generate images with the trained VAE

Now that the VAE is trained, we can use it to generate images. We first load the weights of the trained model that we saved in the local folder, as follows:

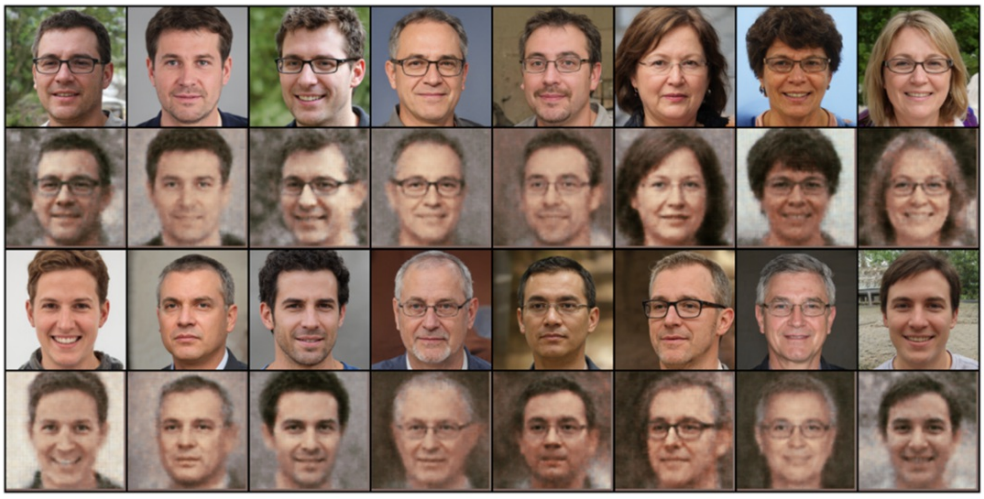
```
vae.eval()
vae.load_state_dict(torch.load('files/VAEglasses.pth'))
```

We then check the VAE's ability to reconstruct images and see how closely they resemble the originals, like so:

```
imgs,_=next(iter(loader))
imgs = imgs.to(device)
mu, std, out = vae(imgs)
images=torch.cat([imgs[:8],out[:8],imgs[8:16],out[8:16]],
                 dim=0).detach().cpu()
images = torchvision.utils.make_grid(images,8,4)
fig, ax = plt.subplots(figsize=(8,4),dpi=100)
plt.imshow(np.transpose(images, (1, 2, 0)))
plt.axis("off")
plt.show()
```

If you run the above code block, you'll see an output similar to figure 7.7.

Figure 7.7 Comparing the reconstructed images by a trained variational autoencoder (VAE) with the originals. The first and the third rows are the original images. We feed them to the trained VAE to obtain the reconstructed images, which are shown below the original images.



The original images are shown in the first and third rows, while the reconstructed images are shown below the originals. The reconstructed images resemble the originals, as shown in figure 7.7. However, some information gets lost during the reconstruction process: they don't look as

realistic as the originals.

Next, we test the VAE's ability to generate novel images that are unseen in the training set, by calling the test_epoch() function we defined before:

```
test_epoch()
```

The function randomly draws 18 vectors from the latent space and passes them to the trained VAE to generate 18 images. The output is shown in figure 7.8.

Figure 7.8 Novel images generated by the trained VAE. We randomly draw vector representations in the latent space and feed them to the decoder in the trained VAE. The decoded images are shown in this figure. Since the vector representations are randomly drawn, the images don't correspond to any originals in the training set.

These images are not present in the training set: the encodings are randomly drawn from the latent space, not the encoded vectors after passing images in the training set through the encoder. This is because the latent space in VAEs are continuous and interpretable. New and unseen encodings in the latent space can be meaningfully decoded into images that resemble but differ from those in the training set.

7.4.4 Encoding arithmetic with the trained VAE

VAEs include a regularization term (KL-divergence) in their loss function,

which encourages the latent space to approximate a normal distribution. This regularization ensures that the latent variables don't just memorize the training data but rather capture the underlying distribution. It helps in achieving a well-structured latent space where similar data points are mapped closely together, making the space continuous and interpretable. As a result, we can manipulate the encodings to achieve new outcomes.

To make results reproducible, I encourage you to download the trained weights from my website https://gattonweb.uky.edu/faculty/lium/gai/VAEglasses.zip and use the same code blocks for the rest of the chapter. As we explained in the introduction, encoding arithmetic allows us to generate images with certain features. To illustrate how encoding arithmetic works in VAEs, let's first hand-collect three images in each of the following four groups: men with glasses, men without glasses, women with glasses, and women without glasses. Like so:

Listing 7.7 Collecting images with different characteristics

```
torch.manual_seed(0)
glasses=[]
for i in range(25):    #A
    img,label=data[i]
    glasses.append(img)
    plt.subplot(5,5,i+1)
    plt.imshow(img.numpy().transpose((1,2,0)))
    plt.axis("off")
plt.show()
men_g=[glasses[0],glasses[3],glasses[14]]    #B
women_g=[glasses[9],glasses[15],glasses[21]]    #C

noglasses=[]
for i in range(25):    #D
    img,label=data[-i-1]
    noglasses.append(img)
    plt.subplot(5,5,i+1)
    plt.imshow(img.numpy().transpose((1,2,0)))
    plt.axis("off")
plt.show()
men_ng=[noglasses[1],noglasses[7],noglasses[22]]    #E
women_ng=[noglasses[4],noglasses[9],noglasses[19]])    #F
```

We select three images in each group instead of just one so that we can

calculate the average of multiple encodings in the same group when performing encoding arithmetic later. VAEs are designed to learn the distribution of the input data in the latent space. By averaging multiple encodings, we effectively smooth out the representation in this space. This helps us find an average representation that captures common features among different samples within a group.

Below, we first feed the three images of men with glasses to the trained VAE to obtain their encodings in the latent space. We then calculate the average encoding for the three images and use it to obtain a reconstructed image of a man with glasses. We then repeat this for the other three groups, like so:

Listing 7.8 Encoding and decoding images in four different groups

```
men_g_batch = torch.cat((men_g[0].unsqueeze(0),    #A
             men_g[1].unsqueeze(0),
             men_g[2].unsqueeze(0)), dim=0).to(device)
_,_,men_g_encodings=vae.encoder(men_g_batch)
men_g_encoding=men_g_encodings.mean(dim=0)    #B
men_g_recon=vae.decoder(men_g_encoding.unsqueeze(0))    #C
women_g_batch = torch.cat((women_g[0].unsqueeze(0),
             women_g[1].unsqueeze(0),
             women_g[2].unsqueeze(0)), dim=0).to(device)
men_ng_batch = torch.cat((men_ng[0].unsqueeze(0),
             men_ng[1].unsqueeze(0),
             men_ng[2].unsqueeze(0)), dim=0).to(device)
women_ng_batch = torch.cat((women_ng[0].unsqueeze(0),
             women_ng[1].unsqueeze(0),
             women_ng[2].unsqueeze(0)), dim=0).to(device)
_,_,women_g_encodings=vae.encoder(women_g_batch)
women_g_encoding=women_g_encodings.mean(dim=0)
_,_,men_ng_encodings=vae.encoder(men_ng_batch)
men_ng_encoding=men_ng_encodings.mean(dim=0)
_,_,women_ng_encodings=vae.encoder(women_ng_batch)
women_ng_encoding=women_ng_encodings.mean(dim=0)    #D
women_g_recon=vae.decoder(women_g_encoding.unsqueeze(0))
men_ng_recon=vae.decoder(men_ng_encoding.unsqueeze(0))
women_ng_recon=vae.decoder(women_ng_encoding.unsqueeze(0))    #E
```

The average encodings for the four groups are `men_g_encoding`, `women_g_encoding`, `men_ng_encoding`, and `women_ng_encoding`, respectively, where `g` stands for glasses and `ng` for no glasses. The decoded images for the four groups are `men_g_recon`, `women_g_recon`, `men_ng_recon`,

and `women_ng_recon`, respectively. We plot the four images below, like so:

```
imgs=torch.cat((men_g_recon,
                women_g_recon,
                men_ng_recon,
                women_ng_recon),dim=0)
imgs=torchvision.utils.make_grid(imgs,4,1).cpu().numpy()
imgs=np.transpose(imgs,(1,2,0))
fig, ax = plt.subplots(figsize=(8,2),dpi=100)
plt.imshow(imgs)
plt.axis("off")
plt.show()
```

You'll see the output as shown in figure 7.9.

Figure 7.9 Decoded images based on average encodings. We first obtain three images in each of the following four groups: men with glasses, women with glasses, men without glasses, and women without glasses. We feed the 12 images to the encoder in the trained VAE to obtain their encodings in the latent space. We then calculate the average encoding of the three images in each group. The four average encodings are fed to the decoder in the trained VAE to obtain four images and they are shown in this figure.

Since both men_g_encoding and women_g_encoding lead to eyeglasses in images when decoded, men_g_encoding - women_g_encoding cancels out eyeglasses features in the resulting image. Similarly, since both women_ng_encoding and women_g_encoding lead to a female face, women_ng_encoding - women_g_encoding cancels out female features in the resulting image. Therefore, if you decode men_g_encoding + women_g_encoding -women_ng_encoding with the trained VAE, you'll get an image of a man without glasses.

The four decoded images are shown in figure 7.9. They are the composite images representing the four groups. Notice that they are different from any of the original 12 images. At the same time, they preserve the defining

characteristics of each group.

Next, let's manipulate the encodings to create a new encoding and then use the trained decoder in the VAE to decode the new encoding and see what happens. For example, we can subtract the average encoding of women with glasses from the average encoding of men with glasses and add the average encoding of women without glasses. We then feed the result to the decoder and see the output, like so:

Listing 7.9 An example of encoding arithmetic

```
z=men_g_encoding-women_g_encoding+women_ng_encoding    #A
out=vae.decoder(z.unsqueeze(0))    #B
imgs=torch.cat((men_g_recon,
                women_g_recon,
                women_ng_recon,out),dim=0)
imgs=torchvision.utils.make_grid(imgs,4,1).cpu().numpy()
imgs=np.transpose(imgs,(1,2,0))
fig, ax = plt.subplots(figsize=(8,2),dpi=100)
plt.imshow(imgs)    #C
plt.title("man with glasses - woman \
with glasses + woman without \
glasses = man without glasses ",fontsize=10,c="r")    #D
plt.axis("off")
plt.show()
```

If you run the code block in listing 7.9, you'll see an output as shown in figure 7.10.

Figure 7.10 An example of encoding arithmetic with the trained VAE. We first obtain the average encodings for the following three groups: men with glasses (z1), women with glasses (z2), and women without glasses (z3). We define a new encoding z=z1-z2+z3. We then feed z to the decoder in the trained VAE and obtain the decoded image, as shown at the far right of this figure.

The first three images in figure 7.10 are the composite images representing the three input groups. The output image, at the far right, is an image of a man without glasses. The encoding arithmetic in this example shows that an encoding for men without glasses can be obtained by manipulating the average encodings in the other three groups.

Exercise 7.1

Perform the following encoding arithmetics by modifying code listing 7.9: (i) Subtract the average encoding of men without glasses from the average encoding of men with glasses and add the average encoding of women without glasses. Feed the result to the decoder and see what happens; (ii) Subtract the average encoding of women without glasses from the average encoding of men without glasses and add the average encoding of women with glasses. Feed the result to the decoder and see what happens; (iii) Subtract the average encoding of men without glasses from the average encoding of women without glasses and add the average encoding of men with glasses. Feed the result to the decoder and see what happens. Make sure you modify the image titles to reflect the changes. The solutions are provided in the book's GitHub repo.

Further, we can interpolate any two encodings in the latent space by assigning different weights to them and creating a new encoding. We can then decode the new encoding and create a composite image as a result. By choosing different weights, we can create a series of intermediate images that transition from one image to another.

Let's use the encodings of women with and without glasses as an example. We'll define a new encoding `z` as `w*women_ng_encoding+(1-w)*women_g_encoding`, where `w` is the weight we put on `women_ng_encoding`. We'll change the value of `w` from 0 to 1 with an increment of 0.2 in each step. We then decode them and display the resulting six images, as follows:

Listing 7.10 Interpolating two encodings to create a series of intermediate images

```
results=[]
for w in [0, 0.2, 0.4, 0.6, 0.8, 1.0]:    #A
    z=w*women_ng_encoding+(1-w)*women_g_encoding    #B
```

```
    out=vae.decoder(z.unsqueeze(0))    #C
    results.append(out)
imgs=torch.cat((results[0],results[1],results[2],
                results[3],results[4],results[5]),dim=0)
imgs=torchvision.utils.make_grid(imgs,6,1).cpu().numpy()
imgs=np.transpose(imgs,(1,2,0))
fig, ax = plt.subplots(dpi=100)
plt.imshow(imgs)    #D
plt.axis("off")
plt.show()
```

After running the code in listing 7.10, you'll see an output as shown in figure 7.11.

Figure 7.11 Interpolating encodings to create a series of intermediate images. We first obtain the average encodings for women with glasses (women_g_encoding) and women without glasses (women_ng_encoding). The interpolated encoding z is defined as w*women_ng_encoding+(1-w)*women_g_encoding, where w is the weight on women_ng_encoding. We change the value of w from 0 to 1 with an increment of 0.2 to create six interpolated encodings. We then decode them and display the resulting six images in the figure.

As you can see in figure 7.11, as you move from left to right, the image gradually transitions from a woman with glasses to a woman without glasses. This shows that the encodings in the latent space are continuous, meaningful, and interpolatable.

Exercise 7.2

Modify listing 7.10 to create a series of intermediate images by using the following pairs of encodings: (i) men_ng_encoding and men_g_encoding; (ii) men_ng_encoding and women_ng_encoding; (iii) men_g_encoding and women_g_encoding. The solutions are provided in the book's GitHub repo.

Starting in the next chapter, you'll embark on a journey in natural language processing (NLP). This will enable you to generate another form of content: text. However, many tools you have used so far will be used again in later

chapters, such as deep neural networks and the encoder-decoder architecture.

7.5 Summary

- Autoencoders (AEs) have a dual-component structure: an encoder and a decoder. The encoder compresses the data into an abstract representation in a lower-dimensional space (the latent space), and the decoder decompresses the encoded information and reconstructs the data.
- Variational autoencoders (VAEs) also consist of an encoder and a decoder. They differ from AEs in two critical ways. Firstly, while an AE encodes each input into a specific point in the latent space, a VAE encodes it into a probability distribution within this space. Secondly, an AE focuses solely on minimizing the reconstruction error, whereas a VAE learns the parameters of the probability distribution for latent variables, minimizing a loss function that includes both reconstruction loss and a regularization term, the Kullback-Liebler (KL) divergence.
- The KL divergence in the loss function when training VAEs ensures the distribution for latent variables resembles a normal distribution. This encourages the encoder to learn continuous, meaningful, and generalizable latent representations.
- A well-trained VAE can map similar inputs to nearby points in the latent space, leading to a more continuous and interpretable latent space. As a result, VAEs can decode random vectors in the latent space into meaningful inputs, leading to outputs that are unseen in the training set.
- The latent space in a VAE is continuous and interpretable, different from that in an AE. As a result, we can manipulate the encodings to achieve new outcomes. We can also create a series of intermediate images transitioning from one instance to another by varying weights on two encodings in the latent space.

[1] Diederik P Kingma and Max Welling, 2013, Auto-Encoding Variational Bayes. https://arxiv.org/abs/1312.6114.

8 Text Generation with Recurrent Neural Networks (RNNs)

This chapter covers

- The idea behind RNNs and why they can handle sequential data
- Character tokenization, word tokenization, and subword tokenization
- How word embedding works
- Building and training an RNN to generate text
- Using temperature and top-K sampling to control the creativeness of text generation

So far in this book, we have discussed how to generate shapes, numbers, and images. Starting from this chapter, we'll focus mainly on text generation. Generating text is often considered the holy grail of generative AI for several compelling reasons. Human language is incredibly complex and nuanced. It involves not only understanding grammar and vocabulary but also context, tone, and cultural references. Successfully generating coherent and contextually appropriate text is a significant challenge that requires deep understanding and processing of language.

As humans, we primarily communicate through language. AI that can generate human-like text can interact more naturally with users, making technology more accessible and user-friendly. Text generation has many applications, from automating customer service responses to creating entire articles, scripting for games and movies, aiding in creative writing, and even building personal assistants. The potential impact across industries is enormous.

In this chapter, we'll make our first attempt at building and training models to generate text. You'll learn to tackle three main challenges in modeling text generation. First, text is sequential data, consisting of data points organized in a specific sequence, where each point is successively ordered to reflect the inherent order and interdependencies within the data. Predicting outcomes for

sequences is challenging due to their sensitive ordering. Altering the sequence of elements changes their meaning. Secondly, text exhibits long-range dependencies: the meaning of a certain part of the text depends on elements that appeared much earlier in the text (e.g., 100 words ago). Understanding and modeling these long-range dependencies is essential for generating coherent text. Lastly, human language is ambiguous and context dependent. Training a model to understand nuances, sarcasm, idioms, and cultural references to generate contextually accurate text is challenging.

You'll explore a specific neural network designed for handling sequential data, such as text or time series: the recurrent neural network (RNN). Traditional neural networks, such as feedforward neural networks or fully connected networks, treat each input independently. This means that the network processes each input separately, without considering any relationship or order between different inputs. In contrast, RNNs are specifically designed to handle sequential data. In an RNN, the output at a given time step depends not only on the current input but also on previous inputs. This allows RNNs to maintain a form of memory, capturing information from previous time steps to influence the processing of the current input.

This sequential processing makes RNNs suitable for tasks where the order of the inputs matters, such as language modeling, where the goal is to predict the next word in a sentence based on previous words. We'll focus on one variant of RNN, Long Short-Term Memory (LSTM) networks, which can recognize both short-term and long-term data patterns in sequential data like text. LSTM models use a hidden state to capture information in previous time steps. Therefore, a trained LSTM model can produce coherent text based on the context.

The style of the generated text depends on the training data. Additionally, as we plan to train a model from scratch for text generation, text length is a crucial factor. It needs to be sufficiently extensive for the model to effectively learn and mimic a particular writing style, yet concise enough to avoid excessive computational demands during training. As a result, we'll use the text from the novel Anna Karenina to train an LSTM model. Since neural networks like an LSTM cannot accept text as input directly, you'll learn to break down text into tokens (individual words in this chapter; but can be parts

of words, as you'll see in later chapters), a process known as tokenization. You'll then create a dictionary to map each unique token into an integer (i.e., an index). Based on this dictionary, you'll convert the text into a long sequence of integers, ready to be fed into a neural network.

You'll use sequences of indexes of a certain length as the input to train the LSTM model. You shift the sequence of inputs by one token to the right and use it as the output: you are effectively training the model to predict the next word in a sentence. This is the so-called *sequence-to-sequence* prediction problem in natural language processing (NLP) and you'll see it again in later chapters.

Once the LSTM is trained, you'll use it to generate text one token at a time based on previous tokens in the sequence as follows: You feed a prompt (part of a sentence such as "Anna and the") to the trained model. The model then predicts the most likely next token and appends the selected token to your prompt. The updated prompt serves again as the input and the model is used once more to predict the next token. The iterative process continues until the prompt reaches a certain length. This approach is similar to the mechanism employed by more advanced generative models like ChatGPT (though ChatGPT is not an LSTM). You'll witness the trained LSTM model generating grammatically correct and coherent text, with a style matching that of the original novel.

Finally, you also learn how to control the creativeness of the generated text by using temperature and top-K sampling. Temperature controls the randomness of the predictions of the trained model. A high temperature makes the generated text more creative while a low temperature makes the text more confident and predictable. Top-K sampling is a method where you select the next token from the top K most probable tokens, rather than selecting from the entire vocabulary. A small value of K leads to the selection of highly likely tokens in each step and this, in turn, makes the generated text less creative and more coherent.

The primary goal of this chapter is not necessarily to generate the most coherent text possible, which, as mentioned earlier, presents substantial challenges. Instead, our objective is to demonstrate the limitations of RNNs, thereby setting the stage for the introduction of Transformers in subsequent

chapters. More importantly, this chapter establishes the basic principles of text generation, including tokenization, word embedding, sequence prediction, temperature settings, and top-K sampling. Consequently, in the later chapters, you will have a solid understanding of the fundamentals of NLP. This foundation will allow us to concentrate on other, more advanced, aspects of NLP, such as how the attention mechanism functions and the architecture of Transformers.

8.1 Introduction to recurrent neural networks (RNNs)

At the beginning of this chapter, we touched upon the complexities involved in generating text, particularly when aiming for coherence and contextual relevance. This section delves deeper into these challenges and explores the architecture of RNNs. We'll explain why RNNs are suitable for the task and their limitations (which are the reasons they have been overtaken by Transformers).

RNNs are specifically designed to handle sequential data, making them capable of text generation, a task inherently sequential in nature. They utilize a form of memory, known as hidden states, to capture and retain information from earlier parts of the sequence. This capability is crucial for maintaining context and understanding dependencies as the sequence progresses.

In this chapter, we will specifically utilize LSTM networks, advanced versions of RNNs, for text generation, leveraging their advanced capabilities to tackle the challenges in this task.

8.1.1 Challenges in generating text

Text represents a quintessential example of sequential data, which is defined as any dataset where the order of elements is critical. This structuring implies that the positioning of individual elements relative to each other holds significant meaning, often conveying essential information for understanding the data. Examples of sequential data include time series (like stock prices), textual content (such as sentences), and musical compositions (a succession

of notes).

This book primarily zeroes in on text generation, although it also ventures into music generation in Chapters 12 and 13. The process of generating text is fraught with complexities. A primary challenge lies in modeling the sequence of words within sentences, where altering the order can drastically change the meaning. For instance, in the sentence “Kentucky defeated Vanderbilt in last night’s football game,” swapping 'Kentucky' and 'Vanderbilt' entirely reverses the sentence's implication, despite using the same words. Furthermore, as mentioned in the introduction, text generation encounters challenges in handling long-range dependencies and dealing with the issue of ambiguity.

In this chapter, we will explore one approach to tackle these challenges, namely, by using recurrent neural networks (RNNs). While this method isn't flawless, it lays the groundwork for more advanced techniques you'll encounter in later chapters. This approach will provide insight into managing word order, addressing long-range dependencies, and navigating the inherent ambiguity in text, equipping you with fundamental skills in text generation. The journey through this chapter serves as a stepping stone to more sophisticated methods and deeper understanding in the subsequent parts of the book. Along the way, you’ll acquire many valuable skills in natural language processing (NLP) such as text tokenization, word embedding, and sequence-to-sequence predictions.

8.1.2 How do recurrent neural networks (RNNs) work?

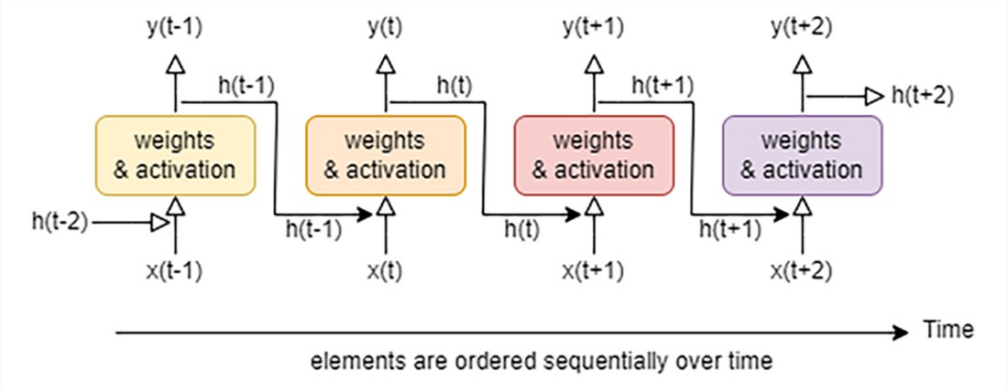
RNNs are a specialized form of artificial neural network designed to recognize patterns in sequences of data, such as text, music, or stock prices. Unlike traditional neural networks, which process inputs independently, RNNs have loops in them, allowing information to persist.

One of the challenges in generating text is how to predict the next word based on all previous words so that the prediction captures both the long-range dependencies and contextual meaning. RNNs take input not just as a standalone item but as a sequence (like words in a sentence, for example). At each time step, the prediction is based on not only the current input but also

all previous inputs, in the form of a summary through a hidden state. Let's consider the phrase "a frog has four legs" as an example. In the first time step, we use the word "a" to predict the second word "frog." In the second time step, we predict the next word using both "a" and "frog." By the time we predict the last word, we need to use all four previous words "a frog has four."

A key feature of RNNs is the so-called *hidden state*, which captures information in all previous elements in a sequence. This feature is crucial for the network's ability to process and generate sequential data effectively. The functioning of RNNs and this sequential processing is depicted in figure 8.1, which illustrates how a layer of recurrent neurons unfolds over time.

Figure 8.1 How a layer of recurrent neurons unfolds through time. When a recurrent neural network makes a prediction on sequential data, it takes the hidden state from the previous time step, h(t-1), along with the input at the current time step, x(t), and generates the output, y(t), and the updated hidden state, h(t). The hidden state at time step t captures the information in all previous time steps, x(0), x(1), …, x(t).



The hidden state in RNNs plays a pivotal role in capturing information across all time steps. This allows RNNs to make predictions that are informed not just by the current input, x(t), but also by the accumulated knowledge from all previous inputs, x(0), x(1), …, x(t-1). This attribute makes RNNs capable of understanding temporal dependencies. They can grasp the context from an input sequence, which is indispensable for tasks like language modeling where the preceding words in a sentence set the stage for predicting the next word.

However, RNNs are not without their drawbacks. Though standard RNNs are

capable of handling short-term dependencies, they struggle with longer-range dependencies within text. This difficulty stems from the vanishing gradient problem, which occurs in long sequences where the gradients (essential for training the network) diminish, hindering the model's ability to learn relationships over longer distances. To mitigate this, advanced versions of RNNs, such as long short-term memory (LSTM) networks, have been developed.

LSTM networks were introduced by Hochreiter & Schmidhuber in 1997.[1] An LSTM network is composed of LSTM units (or cells), each of which has a more complex structure than a standard RNN neuron. The cell state is the key innovation of LSTMs: it acts as a kind of conveyor belt, running straight down the entire chain of LSTM units. It has the ability to carry relevant information through the network. The ability to add or remove information to the cell state, carefully regulated by the gates, allows LSTMs to capture long-term dependencies and remember information for long periods. This makes them more effective for tasks like language modeling and text generation. In this chapter, we will harness the LSTM model to undertake a project on text generation, aiming to mimic the style of the novel *Anna Karenina*.

However, it's noteworthy that even advanced RNN variants like LSTMs encounter hurdles in capturing extremely long-range dependencies in sequence data. We will delve into these challenges and explore solutions in the next chapter, continuing our exploration of sophisticated models for effective sequence data processing and generation.

8.1.3 Steps in training a Long Short-Term Memory (LSTM) model

Next, we’ll discuss the steps involved in training an LSTM model to generate text. This overview aims to provide a foundational understanding of the training process before embarking on the project.

The choice of text for training depends on the desired output. A lengthy novel serves as a good starting point. Its extensive content enables the model to learn and replicate a specific writing style effectively. An ample amount of text data enhances the model's proficiency in this style. At the same time,

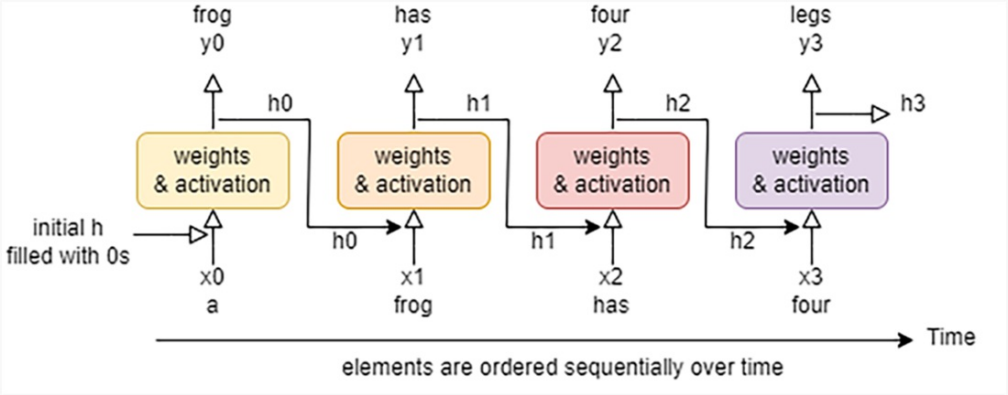
novels are generally not excessively long, which helps in managing the training time. For our LSTM model training, we'll utilize the text from *Anna Karenina*, aligning with our previously outlined training data criteria.

Similar to other deep neural networks, LSTM models cannot process raw text directly. Instead, we'll convert the text into numerical form. This begins by breaking down the text into smaller pieces, a process known as tokenization, where each piece is a token. Tokens can be entire words, punctuation marks (like an exclamation mark or a comma), or special characters (such as & or %). For this chapter, each of these elements will be treated as separate tokens. Although this method of tokenization may not be the most efficient, it is easy to implement since all we need is to map words to tokens. We will use subword tokenization in subsequent chapters where some infrequent words are broken into smaller pieces such as syllables. Following tokenization, we assign a unique integer to each token, creating a numerical representation of the text as a sequence of integers.

To prepare the training data, we divide this long sequence into shorter sequences of equal length. For our project, we'll use sequences comprising 100 integers each. These sequences form the features (the x variable) of our model. We then generate the output y by shifting the input sequence one token to the right. This setup enables the LSTM model to predict the next token in a sequence. The pairs of input and output serve as the training data. Our model includes LSTM layers to understand long-term patterns in the text and an embedding layer to grasp semantic meanings.

Let's revisit the example of predicting the sentence "a frog has four legs" that we mentioned earlier. Figure 8.2 is a diagram of how the training of the LSTM model works.

Figure 8.2 An example of how an LSTM model is trained. We first break down the training text into tokens and assign a unique integer to each token, creating a numerical representation of the text as a sequence of indexes. We then divide this long sequence into shorter sequences of equal length. These sequences form the features (the x variable) of our model. We then generate the output y by shifting the input sequence one token to the right. This setup enables the LSTM model to predict the next token based on previous tokens in the sequence.



In the first time step, the model uses the word “a” to predict the word “frog.” Since there's no preceding word for "a," we initialize the hidden state with zeros. The LSTM model receives both the index for "a" and this initial hidden state as input, and outputs the predicted next word along with an updated hidden state, h0. In the subsequent time step, the word "frog" and the updated state h0 are used to predict "has" and generate a new hidden state, h1. This sequence of predicting the next word and updating the hidden state continues until the model forecasts the final word in the sentence, "legs."

The predictions are then compared to the actual next word in the sentence. Since the model is effectively predicting the next token out of all possible tokens in the vocabulary, there is a multi-category classification problem. We tweak the model parameters in each iteration to minimize the cross-entropy loss, so that in the next iteration, the model predictions move closer to actual outputs in the training data.

Once the model is trained, generating text begins with a seed sequence input into the model. The model predicts the next token, which is then appended to your sequence. This iterative process of prediction and sequence updating is repeated to generate text for as long as desired.

8.2 Fundamentals of Natural Language Processing (NLP)

Deep learning models, including LSTM models that we discussed above and Transformers that you’ll learn in later chapters, cannot process raw text directly because they are designed to work with numerical data, typically in

the form of vectors or matrices. The processing and learning capabilities of neural networks are based on mathematical operations like addition, multiplication, and activation functions, which require numerical input. Consequently, it's essential first to break down text into smaller, more manageable elements known as tokens. These tokens can range from individual characters and words to subword units.

The next crucial step in natural language processing (NLP) tasks is transforming these tokens into numerical representations. This conversion is necessary for feeding them into deep neural networks, which is a fundamental part of training our models.

In this section, we'll discuss different tokenization methods, along with their advantages and drawbacks. Additionally, you'll gain insights into the process of converting tokens into dense vector representations, a method known as word embedding. This technique is pivotal in capturing the meaning of language in a format that deep learning models can effectively utilize.

8.2.1 Different tokenization methods

Tokenization involves dividing text into smaller parts, known as tokens, which can be in the form of words, characters, symbols, or other significant units. The primary goal of tokenization is to streamline the process of text data analysis and processing.

Broadly speaking, there are three approaches to tokenization. The first is character tokenization, where the text is divided into its constituent characters. This method is used in languages with complex morphological structures, such as Turkish or Finnish, in which the meaning of words can change significantly with slight variations in characters. Take the English phrase “It is unbelievably good!” as an example; it's broken down into individual characters as follows: ['I', 't', ' ', 'i', 's', ' ', 'u', 'n', 'b', 'e', 'l', 'i', 'e', 'v', 'a', 'b', 'l', 'y', ' ', 'g', 'o', 'o', 'd', '!']. A key advantage of character tokenization is the limited number of unique tokens. This limitation significantly reduces the parameters in deep learning models, leading to faster and more efficient training. However, the major drawback is that individual characters often lack significant meaning, making it challenging for machine learning models to

derive meaningful insights from a sequence of characters.

Exercise 8.1

Use character tokenization to divide the phrase “Hi, there!” into individual tokens.

The second approach is word tokenization, where the text is split into individual words and punctuation marks. It is used often in situations where the number of unique words is not too large. For instance, the same phrase “It is unbelievably good!” becomes five tokens: ['It', 'is', 'unbelievably', 'good', '!']. The main advantage of this method is that each word inherently carries semantic meaning, making it more straightforward for models to interpret the text. The downside, however, lies in the substantial increase in unique tokens, which escalates the number of parameters in deep learning models. This increase can lead to slower and less efficient training processes.

Exercise 8.2

Use word tokenization to break down the phrase “Hi, how are you?” into individual tokens.

The third approach is subword tokenization. This method, a key concept in NLP, breaks text into smaller, meaningful components called subwords. For instance, the phrase “It is unbelievably good!” would be divided into tokens like ['It', 'is', 'un', 'believ', 'ably', 'good', '!']. Most advanced language models, including ChatGPT, use subword tokenization, and you’ll use this method in the next few chapters. Subword tokenization strikes a balance between the more traditional tokenization techniques that typically split text into either individual words or characters. Word-based tokenization, while capturing more meaning, leads to a vast vocabulary. Conversely, character-based tokenization results in a smaller vocabulary, but each token encompasses less semantic value.

Subword tokenization effectively mitigates these issues by keeping frequently used words whole in the vocabulary, while dividing less common or more complex words into subcomponents. This technique is particularly advantageous for languages with large vocabularies or those exhibiting a high

degree of word form variation. By adopting subword tokenization, the overall vocabulary size is substantially reduced. This reduction enhances the efficiency and effectiveness of language processing tasks, especially when dealing with a wide range of linguistic structures.

In this chapter, we will focus on word tokenization, as it offers a straightforward foundation for beginners. As we progress to later chapters, our attention will shift to subword tokenization, utilizing models that have already been trained with this technique. This approach allows us to concentrate on more advanced topics, such as understanding the Transformer architecture and exploring the inner workings of the attention mechanism.

8.2.2 Word embedding

Word embedding is a method that transforms tokens into compact vector representations, capturing their semantic information and interrelationships. This technique is vital in natural language processing (NLP), especially since deep neural networks, including models like LSTM and Transformers, require numerical input.

Traditionally, tokens are converted into numbers using one-hot encoding before being fed into NLP models. In one-hot encoding, each token is represented by a vector where only one element is '1', and the rest are '0's. For example, in this chapter, there are 12,778 unique word-based tokens in the text for the novel *Anna Karenina*. Each token is represented by a vector of 12,778 dimensions. Consequently, a phrase like "happy families are all alike" is represented as a 5x12778 matrix, where 5 represents the number of tokens. This representation, however, is highly inefficient due to its large dimensionality, leading to an increased number of parameters, which can hinder training speed and efficiency.

LSTMs, Transformers, and other advanced NLP models address this inefficiency through word embedding. Instead of bulky one-hot vectors, word embedding uses continuous, lower-dimensional vectors (e.g., 128-length vectors we use in this chapter). As a result, the phrase "happy families are all alike" is represented by a more compact 5x128 matrix after word embedding. This streamlined representation drastically reduces the model's complexity

and enhances training efficiency.

Word embedding not only reduces word complexity by condensing it into a lower-dimensional space but also effectively captures the context and the intricate semantic relationships between words, a feature that simpler representations like one-hot encoding lack, for the following reasons. In one-hot encoding, all tokens have the same distance from each other in vector space. However, in word embeddings, similar tokens are represented by vectors that are close to each other in the embedding space. Word embeddings are learned from the text in the training data, the resulting vectors capture contextual information. Tokens that appear in similar contexts will have similar embeddings, even if they are not explicitly related.

Word Embedding in NLP

Word embeddings are a powerful method for representing tokens in natural language processing (NLP) that offer significant advantages over traditional one-hot encoding in capturing context and semantic relationships between words.

One-hot encoding represents tokens as sparse vectors with a dimension equal to the size of the vocabulary, where each token is represented by a vector with all zeros except for a single one at the index corresponding to the token. In contrast, word embeddings represent tokens as dense vectors with much lower dimensions (e.g., 128 dimensions in this chapter and 256 dimensions in Chapter 9). This dense representation is more efficient and can capture more information.

Specifically, in one-hot encoding, all tokens have the same distance from each other in the vector space, meaning there is no notion of similarity between tokens. However, in word embeddings, similar tokens are represented by vectors that are close to each other in the embedding space. For example, the words "king" and "queen" would have similar embeddings, reflecting their semantic relationship.

Word embeddings are learned from the text in the training data. The embedding process uses the context in which tokens appear to learn their embeddings, meaning that the resulting vectors capture contextual

information. Tokens that appear in similar contexts will have similar embeddings, even if they are not explicitly related.

Overall, word embeddings provide a more nuanced and efficient representation of words that captures semantic relationships and contextual information, making them more suitable for NLP tasks compared to one-hot encoding.

In practical terms, particularly in frameworks like PyTorch, word embedding is implemented by passing one-hot encoded vectors through a linear layer, which compresses them into a lower-dimensional space. The weights of this embedding layer are not predefined but are learned during the training process. This learning aspect enables the model to refine its understanding of word semantics based on the training data, leading to a more nuanced and context-aware representation of language in the neural network. This approach significantly enhances the model's ability to process and interpret language data efficiently and meaningfully.

8.3 Prepare data to train the LSTM model

In this section, we'll process text data and get them ready for training. We'll first break text down into individual tokens. Our next step involves creating a dictionary that assigns each token an index, essentially mapping them to integers. After this setup, we will organize these tokens into batches of training data, which will be crucial for training an LSTM model in the subsequent section.

We'll walk through the tokenization process in a detailed, step-by-step manner, ensuring you gain a thorough understanding of how tokenization functions. We'll use word tokenization, owing to its simplicity in dividing text into words, as opposed to the more complex subword tokenization that demands a nuanced grasp of linguistic structure. In later chapters, we'll employ pre-trained tokenizers for subword tokenization, leveraging more sophisticated methods. This will allow us to focus on advanced topics, such as the attention mechanism and the Transformer architecture, without getting bogged down in the initial stages of text processing.

8.3.1 Download and clean up the text

We'll use the text from the novel *Anna Karenina* to train our model. Go to https://github.com/LeanManager/NLP-PyTorch/tree/master/data to download the text file and save it as *anna.txt* in the folder /files/ on your computer. After that, open the file and delete everything after line 39888, which says "END OF THIS PROJECT GUTENBERG EBOOK ANNA KARENINA." Or you can simply download the file anna.txt from the book's GitHub repository https://github.com/markhliu/DGAI.

First, we load up the data and print out some passages to get a feeling about the dataset, as follows:

```
with open("files/anna.txt","r") as f:
    text=f.read()
words=text.split(" ")
print(words[:20])
```

The output is as follows:

```
['Chapter', '1\n\n\nHappy', 'families', 'are', 'all', 'alike;', '
```

As you can see, line breaks (represented by \n) are considered part of the text. Therefore, we should replace these line breaks with spaces so they are not in the vocabulary. Additionally, converting all words to lowercase is helpful in our setting, as it ensures words like "The" and "the" are recognized as the same token. This step is vital for reducing the variety of unique tokens, thereby making the training process more efficient. Furthermore, punctuation marks should be spaced apart from the words they follow. Without this separation, combinations like "way." and "way" would be erroneously treated as distinct tokens. To address these issues, we'll clean up the text in the following way:

```
clean_text=text.lower().replace("\n", " ")    #A
clean_text=clean_text.replace("-", " ")    #B
for x in ",.:;?!$()/_&%*@'`":
    clean_text=clean_text.replace(f"{x}", f" {x} ")
clean_text=clean_text.replace('"', ' " ')    #C
text=clean_text.split()
```

Next, we obtain unique tokens as follows:

```
from collections import Counter
word_counts = Counter(text)
words=sorted(word_counts, key=word_counts.get,
                      reverse=True)
print(words[:10])
```

The list `words` contains all the unique tokens in the text, with the most frequent one appearing first, and the least frequent one last. The output from the above code block is:

```
[',', '.', 'the', '"', 'and', 'to', 'of', 'he', "'", 'a']
```

The above output shows the most frequent ten tokens. The comma (,) and the period (.) are the most and the second most frequent tokens, respectively. The word "the" is the third most frequent token, and so on.

We now create two dictionaries: one mapping tokens to indexes and the other mapping indexes to tokens, as follows:

Listing 8.1 Dictionaries to map tokens to indexes and indexes to tokens

```
text_length=len(text)    #A
num_unique_words=len(words)    #B
print(f"the text contains {text_length} words")
print(f"there are {num_unique_words} unique tokens")
word_to_int={v:k for k,v in enumerate(words)}    #C
int_to_word={k:v for k,v in enumerate(words)}    #D
print({k:v for k,v in word_to_int.items() if k in words[:10]})
print({k:v for k,v in int_to_word.items() if v in words[:10]})
```

The output from the above code block is as follows:

```
the text contains 437098 words
there are 12778 unique tokens
{',': 0, '.': 1, 'the': 2, '"': 3, 'and': 4, 'to': 5, 'of': 6, 'h
{0: ',', 1: '.', 2: 'the', 3: '"', 4: 'and', 5: 'to', 6: 'of', 7:
```

The text for the novel *Anna Karenina* has a total of 437098 tokens. There are 12,778 unique tokens. The dictionary `word_to_int` assigns an index to each unique token. For example, the comma (,) is assigned an index of 0, and the

period (.) is assigned an index of 1. The dictionary `int_to_word` translates an index back to a token. For example, index 2 is translated back to the token "the". Index 4 is translated back to the token "and", and so on.

Finally, we convert the whole text to indexes, as follows:

```
print(text[0:20])
wordidx=[word_to_int[w] for w in text]
print([word_to_int[w] for w in text[0:20]])
```

The output is:

```
['chapter', '1', 'happy', 'families', 'are', 'all', 'alike', ';',
[208, 670, 283, 3024, 82, 31, 2461, 35, 202, 690, 365, 38, 690, 1
```

We convert all tokens in the text into the corresponding indexes and save them in a list `wordidx`. The above output shows the first 20 tokens in the text, as well as the corresponding indexes. For example, the first token in the text is *chapter*, with an index value of 208.

Exercise 8.3

Find out the index value of the token “anna” in the dictionary word_to_int.

8.3.2 Create batches of training data

Next, we create pairs of (x,y) for training purposes. Each x is a sequence with 100 indexes. There is nothing magic about the number 100, and you can easily change it to 90 or 110 and have similar results. Setting the number too high may slow down training while setting the number too small may lead to the model’s failure to capture long-range dependencies. We then slide the window right by one token and use it as the target y. Shifting the sequence by one token to the right and using it as the output during sequence generation is a common technique in training language models, including Transformers. The code block below creates the training data, as follows:

Listing 8.2 Creating training data

```
import torch
```

```
seq_len=100    #A
xys=[]
for n in range(0, len(wordidx)-seq_len-1):    #B
    x = wordidx[n:n+seq_len]    #C
    y = wordidx[n+1:n+seq_len+1]    #D
    xys.append((torch.tensor(x),(torch.tensor(y))))
```

By shifting the sequence one token to the right and using it as output, the model is trained to predict the next token given the previous tokens. For instance, if the input sequence is "how are you", then the shifted sequence would be "are you today". During training, the model learns to predict 'are' after seeing 'how', 'you' after seeing 'are', and so on. This helps the model learn the probability distribution of the next token in a sequence. You'll see this practice again and again later in this book.

We'll create batches of data for training, with 32 pairs of (x, y) in each batch:

```
from torch.utils.data import DataLoader

torch.manual_seed(42)
batch_size=32
loader = DataLoader(xys, batch_size=batch_size, shuffle=True)
```

We now have the training dataset. Next, we'll create an LSTM model and train it using the data we just processed.

8.4 Build and train the LSTM model

In this section, you'll begin by constructing an LSTM model using PyTorch's built-in LSTM layer. This model will start with a word embedding layer, which transforms each index into a dense vector of 128 dimensions. Your training data will pass through this embedding layer before being fed into the LSTM layer. This LSTM layer is designed to process elements of a sequence in a sequential manner. Following the LSTM layer, the data will proceed to a linear layer, which has an output size matching the size of your vocabulary. The outputs generated by the LSTM model are essentially logits, serving as inputs for the softmax function to compute probabilities.

Once you have built the LSTM model, the next step will involve using your training data to train this model. This training phase is crucial to refine the

model's ability to understand and generate patterns consistent with the data it has been fed.

8.4.1 Build an LSTM model

In listing 8.3, we define a `WordLSTM()` class to serve as our LSTM model to be trained to generate text in the style of *Anna Karenina*. The class is defined as follows:

Listing 8.3 Defining the WordLSTM() class

```
from torch import nn
device="cuda" if torch.cuda.is_available() else "cpu"
class WordLSTM(nn.Module):
    def __init__(self, input_size=128, n_embed=128,
                 n_layers=3, drop_prob=0.2):
        super().__init__()
        self.input_size=input_size
        self.drop_prob = drop_prob
        self.n_layers = n_layers
        self.n_embed = n_embed
        vocab_size=len(word_to_int)
        self.embedding=nn.Embedding(vocab_size,n_embed)    #A
        self.lstm = nn.LSTM(input_size=self.input_size,
            hidden_size=self.n_embed,
            num_layers=self.n_layers,
            dropout=self.drop_prob,batch_first=True)    #B
        self.fc = nn.Linear(input_size, vocab_size)

    def forward(self, x, hc):
        embed=self.embedding(x)
        x, hc = self.lstm(embed, hc)    #C
        x = self.fc(x)
        return x, hc

    def init_hidden(self, n_seqs):    #D
        weight = next(self.parameters()).data
        return (weight.new(self.n_layers,
                           n_seqs, self.n_embed).zero_(),
                weight.new(self.n_layers,
                           n_seqs, self.n_embed).zero_())
```

The `WordLSTM()` class defined above has three layers: the word embedding layer, the LSTM layer, and a final linear layer. We set the value of the

argument `n_layers` to 3, which means the LSTM layer stacks three LSTMs together to form a stacked LSTM, with the last two LSTMs taking the output from the previous LSTM as input. The `init_hidden()` method fills the hidden state with 0s when the model uses the first element in the sequence to make predictions. In each time step, the input is the current token and the previous hidden state while the output is the next token and the next hidden state.

How the torch.nn.Embedding() class works

The torch.nn.Embedding() class in PyTorch is used to create an embedding layer in a neural network. An embedding layer is a trainable lookup table that maps integer indexes to dense, continuous vector representations (embeddings).

When you create an instance of torch.nn.Embedding(), you need to specify two main parameters: num_embeddings, the size of the vocabulary (total number of unique tokens), and embedding_dim, the size of each embedding vector (the dimensionality of the output embeddings).

Internally, the class creates a matrix (or lookup table) of shape (num_embeddings, embedding_dim) where each row corresponds to the embedding vector for a particular index. Initially, these embeddings are randomly initialized but are learned and updated during training through backpropagation.

When you pass a tensor of indexes to the embedding layer (during the forward pass of the network), it looks up the corresponding embedding vectors in the lookup table and returns them. More information about the class is provided by PyTorch here: https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html.

We create an instance of the `WordLSTM()` class and use it as our LSTM model, as follows:

```
model=WordLSTM().to(device)
```

When the LSTM model is created, the weights are randomly initialized.

When we use pairs of (x, y) to train the model, LSTM learns to predict the next token based on all previous tokens in the sequence by adjusting the model parameters. As we have illustrated in figure 8.2, LSTM learns to predict the next token and the next hidden state based on the current token and the current hidden state, which is a summary of the information in all previous tokens.

We use Adam optimizer with a learning rate of 0.0001. The loss function is the cross entropy loss since this is essentially a multi-category classification problem: the model is trying to predict the next token from a dictionary with 12,778 choices:

```
lr=0.0001
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
loss_func = nn.CrossEntropyLoss()
```

Now that the LSTM model is built, we'll train the model with the batches of training data we prepared before.

8.4.2 Train the LSTM model

During each training epoch, we go through all data batches of data (x, y) in the training set. The LSTM model receives the input sequence, x, and generates a predicted output sequence, $\hat{y}$. This prediction is compared with the actual output sequence, y, to compute the cross-entropy loss since we essentially conduct a multi-category classification here. We then tweak the model's parameters to reduce this loss, as we did in Chapter 2 when classifying clothing items.

Though we could divide our data into training and validation sets, training the model until no further improvements are seen on the validation set (as we have done in Chapter 2), our primary aim here is to grasp how LSTM models function, not necessarily to achieve the best parameter tuning. Therefore, we'll train the model for 50 epochs, as follows:

Listing 8.4 Training the LSTM model to generate text

```
model.train()
```

```
for epoch in range(50):
    tloss=0
    sh,sc = model.init_hidden(batch_size)
    for i, (x,y) in enumerate(loader):    #A
        if x.shape[0]==batch_size:
            inputs, targets = x.to(device), y.to(device)
            optimizer.zero_grad()
            output, (sh,sc) = model(inputs, (sh,sc))    #B
            loss = loss_func(output.transpose(1,2),targets)    #C
            sh,sc=sh.detach(),sc.detach()
            loss.backward()
            nn.utils.clip_grad_norm_(model.parameters(), 5)
            optimizer.step()    #D
            tloss+=loss.item()
        if (i+1)%1000==0:
            print(f"at epoch {epoch} iteration {i+1}\
            average loss = {tloss/(i+1)}")
```

If you have a CUDA-enabled GPU, the above training takes about six hours. If you use CPU only, it may take a day or two, depending on your hardware. Or you can download the pre-trained weights from my website https://gattonweb.uky.edu/faculty/lium/gai/wordLSTM.zip.

Next, we save the trained model weights in the local folder:

```
import pickle

torch.save(model.state_dict(),"files/wordLSTM.pth")
with open("files/word_to_int.p","wb") as fb:
    pickle.dump(word_to_int, fb)
```

The dictionary `word_to_int` is also saved on your computer, which is a practical step ensuring that you can generate text using the trained model without needing to repeat the tokenization process.

8.5 Generate text with the trained LSTM model

Now that you have a trained LSTM model, you'll learn how to use it to generate text in this section. The goal is to see if the trained model can generate grammatically correct and coherent text by iteratively predicting the next token based on previous tokens. You'll also learn to use temperature and top-K sampling to control the creativeness of the generated text.

When generating text with the trained LSTM model, we start with a prompt as the initial input to the model. We use the trained model to predict what's the most likely next token. After appending the next token to the prompt, we feed the new sequence to the trained model to predict the next token again. We repeat this process until the sequence reaches a certain length.

8.5.1 Generate text by predicting the next token

First, we load the trained model weights and the dictionary `word_to_int` from the local folder, as follows:

```
model.load_state_dict(torch.load("files/wordLSTM.pth"))
with open("files/word_to_int.p","rb") as fb:
    word_to_int = pickle.load(fb)
int_to_word={v:k for k,v in word_to_int.items()}
```

The file `word_to_int.p` is also available in the book's GitHub repository. We switch the positions of keys and values in the dictionary `word_to_int` to create the dictionary `int_to_word`.

To generate text with the trained LSTM model, we need a prompt as the starting point of the generated text. We'll set the default prompt to "Anna and the." An easy way to determine when to stop is to limit the generated text to a certain length, say 200 tokens: once the desired length is reached, we ask the model to stop generating.

Listing 8.5 defines a `sample()` function to generate text based on a prompt:

Listing 8.5 Define a sample() function to generate text

```
import numpy as np
def sample(model, prompt, length=200):
    model.eval()
    text = prompt.lower().split(' ')
    hc = model.init_hidden(1)
    length = length - len(text)    #A
    for i in range(0, length):
        if len(text)<= seq_len:
            x = torch.tensor([[word_to_int[w] for w in text]])
        else:
```

```
            x = torch.tensor([[word_to_int[w] for w \
in text[-seq_len:]]])    #B
        inputs = x.to(device)
        output, hc = model(inputs, hc)    #C
        logits = output[0][-1]
        p = nn.functional.softmax(logits, dim=0).detach().cpu().n
        idx = np.random.choice(len(logits), p=p)    #D
        text.append(int_to_word[idx])    #E
    text=" ".join(text)
    for m in ",.:;?!$()/_&%*@'`":
        text=text.replace(f" {m}", f"{m} ")
    text=text.replace('"  ', '"')
    text=text.replace("'  ", "'")
    text=text.replace('" ', '"')
    text=text.replace("' ", "'")
    return text
```

The function `sample()` takes three arguments. The first is the trained LSTM model you will be using. The second is the starting prompt for text generation, which can be a phrase of any length, in quotes. The third parameter specifies the length of the text to be generated, measured in tokens, with a default value of 200 tokens.

Within the function, we first deduct the number of tokens in the prompt from the total desired length to determine the number of tokens that need to be generated. When generating the next token, we consider the current sequence's length. If it's under 100 tokens, we input the entire sequence into the model; if it's over 100 tokens, only the last 100 tokens of the sequence are used as input. This input is then fed into the trained LSTM model to predict the subsequent token, which we then add to the current sequence. We continue this process until the sequence reaches the desired length.

When generating the next token, the model employs the random.choice(len(logits), p=p) method from NumPy. Here, the method's first parameter indicates the range of choices, which in this case is len(logits)=12778. This signifies that the model will randomly select an integer from 0 to 12,777, with each integer corresponding to a different token in the vocabulary. The second parameter, p, is an array containing 12,778 elements where each element denotes the probability of selecting a corresponding token from the vocabulary. Tokens with a higher probability in this array are more likely to be chosen.

Let's generate a passage with the model by using "Anna and the prince" as the prompt (make sure you put a space before punctuation marks when you use your own prompt):

```
torch.manual_seed(42)
np.random.seed(42)
print(sample(model, prompt='Anna and the prince'))
```

Here I fixed the random seed number to 42 in both PyTorch and NumPy in case you want to reproduce results. The generated passage is as follows:

```
anna and the prince did not forget what he had not spoken.  when
```

You may have noticed that the text generated is entirely in lowercase. This is because, during the text processing stage, we converted all uppercase letters to lowercase to minimize the number of unique tokens.

The text generated from six hours of training is quite impressive! Most of the sentences adhere to grammatical norms. While it may not match the level of sophistication seen in text generated by advanced systems like ChatGPT, it's a significant achievement. With skills acquired in this exercise, you are ready to delve into training more advanced text generation models in later chapters.

8.5.2 Temperature and top-K sampling in text generation

The creativity of the generated text can be controlled by using techniques like temperature and top-K sampling.

Temperature adjusts the distribution of probabilities assigned to each potential token before selecting the next one. It effectively scales the logits, which are the inputs to the softmax function calculating these probabilities, by the value of the temperature. Logits are the outputs of the LSTM model prior to the application of the softmax function. When the temperature is greater than 1, the logits are scaled down. This makes the softmax probabilities more uniform, meaning the model is more likely to sample less likely tokens. When the temperature is less than 1, the logits are scaled up. This sharpens the softmax probabilities, making the model more confident in its predictions and more likely to choose the most likely token.

In the `sample()` function we just defined, we didn't adjust the logits, implying a default temperature of 1. A lower temperature (below 1, e.g., 0.8) results in fewer variations, making the model more deterministic and conservative, favoring more likely choices. Conversely, a higher temperature (above 1, e.g., 1.5) makes it more likely to choose improbable words in text generation, leading to more varied and inventive outputs. However, this could also make the text less coherent or relevant, as the model might opt for less probable words.

Top-K sampling is another method to influence the output. This approach involves selecting the next word from the top K most probable options as predicted by the model. The probability distribution is truncated to include only the top K words. With a small K value, such as 5, the model's choices are limited to a few highly probable words, resulting in more predictable and coherent outputs, but potentially less diverse and interesting. In the `sample()` function we defined earlier, we did not apply top-K sampling, so the value of K was effectively the size of the vocabulary (12,778 in our case).

Below, we introduce a new function, `generate()`, for text generation. This function is similar to the `sample()` function but includes two additional parameters: `temperature` and `top_k`, allowing for more control over the creativity and randomness of the generated text. The function `generate()` is defined as follows:

Listing 8.6 Generate text with temperature and top-K sampling

```
def generate(model, prompt , top_k=None,
             length=200, temperature=1):
    model.eval()
    text = prompt.lower().split(' ')
    hc = model.init_hidden(1)
    length = length - len(text)
    for i in range(0, length):
        if len(text)<= seq_len:
            x = torch.tensor([[word_to_int[w] for w in text]])
        else:
            x = torch.tensor([[word_to_int[w] for w in text[-seq_
        inputs = x.to(device)
        output, hc = model(inputs, hc)
        logits = output[0][-1]
        logits = logits/temperature    #A
```

```
        p = nn.functional.softmax(logits, dim=0).detach().cpu()
        if top_k is None:
            idx = np.random.choice(len(logits), p=p.numpy())
        else:
            ps, tops = p.topk(top_k)    #B
            ps=ps/ps.sum()
            idx = np.random.choice(tops, p=ps.numpy())    #C
        text.append(int_to_word[idx])

    text=" ".join(text)
    for m in ",.:;?!$()/_&%*@'`":
        text=text.replace(f" {m}", f"{m} ")
    text=text.replace('"  ', '"')
    text=text.replace("'  ", "'")
    text=text.replace('" ', '"')
    text=text.replace("' ", "'")
    return text
```

Compared to the `sample()` function, the new function `generate()` has two more optional arguments: `top_k` and `temperature`. By default, `top_k` is set to `None` and `temperature` is set to 1. Therefore, if you call the `generate()` function without specifying these two arguments, the output will be the same as what you would get from the function `sample()`.

Let's illustrate the variations in generated text by focusing on the creation of a single token. For this purpose, we'll use “I ' m not going to see” as the prompt (note the space before the apostrophe, as we previously have done in the chapter). We call the generate() function ten times, setting its length argument to be one more than the prompt's length. This approach ensures that the function appends only one extra token to the prompt, as shown below:

```
prompt="I ' m not going to see"
torch.manual_seed(42)
np.random.seed(42)
for _ in range(10):
    print(generate(model, prompt, top_k=None,
         length=len(prompt.split(" "))+1, temperature=1))
```

The output is as follows:

```
i'm not going to see you
i'm not going to see those
i'm not going to see me
i'm not going to see you
```

```
i'm not going to see her
i'm not going to see her
i'm not going to see the
i'm not going to see my
i'm not going to see you
i'm not going to see me
```

With the default setting of top_k=None and temperature=1, there is some degree of repetition in the output. For example, the word “you” was repeated three times. There are a total of six unique tokens.

However, the functionality of `generate()` expands when you adjust these two arguments. For instance, setting a low temperature, like 0.5, and a small `top_k` value, such as 3, results in generated text that is more predictable and less creative.

Let's repeat the single token example. This time, we set the temperature to 0.5 and top_k value to 3, as shown below:

```
prompt="I ' m not going to see"
torch.manual_seed(42)
np.random.seed(42)
for _ in range(10):
    print(generate(model, prompt, top_k=3,
         length=len(prompt.split(" "))+1, temperature=0.5))
```

The output is as follows:

```
i'm not going to see you
i'm not going to see the
i'm not going to see her
i'm not going to see you
i'm not going to see you
i'm not going to see you
i'm not going to see you
i'm not going to see her
i'm not going to see you
i'm not going to see her
```

The output has fewer variations: there are only three unique tokens from ten attempts, “you”, “the”, and “her.”

Let’s see this in action below by using “Anna and the prince” as our starting

prompt when we set the temperature to 0.5 and top_k value to 3:

```
torch.manual_seed(42)
np.random.seed(42)
print(generate(model, prompt='Anna and the prince',
               top_k=3,
               temperature=0.5))
```

The output is the following:

```
anna and the prince had no milk.  but,  "answered levin,  and he
```

Exercise 8.4

Generate text by setting temperature to 0.6 and `top_k` to 10 and using “Anna and the nurse” as the starting prompt. Set the random seed number to 0 in both PyTorch and NumPy.

Conversely, opting for a higher `temperature` value, like 1.5, coupled with a higher `top_k` value, for instance, `None` (enabling selection from the entire pool of 12,778 tokens), leads to outputs that are more creative and less foreseeable. This is demonstrated below, in the single token example. This time, we set the temperature to 2 and top_k value to None, as shown below:

```
prompt="I ' m not going to see"
torch.manual_seed(42)
np.random.seed(42)
for _ in range(10):
    print(generate(model, prompt, top_k=None,
         length=len(prompt.split(" "))+1, temperature=2))
```

The output is as follows:

```
i'm not going to see them
i'm not going to see scarlatina
i'm not going to see behind
i'm not going to see us
i'm not going to see it
i'm not going to see it
i'm not going to see a
i'm not going to see misery
i'm not going to see another
i'm not going to see seryozha
```

The output has almost no repetition: there are nine unique tokens from ten attempts, only the word “it” was repeated.

Let’s again use "Anna and the prince" as the initial prompt but set the temperature to 2 and top_k value to None and see what happens:

```
torch.manual_seed(42)
np.random.seed(42)
print(generate(model, prompt='Anna and the prince',
               top_k=None,
               temperature=2))
```

The generated text is below:

```
anna and the prince took sheaves covered suddenly people.  "pyotr
```

The output generated above is not repetitive, although it lacks coherence in many places.

Exercise 8.5

Generate text by setting temperature to 2 and `top_k` to 10000 and using “Anna and the nurse” as the starting prompt. Set the random seed number to 0 in both PyTorch and NumPy.

In this chapter, you have acquired foundational skills in NLP, including word-level tokenization, word embedding, and sequence prediction. Through these exercises, you've learned to construct a language model based on word-level tokenization and have trained it using LSTM for text generation. Moving forward, the next few chapters will introduce you to training Transformers, the type of models used in systems like ChatGPT. This will provide you with a more in-depth exploration of advanced text generation techniques.

8.6 Summary

- Recurrent neural networks (RNNs) are a specialized form of artificial neural network designed to recognize patterns in sequences of data, such as text, music, or stock prices. Unlike traditional neural networks, which

process inputs independently, RNNs have loops in them, allowing information to persist. Long short-term memory (LSTM) networks are improved versions of RNNs.

- There are three approaches to tokenization. The first is character tokenization, where the text is divided into its constituent characters. The second approach is word tokenization, where the text is split into individual words. The third approach is subword tokenization, which breaks words into smaller, meaningful components called subwords.
- Word embedding is a method that transforms words into compact vector representations, capturing their semantic information and interrelationships. This technique is vital in NLP, especially since deep neural networks, including models like LSTM and Transformers, require numerical input.
- Temperature is a parameter to influence the behavior of text generation models. It controls the randomness of the predictions by scaling the logits (the inputs to the softmax function for probability calculation) before applying softmax. Low temperature makes the model more conservative in its predictions but also more repetitive. At higher temperatures, the model becomes less repetitive and more innovative increasing the diversity of the generated text.
- Top-K sampling is another way to influence the behavior of text generation models. It involves selecting the next word from the K most likely candidates, as determined by the model. The probability distribution is truncated to keep only the top K words. Small values of K make the output more predictable and coherent but potentially less diverse and interesting.

[1] Sepp Hochreiter and Jurgen Schmidhuber, 1997, Long Short-Term Memory. Neural Computation 9(8): 1735-1780.

9 A Line-by-Line Implementation of Attention and Transformer

This chapter covers

- Architecture and functionalities of encoders and decoders in Transformers
- How the attention mechanism uses query, key, and value to assign weights to elements in a sequence
- Building and training a Transformer from scratch to translate English to French
- Using the trained Transformer to translate an English phrase into French

Transformers are advanced deep learning models that excel in handling sequence-to-sequence prediction challenges, outperforming older models like recurrent neural networks (RNNs) and convolutional neural networks (CNNs). Their strength lies in effectively understanding the relationships between elements in input and output sequences over long distances, such as two words far apart in the text. Unlike RNNs, Transformers are capable of parallel training, significantly cutting down training times and enabling the handling of vast datasets. This transformative architecture has been pivotal in the development of large language models (LLMs) like ChatGPT, BERT, and T5, marking a significant milestone in AI progress.

Prior to the introduction of Transformers in the groundbreaking 2017 paper *Attention Is All You Need* by a group of Google researchers,[1] natural language processing (NLP) and similar tasks primarily relied on RNNs, including long short-term memory (LSTM) models. RNNs, however, process information sequentially, limiting their speed due to the inability to train in parallel and struggling with maintaining information about earlier parts of a sequence, thus failing to capture long-term dependencies.

The revolutionary aspect of the Transformer architecture is its attention mechanism. This mechanism assesses the relationship between words in a

sequence by assigning weights, determining the degree of relatedness in meaning among words based on the training data. This enables models like ChatGPT to comprehend relationships between words, thus understanding human language more effectively. The non-sequential processing of inputs allows for parallel training, reducing training time and facilitating the use of large datasets, thereby powering the rise of knowledgeable LLMs and the current surge in AI advancements.

In this chapter, we will delve into building a Transformer from the ground up, based on the paper *Attention Is All You Need*, to translate English into French. We'll explore the inner workings of the self-attention mechanism, including the roles of query, key, and value vectors, and the computation of scaled dot product attention (SDPA). We'll construct an encoder layer by integrating layer normalization and residual connection into a multi-head attention layer and combining it with a feed-forward layer, and then stack six of these encoder layers to form the encoder. Similarly, we'll develop a decoder in the Transformer and learn to generate French translations one token at a time, based on previous tokens in the French phrase and the encoder's output.

Finally, we'll train our model on a dataset containing over 47,000 English-to-French translations. The trained model can translate common English phrases accurately as if you are using Google Translate for the task.

9.1 Introduction to attention and Transformers

To grasp the concept of Transformers in machine learning, it's essential to first understand the attention mechanism. This mechanism allows Transformers to recognize long-range dependencies between sequence elements, a feature that sets them apart from earlier sequence prediction models like RNNs. With this mechanism, Transformers can simultaneously focus on every element in a sequence, comprehending the context of each word.

Consider the word "bank" to illustrate how the attention mechanism interprets words based on context. In the sentence "I went fishing by the river yesterday, remaining near the bank the whole afternoon." the word "bank" is linked to "fishing" because it refers to the area beside a river. Here, a

Transformer understands "bank" as part of the river's terrain.

Conversely, in "Kate went to the bank after work yesterday and deposited a check there," "bank" is connected to "check," leading the Transformer to identify "bank" as a financial institution. This example showcases how Transformers discern word meanings based on their surrounding context.

In this section, you'll dive deeper into the attention mechanism, exploring how it works. This process is crucial for determining the importance, or weights, of various words within a sentence. After that, we'll examine the structure of different Transformer models, including one that can translate English to French.

9.1.1 The attention mechanism

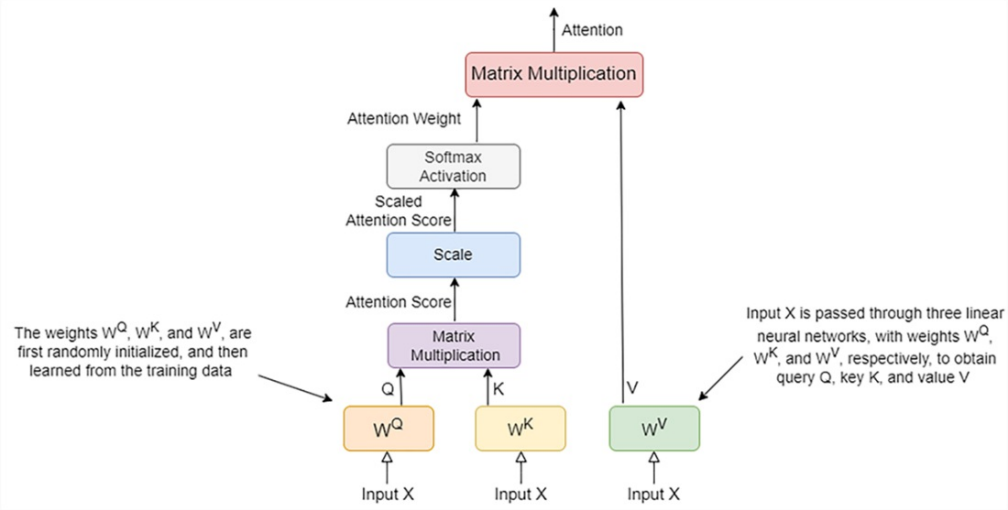
The attention mechanism is a method used to determine the interconnections between elements in a sequence. It calculates scores to indicate how one element relates to others in the sequence, with higher scores denoting a stronger relationship. In NLP, this mechanism is instrumental in linking words within a sentence meaningfully. This chapter will guide you through implementing the attention mechanism for translating English into French.

We'll construct a Transformer composed of an encoder and a decoder for that purpose. The encoder transforms an English sentence, such as "How are you?", into vector representations that encapsulate its meaning. The decoder then uses these vector representations to generate the French translation.

To transform the phrase "How are you?" into vector representations, the model initially breaks it down into tokens [how, are, you, ?], a process similar to what you have done in Chapter 8. These tokens are each represented by a 256-dimensional vector known as word embeddings, which capture the meaning of each token. The encoder also employs positional encoding, a method to determine the positions of tokens in the sequence. This positional encoding is added to the word embeddings to create input embeddings, which are then used in calculating self-attention. The input embedding for "how are you?" forms a tensor with dimensions (4, 256), where 4 represents the number of tokens, and 256 is the dimensionality of each embedding.

While there are different ways to calculate attention, we'll use the most common method, scaled dot product attention (SDPA). This mechanism is also called self-attention because the algorithm calculates how a word attends to all words in the sequence, including the word itself. Figure 9.1 provides a diagram of how to calculate SDPA.

Figure 9.1 A diagram of the self-attention mechanism. To calculate attention, the input embedding X is first passed through three neural layers with weights, W^Q, W^K, and W^V, respectively. The outputs are query Q, key K, and value V. The scaled attention score is the product of Q and K divided by the square root of the dimension of K, d_k. We apply the softmax function on the scaled attention score to obtain the attention weight. The attention is the product of the attention weight and value V.

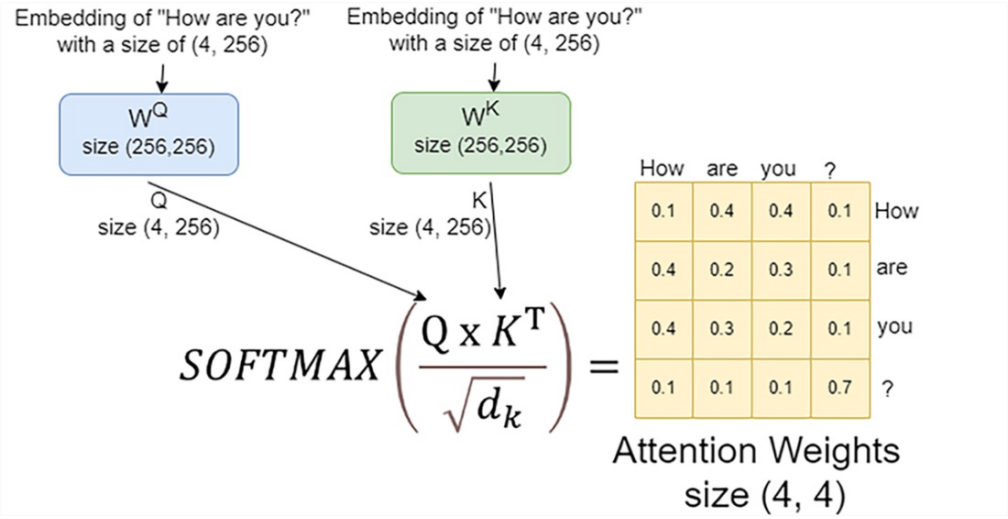


The utilization of query, key, and value in calculating attention is inspired by retrieval systems. Consider visiting a public library to find a book. If you search for "machine learning in finance" in the library's search engine, this phrase becomes your query. The book titles and descriptions in the library serve as the keys. Based on the similarity between your query and these keys, the library's retrieval system suggests a list of books (values). Books containing "machine learning," "finance," or both in their titles or descriptions are likely to rank higher. In contrast, books unrelated to these terms will have a lower matching score and thus are less likely to be recommended.

To calculate SDPA, the input embedding X is processed through three distinct neural network layers. The corresponding weights for these layers are

W^Q, W^K, and W^V, each having a dimension of 256 by 256. These weights are learned from data during the training phase. Thus, we can calculate query Q, key K, and value V as $Q=X*W^Q$, $K=X*Q^K$, and $V=X*W^V$. The dimensions of Q, K, and V match those of the input embedding X, which are 4 by 256.

Figure 9.2 Steps to calculate attention weights. The input embedding is passed through two neural networks to obtain query Q and key K. The scaled attention scores are calculated as the dot product of Q and K divided by the square root of the dimension of K. Finally, we apply the softmax function on the scaled attention scores to obtain attention weights, which demonstrate how each element is related to all other elements in the sequence.



Similar to the retrieval system example we mentioned above, in the attention mechanism, we assess the similarities between the query and key vectors using the SDPA approach. SDPA involves calculating the dot product of the query (Q) and key (K) vectors. A high dot product indicates a strong similarity between the two vectors, and vice versa. For instance, in the sentence "How are you?", the scaled attention score is computed as follows:

(Equation 9.1)

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

where d_k represents the dimension of the key vector K, which in our case is

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

The next step is to apply the softmax function to these attention scores, converting them into attention weights. This ensures that the total attention a word gives to all words in the sentence sums to 100%.

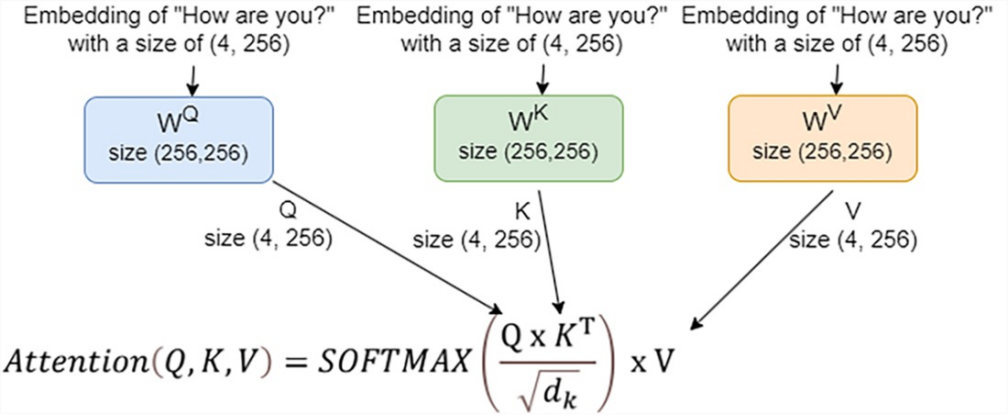
Figure 9.2 shows how this is done. For the sentence “How are you?”, the attention weights form a four-by-four matrix, which shows how each token in ['How', 'are', 'you', '?'] is related to all other tokens (including itself). The numbers in figure 9.2 are made-up numbers to illustrate the point. For example, the first row in the attention weights shows that the token “How” gives 10% of its attention to itself and 40%, 40%, and 10% to the other three tokens, respectively.

The final attention is then calculated as the dot product of these attention weights and the value vector V:

(Equation 9.2)

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

Figure 9.3 Use attention weights and the value vector to calculate the attention vector. The input embedding is passed through a neural network to obtain value V. The final attention is the dot product of the attention weights that we calculated earlier and the value vector V.



This output also maintains a dimension of 4 by 256, consistent with our input dimensions.

To summarize, the process begins with the input embedding X of the sentence “how are you?”, which has a dimension of 4 by 256. This embedding captures the meanings of the four individual tokens but lacks contextualized understanding. The attention mechanism ends with the output `attention(Q, K, V)`, which maintains the same dimension of 4 by 256. This output can be viewed as a contextually-enriched combination of the original four tokens. The weighting of the original tokens varies based on the contextual relevance of each token, granting more significance to words that are more important within the sentence’s context. Through this procedure, the attention mechanism transforms vectors representing isolated tokens into vectors imbued with contextualized meanings, thereby extracting a richer, more nuanced understanding from the sentence.

The magic power of the attention mechanism

The revolutionary aspect of Transformers lies in their utilization of the attention mechanism. This mechanism operates by taking the embedding of a sequence, such as an English sentence, as its input. Initially, this input captures the meanings of individual tokens but does not encompass a contextualized understanding. The attention mechanism first passes the input through three neural layers with weights W^Q, W^K, and W^V to obtain query Q, key K, and value V. The output is the attention as calculated by the formula in equation 9.2.

Viewed as a contextually enriched combination of the original tokens, this output assigns varying weights to each token based on their contextual relevance. This differential weighting amplifies the significance of words that hold more importance within the context of the sentence. Consequently, the attention mechanism transforms vectors that initially represent discrete tokens into vectors enriched with contextual meanings. This process effectively extracts a deeper and more nuanced understanding of the sentence, showcasing the extraordinary capability of the attention mechanism. And that is the magic power of the attention mechanism!

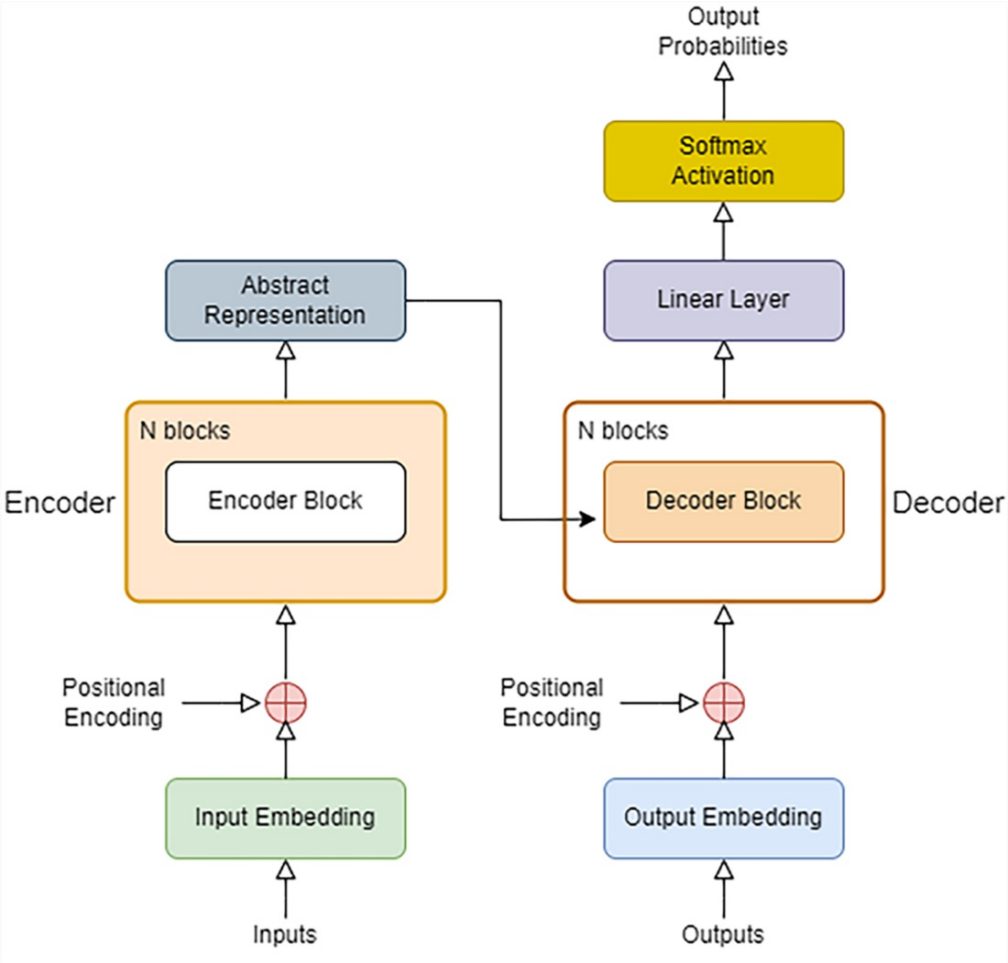
Further, instead of using one set of query, key, and value vectors, Transformer models use a concept called multi-head attention. For example, the 256-dimensional query, key, and value vectors can be split into say, 8, heads, and each head has a set of query, key, and value vectors with dimensions of 32 (because 256/8=32). Each head pays attention to different parts or aspects of the input, enabling the model to capture a broader range of information and form a more detailed and contextual understanding of the input data. Multi-head attention is especially useful when a word has multiple meanings in a sentence, such as in a pun. Let's continue the "bank" example we mentioned earlier. Consider the pun joke, "Why is the river so rich? Because it has two banks." In the project of translating English to French, you'll implement first-hand splitting Q, K, and V into multiple heads to calculate attention in each head before concatenating them back into one single attention vector.

9.1.2 The Transformer architecture

The concept of the attention mechanism was introduced by Bahdanau, Cho, and Bengio in 2014.[2] It became widely used after the groundbreaking paper *Attention Is All You Need*, which focused on creating a model for machine language translation. The architecture of this model, known as the Transformer, is depicted in Figure 9.4. It features an encoder-decoder structure that relies heavily on the attention mechanism. In this chapter, you'll build this model from scratch, coding it line by line, with the objective of training it to translate from English to French.

Figure 9.4 The Transformer architecture. The encoder in the Transformer (left side of the

diagram), which consists of N identical encoder layers, learns the meaning of the input sequence and converts it into a vector that represents this meaning. It then passes this vector to the decoder (right side of the diagram), which consists of N identical decoder layers. The decoder constructs the output (e.g., the French translation of an English phrase) by predicting one token at a time, based on previous tokens in the sequence and the vector representation from the encoder. The generator on the top right is the head attached to the output from the decoder so that the output is the probability distribution over all tokens in the target language (e.g., the French vocabulary).



The Transformer's encoder transforms an English sentence like "I don't speak French" into vector representations that encapsulate its meaning. The Transformer's decoder then processes them to produce the French translation "Je ne parle pas français". The encoder's role is to capture the essence of the original English sentence. For instance, if the encoder is effective, it should translate both "I don't speak French" and "I do not speak French" into similar vector representations. Consequently, the decoder will interpret these vectors and generate similar translations. Interestingly, when using ChatGPT, these two English phrases indeed result in the same French translation.

The encoder in the Transformer approaches the task by first tokenizing both the English and French sentences. This is similar to the process described in Chapter 8, but with a key difference: it employs subword tokenization. Subword tokenization is a technique used in NLP to break words into smaller components, or subwords, allowing for more efficient and nuanced processing. For example, the English phrase "I do not speak French" is divided into six tokens: (i, do, not, speak, fr, ench). Similarly, its French counterpart "Je ne parle pas français" is tokenized into six parts: (je, ne, parle, pas, franc, ais). This method of tokenization enhances the Transformer's ability to handle language variations and complexities.

Deep learning models, including Transformers, can't directly process text, so tokens are indexed using integers before being fed to the model. These tokens are typically first represented using one-hot encoding, as we discussed in Chapter 8. We then pass them through a word embedding layer to compress them into vectors with continuous values of a much smaller size, such as a length of 256. Thus, after applying word embedding, the sentence "I do not speak French" is represented by a 6x256 matrix.

Transformers process input data such as sentences in parallel, unlike Recurrent Neural Networks (RNNs) which handle data sequentially. This parallelism enhances their efficiency but doesn't inherently allow them to recognize the sequence order of the input. To address this, Transformers add positional encodings to the input embeddings. These positional encodings are unique vectors assigned to each position in the input sequence and align in dimension with the input embeddings. The vector values are determined by a specific positional function, particularly involving sine and cosine functions of varying frequencies, defined as:

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

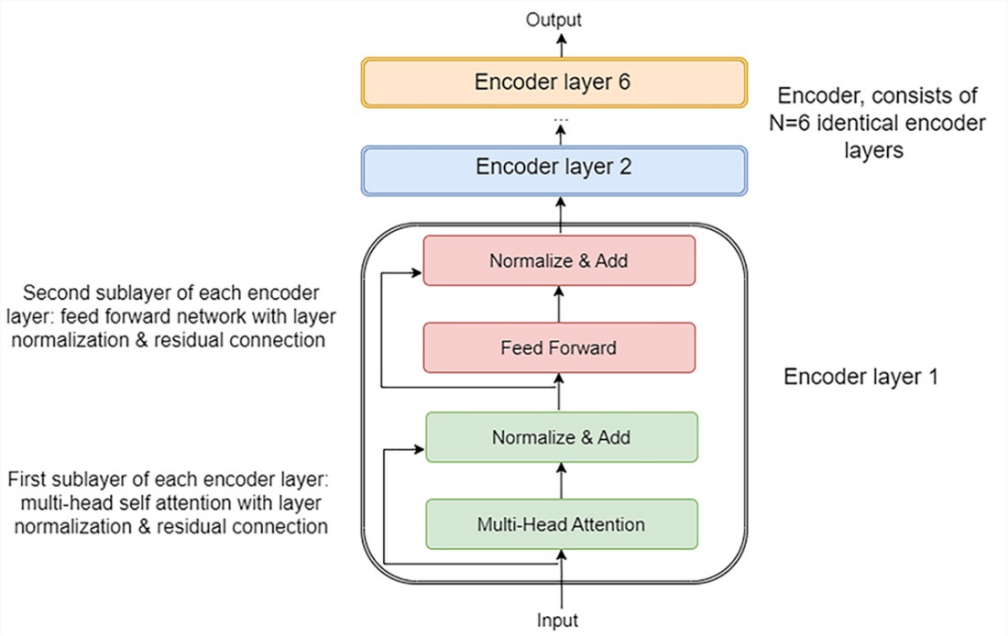
$$\text{PositionalEncoding}(\text{pos}, 2\text{i}+1) = \cos\left(\frac{\text{pos}}{n^{2\text{i}/\text{d}}}\right)$$

In the above equations, vectors are calculated using the sine function for even indexes and the cosine function for odd indexes. The two parameters *pos* and

i represent the position of a token within the sequence and the index within the vector, respectively. As an illustration, consider the positional encoding for the phrase "I do not speak French." This is depicted as a 6x256 matrix, the same size as the word embedding for the sentence. Here, *pos* ranges from 0 to 5, and the indexes *2i* and *2i+1* collectively span 256 distinct values (from 0 to 255). Numerous approaches exist for modeling positional encoding, and this is how Transformers incorporate positional information about elements in a sequence. A beneficial aspect of this positional encoding approach is that all values are constrained within the range of -1 to 1.

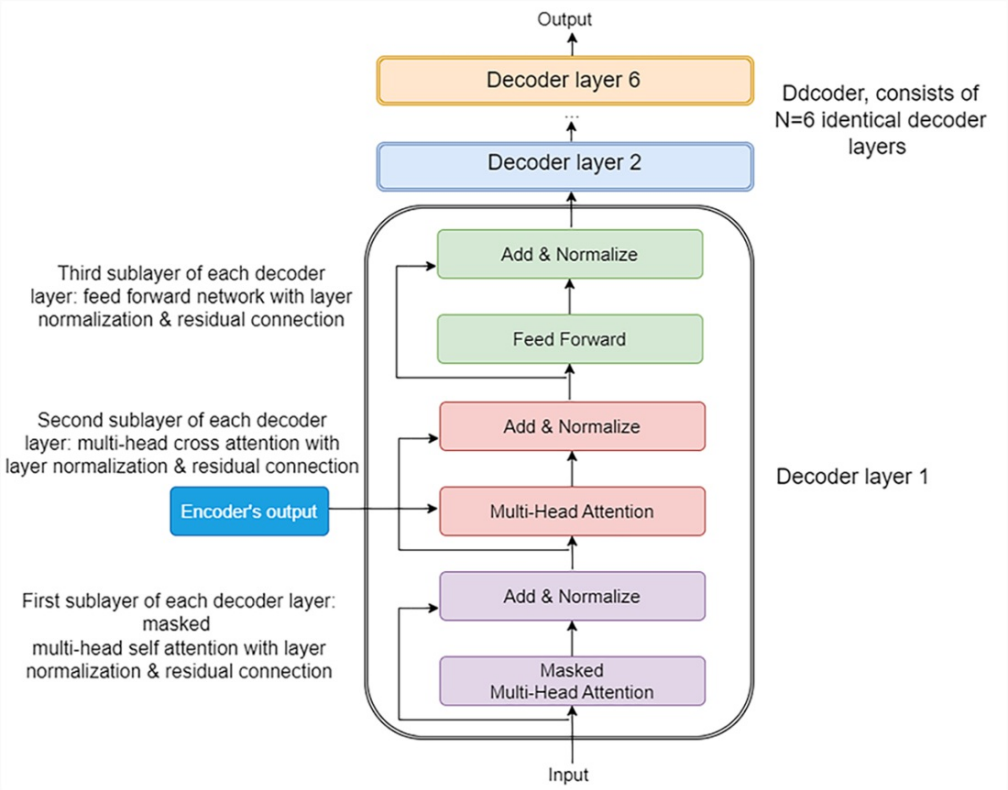
It's important to note that each token position is uniquely identified by a 256-dimensional vector, and these vector values remain constant throughout training. Before being input to the attention layers, these positional encodings are added to the word embeddings of the sequence. In the example of the sentence "I do not speak French.", the encoder generates both word embedding and positional encoding, each having dimensions of 6x256, before combining them into a single 6x256-dimensional representation. Subsequently, the encoder applies the attention mechanism to refine this embedding into more sophisticated vector representations that capture the overall meaning of the phrase, before passing them to the decoder.

Figure 9.5 The structure of the encoder in the Transformer. The encoder consists of N identical encoder layers. Each encoder layer contains two sublayers. The first sublayer is a multi-head self-attention layer and the second is a feed forward network. Each sublayer uses residual connection and layer normalization.



The Transformer's encoder, as depicted in figure 9.5, is made up of six identical layers (N=6). Each of these layers comprises two distinct sub-layers. The first sub-layer is a multi-head self-attention layer, similar to what was discussed earlier. The second sub-layer is a basic, position-wise, fully connected feed-forward network. This network treats each position in the sequence independently rather than as sequential elements. In the model's architecture, each sublayer incorporates a layer normalization and the residual connection. Layer normalization normalizes observations to have zero mean and unit standard deviation. Such normalization helps stabilize the training process. After the normalization layer, we perform the residual connection. This means the input to each sublayer is added to its output, enhancing the flow of information through the network.

Figure 9.6 The structure of the decoder in the Transformer. The decoder consists of N identical decoder layers. Each decoder layer contains three sublayers. The first sublayer is a masked multi-head self-attention layer. The second is a multi-head attention layer to calculate the cross attention between the output from the first sublayer and the output from the encoder stack. The third sublayer is a feed forward network. Each sublayer uses residual connection and layer normalization.



The decoder of the Transformer model, as seen in figure 9.6, is comprised of six identical decoder layers (N=6). Each of these decoder layers features three sublayers: a multi-head self-attention sublayer, a sub-layer that performs multi-head cross-attention on the outputs from the encoder, and a position-wise, fully connected feed-forward sublayer. Note that the input to each sublayer is the output from the previous sublayer. Further, the second sublayer in the decoder layer also takes the output from the encoder as input. This design is crucial for integrating information from the encoder: this is how the decode generates translations based on the output from the encoder.

A key aspect of the decoder's self-attention sub-layer is the masking mechanism. This mask prevents the model from accessing future positions in the sequence, ensuring that predictions for a particular position can only depend on previously known elements. This sequential dependency is vital for tasks like language translation or text generation. This is similar to what you have seen in Chapter 8. For example, in the sentence “a frog has four legs,” the masking mechanism forces the model to use only “a frog has” to predict the word “four” by hiding the words “four legs” in the sentence.

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

The decoder operates in an autoregressive manner, generating the output sequence one token at a time. At the first time step, it starts with the “BOS” token, which indicates the beginning of a sentence. Using this start token as its initial input, the decoder examines vector representations of the English phrase "I do not speak French" and attempts to predict the first token following “BOS”. Suppose the decoder's first prediction is "Je." In the next time step, it then uses the sequence "BOS Je" as its new input to predict the following token. This process continues iteratively, with the decoder adding each newly predicted token to its input sequence for the subsequent prediction.

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

9.1.3 Different types of Transformers

There are three types of Transformers: encoder-only Transformers, decoder-only Transformers, and encoder-decoder Transformers. We are using an encoder-decoder Transformer in this chapter, but you’ll get a chance to explore firsthand encoder-only and decoder-only Transformers in the next couple of chapters.

An encoder-only Transformer consists of N identical encoder layers as shown on the left side of figure 9.4 and is capable of converting a sequence into

abstract continuous vector representations. For example, BERT is an encoder-only Transformer that contains 12 encoder layers. An encoder-only Transformer can be used for text classification, for example. If two sentences have similar vector representations, we can classify the two sentences into one category. On the other hand, if two sequences have very different vector representations, we can put them in different categories.

A decoder-only Transformer also consists of N identical layers, and each layer is a decoder layer as shown on the right side of Figure 9.4. For example, ChatGPT is a decoder-only Transformer that contains many decoder layers. The decoder-only Transformer can generate text based on a prompt, for example. It extracts the semantic meaning of the words in the prompt and predicts the most likely next word. It then adds the word to the end of the prompt and repeats the process until the text reaches a certain length.

The English-French translation Transformer we discussed above is an example of an encoder-decoder Transformer. Encoder-decoder Transformers are needed for handling complicated models, including multi-modal models for text-to-image tasks or speech recognition tasks. Encoder-decoder Transformers combine the strengths of both encoders and decoders. Encoders are efficient in processing and understanding input data, while decoders excel in generating output. This combination allows the model to effectively understand complex inputs (like text or speech) and generate intricate outputs (like images or transcribed text).

9.2 Tokenization, word embedding, and positional encoding

To better illustrate self-attention and Transformers, we'll use English-to-French translation as our case study. By exploring the process of training a model for converting English sentences into French, you will gain a comprehensive understanding of the Transformer's architecture and the functioning of the attention mechanism.

Picture yourself having amassed a collection of over 47,000 English-to-French translation pairs. Your objective is to develop a machine learning model and train it using this dataset. This section will walk you through the

initial three phases of the project: tokenization, word embedding, and positional encoding.

9.2.1 Tokenize English and French phrases

Go to https://gattonweb.uky.edu/faculty/lium/gai/en2fr.zip to download the zip file that contains the English-to-French translations I collected from various sources. Unzip the file and place en2fr.csv in the folder /files/ on your computer.

We'll load the data and print out an English phrase, along with its French translation, as follows:

```
import pandas as pd

df=pd.read_csv("files/en2fr.csv")    #A
num_examples=len(df)    #B
print(f"there are {num_examples} examples in the training data")
print(df.iloc[30856]["en"])    #C
print(df.iloc[30856]["fr"])    #D
```

The output from the above code snippet is:

```
there are 47173 examples in the training data
How are you?
Comment êtes-vous?
```

There are a total of 47,173 pairs of English-to-French translations in the training data. We have printed out the English phrase "How are you?" and the corresponding French translation "Comment êtes-vous?" as an example.

Next, we'll tokenize both the English and the French phrases in the dataset. We'll use the pre-trained XLM model from Hugging Face as the tokenizer because it excels at handling multiple languages. The model can tokenize both English and French phrases.

Listing 9.1 A pre-trained tokenizer

```
from transformers import XLMTokenizer    #A
```

```
tokenizer = XLMTokenizer.from_pretrained("xlm-clm-enfr-1024")

tokenized_en=tokenizer.tokenize("I don't speak French.")    #B
print(tokenized_en)
tokenized_fr=tokenizer.tokenize("Je ne parle pas français.")    #
print(tokenized_fr)
print(tokenizer.tokenize("How are you?"))
print(tokenizer.tokenize("Comment êtes-vous?"))
```

The output from code listing 9.1 is as follows:

```
['i</w>', 'don</w>', "'t</w>", 'speak</w>', 'fr', 'ench</w>', '.<
['je</w>', 'ne</w>', 'parle</w>', 'pas</w>', 'franc', 'ais</w>',
['how</w>', 'are</w>', 'you</w>', '?</w>']
['comment</w>', 'et', 'es-vous</w>', '?</w>']
```

In the above code block, we use a pre-trained tokenizer from the XLM model to divide the English sentence "I don't speak French." into distinct tokens. In NLP, tokens represent the basic semantic elements, encompassing individual words or fragments of words. Earlier, in Chapter 8, you developed a custom word-level tokenizer. However, this chapter introduces the use of a more efficient pre-trained subword tokenizer, surpassing the word-level tokenizer in effectiveness. The sentence "I don't speak French." is thus tokenized into ['i', 'don', "'t", 'speak', 'fr', 'ench', '.']. Similarly, the French sentence "Je ne parle pas français." is split into six tokens: ['je', 'ne', 'parle', 'pas', 'franc', 'ais', '.'], using the same pre-trained tokenizer. We have also tokenized the English phrase “How are you?” and its French translation. The results are shown in the last two lines of the above output.

NOTE

You may have noticed that the XLM model uses '</w>' as a token separator, except in cases where two tokens are part of the same word. Subword tokenization typically results in each token being either a complete word or a punctuation mark, but there are occasions when a word is divided into syllables. For example, the word 'French' is divided into 'fr' and 'ench'. It's noteworthy that the model doesn't insert '</w>' between 'fr' and 'ench', as these syllables jointly constitute the word 'French'.

Deep learning models such as Transformers cannot process raw text directly, hence we need to convert text into numerical representations before feeding them to the models. For that purpose, we create a dictionary to map all English tokens to integers:

Listing 9.2 Mapping English tokens to indexes

```
from collections import Counter

en=df["en"].tolist()    #A

en_tokens=[["BOS"]+tokenizer.tokenize(x)+["EOS"] for x in en]
PAD=0
UNK=1
word_count=Counter()
for sentence in en_tokens:
    for word in sentence:
        word_count[word]+=1
frequency=word_count.most_common(50000)    #C
total_en_words=len(frequency)+2
en_word_dict={w[0]:idx+2 for idx,w in enumerate(frequency)}    #D
en_word_dict["PAD"]=PAD
en_word_dict["UNK"]=UNK
en_idx_dict={v:k for k,v in en_word_dict.items()}    #E
```

We incorporate the tokens "BOS" (beginning of the sentence) and "EOS" (end of sentence) at the start and end of each phrase, respectively. In our setup, there is a dictionary named en_word_dict, which assigns each token a unique integer value. Further, the "PAD" token, used for padding, is allocated the integer 0, while the "UNK" token, representing unknown tokens, is given the integer 1. Additionally, we have established a reverse dictionary, en_idx_dict, which maps integers (indexes) back to their corresponding tokens. This reverse mapping is essential for converting a sequence of integers back into a sequence of tokens, enabling us to reconstruct the original English phrase.

Using the dictionary en_word_dict, we can transform the English sentence "I don't speak French." into its numerical representation. This process involves looking up each token in the dictionary to find its corresponding integer value. For instance:

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

The above lines of code produce the following output:

```
[15, 100, 38, 377, 476, 574, 5]
```

This means that the English sentence "I don't speak French." is now represented by a sequence of integers [15, 100, 38, 377, 476, 574, 5].

We can also revert the numerical representations into tokens using the dictionary en_idx_dict. This process involves mapping each integer in the numerical sequence back to its corresponding token as defined in the dictionary. Here's how it is done:

```
entokens=[en_idx_dict.get(i,"UNK") for i in enidx]    #A
print(entokens)
en_phrase="".join(entokens)    #B
en_phrase=en_phrase.replace("</w>"," ")    #C
for x in '''?:;.,'("-!&)%''':
    en_phrase=en_phrase.replace(f" {x}",f"{x}")    #D
print(en_phrase)
```

The output of the above code snippet is

```
['i</w>', 'don</w>', "'t</w>", 'speak</w>', 'fr', 'ench</w>', '.<
i don't speak french.
```

The dictionary en_idx_dict is used to translate numbers back into their original tokens. Following this, these tokens are transformed into the complete English phrase. This is done by first joining the tokens into a single string and then substituting the separator "</w>" with a space. At the end, we also remove the space before punctuation marks so the result appears as “i don't speak french.” instead of “i don 't speak french .” Notice that the restored English phrase has all lowercase letters in it because the pre-trained tokenizer automatically converts uppercase letters into lowercase to reduce the number of unique tokens. As you’ll see in the next chapter, some models such as GPT2 or ChatGPT don’t do this; hence they have a larger vocabulary.

Exercise 9.1

In listing 9.1, we have split the sentence “How are you?” into tokens ['how</w>', 'are</w>', 'you</w>', '?</w>']. Follow the steps in this subsection to (i) convert the tokens into indexes using the dictionary en_word_dict; (ii) convert the indexes back to tokens using the dictionary en_idx_dict; (iii) restore the English sentence by joining the tokens into a string, changing the separator '</w>' to a space, and removing the space before punctuation marks.

We can apply the same steps to French phrases to map tokens to indexes and vice versa:

Listing 9.3 Mapping French tokens to indexes

```
fr=df["fr"].tolist()
fr_tokens=[["BOS"]+tokenizer.tokenize(x)+["EOS"] for x in fr]
word_count=Counter()
for sentence in fr_tokens:
    for word in sentence:
        word_count[word]+=1
frequency=word_count.most_common(50000)    #B
total_fr_words=len(frequency)+2
fr_word_dict={w[0]:idx+2 for idx,w in enumerate(frequency)}    #C
fr_word_dict["PAD"]=PAD
fr_word_dict["UNK"]=UNK
fr_idx_dict={v:k for k,v in fr_word_dict.items()}    #D
```

The dictionary fr_word_dict assigns an integer to each French token, while en_idx_dict maps these integers back to their corresponding French tokens. Next, we'll demonstrate how to transform the French phrase "Je ne parle pas français." into its numerical representation:

```
fridx=[fr_word_dict.get(i,UNK) for i in tokenized_fr]
print(fridx)
```

The output from the above code snippet is as follows:

```
[28, 40, 231, 32, 726, 370, 4]
```

The tokens for the French phrase "Je ne parle pas français." are converted into a sequence of integers as shown above.

We can transform the numerical representations back into French tokens using the dictionary fr_idx_dict. This involves translating each number in the sequence back to its respective French token as outlined in the dictionary. Once the tokens are retrieved, they can be joined together to reconstruct the original French phrase. Here's how it's done:

```
frtokens=[fr_idx_dict.get(i,"UNK") for i in fridx]
print(frtokens)
fr_phrase="".join(frtokens)
fr_phrase=fr_phrase.replace("</w>"," ")
for x in '''?:;.,'("-!&)%''':
    fr_phrase=fr_phrase.replace(f" {x}",f"{x}")
print(fr_phrase)
```

The output from the above code block is:

```
['je</w>', 'ne</w>', 'parle</w>', 'pas</w>', 'franc', 'ais</w>',
je ne parle pas francais.
```

It's important to recognize that the restored French phrase doesn’t exactly match its original form. This discrepancy is due to the tokenization process, which transforms all uppercase letters into lowercase and eliminates accent marks in French.

Exercise 9.2

In listing 9.1, we have split the sentence “Comment êtes-vous?” into tokens ['comment</w>', 'et', 'es-vous</w>', '?</w>']. Follow the steps in this subsection to (i) convert the tokens into indexes using the dictionary fr_word_dict; (ii) convert the indexes back to tokens using the dictionary fr_idx_dict; (iii) restore the French phrase by joining the tokens into a string, changing the separator '</w>' to a space, and removing the space before punctuation marks.

We save the four dictionaries in the folder /files/ on your computer so that

you can load them up and start translating later without worrying about first mapping tokens to indexes and vice versa:

```
import pickle

with open("files/dict.p","wb") as fb:
    pickle.dump((en_word_dict,en_idx_dict,
                 fr_word_dict,fr_idx_dict),fb)
```

The four dictionaries are now saved in a single pickle file `dict.p`. Alternatively, you can download the file from the book's GitHub repository.

9.2.2 Sequence Padding and Batch Creation

We'll divide the training data into batches during training for computational efficiency and accelerated convergence, as we have done in previous chapters.

Creating batches for other data formats such as images is straightforward: simply group a specific number of inputs to form a batch since they all have the same size. However, in NLP, batching can be more complex due to the varying lengths of sentences. To standardize the length within a batch, we pad the shorter sequences. This uniformity is crucial since the numerical representations fed into the Transformer need to have the same length. For instance, English phrases in a batch may vary in length (this can also happen to French phrases in a batch). To address this, we append zeros to the end of the numerical representations of shorter phrases in a batch, ensuring that all inputs to the Transformer model are of equal length.

Note

Incorporating BOS and EOS tokens at the beginning and end of each sentence, as well as padding shorter sequences within a batch, is a distinctive feature in machine language translation. This distinction arises from the fact that the input consists of entire sentences or phrases. In contrast, as you will see in the next two chapters, training a text generation model does not entail these processes; the model's input contains a predetermined number of tokens.

We start by converting all English phrases into their numerical representations, and then apply the same process to the French phrases:

```
out_en_ids=[[en_word_dict.get(w,1) for w in s] for s in en_tokens
out_fr_ids=[[fr_word_dict.get(w,1) for w in s] for s in fr_tokens
sorted_ids=sorted(range(len(out_en_ids)),
                  key=lambda x:len(out_en_ids[x]))
out_en_ids=[out_en_ids[x] for x in sorted_ids]
out_fr_ids=[out_fr_ids[x] for x in sorted_ids]
```

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

```
import numpy as np

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

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

Note that we have sorted observations in the training dataset by the length of the English phrases before placing them into batches. This method ensures that the observations within each batch are of a comparable length, consequently decreasing the need for padding. As a result, this approach not only reduces the overall size of the training data but also accelerates the training process.

To pad sequences in a batch to the same length, we define the following function:

```
def seq_padding(X, padding=0):
    L = [len(x) for x in X]
    ML = max(L)    #A
    padded_seq = np.array([np.concatenate([x, [padding] * (ML - l
        if len(x) < ML else x for x in X])    #B
    return padded_seq
```

The function seq_padding() first identifies the longest sequence within the

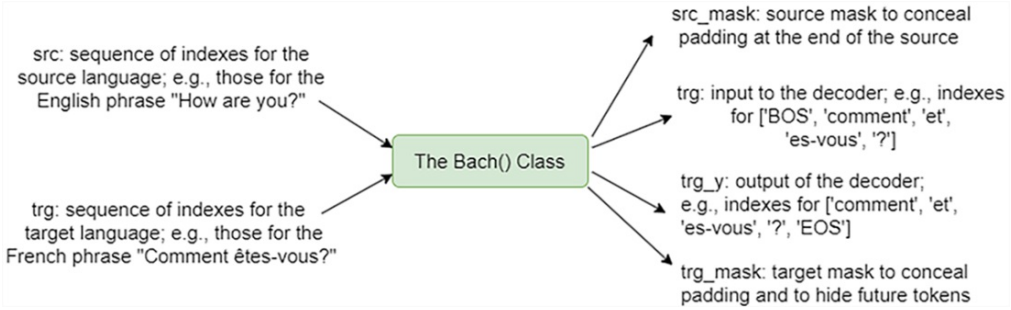
batch. Then, it appends zeros to the end of shorter sequences to ensure that every sequence in the batch matches this maximum length.

To conserve space, we have created a Batch() class within the local module named ch09util. You can obtain the ch09util.py file from the book's GitHub repository. Once downloaded, store it in the /utils/ directory on your computer. I encourage you to open this file and review its contents to understand its functionality better. The Batch() class is defined as follows in the local module:

Listing 9.4 Creating a Bach() class in the local module

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

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

Figure 9.7 What does the Bach() class do. The Bach() class takes two inputs: src and trg, sequences of indexes for the source language and the target language, respectively. It adds several attributes to the training data: src_mask, the source mask to conceal padding; modified trg, the input to the encoder; trg_y, the output to the decoder; trg_mask, the target mask to hide padding and future tokens.



The Batch() class processes a batch of English and French phrases,

converting them into a format suitable for training. To make this explanation more tangible, consider the English phrase 'How are you?' and its French equivalent 'Comment êtes-vous?' as our example. The Batch() class receives two inputs: `src`, which is the sequence of indexes representing the tokens in 'How are you?', and `trg`, the sequence of indexes for the tokens in 'Comment êtes-vous?'. This class generates a tensor, `src_mask`, to conceal the padding at the sentence's end. For instance, the sentence 'How are you?' is broken down into six tokens: ['BOS', 'how', 'are', 'you', '?', 'EOS']. If this sequence is part of a batch with a maximum length of eight tokens, two zeros are added to the end. The `src_mask` tensor instructs the model to disregard the final two tokens in such scenarios.

The Batch() class additionally prepares the input and output for the Transformer's decoder. Consider the French phrase 'Comment êtes-vous?', which is transformed into six tokens: ['BOS', 'comment', 'et', 'es-vous', '?', 'EOS']. The indexes of these first five tokens serve as the input to the decoder, named `trg`. Next, we shift this input one token to the right to form the decoder's output, `trg_y`. Hence, the input comprises indexes for ['BOS', 'comment', 'et', 'es-vous', '?'], while the output consists of indexes for ['comment', 'et', 'es-vous', '?', 'EOS']. This approach mirrors what we discussed in Chapter 8 and is designed to compel the model to predict the next token based on the previous ones.

The Batch() class also generates a mask, `trg_mask`, for the decoder's input. The aim of this mask is to conceal the subsequent tokens in the input, ensuring that the model relies solely on previous tokens for making predictions. This mask is produced by the make_std_mask() function, which is defined within the local module ch09util you just downloaded, as follows:

```
import numpy as np
def subsequent_mask(size):
    attn_shape = (1, size, size)
    subsequent_mask = np.triu(np.ones(attn_shape),k=1).astype('ui
    output = torch.from_numpy(subsequent_mask) == 0
    return output

def make_std_mask(tgt, pad):
    tgt_mask=(tgt != pad).unsqueeze(-2)
    output=tgt_mask & subsequent_mask(tgt.size(-1)).type_as(tgt_m
    return output
```

The subsequent_mask() function generates a mask specifically for a sequence, instructing the model to focus solely on the actual sequence and disregard the padded zeros at the end, which are used only to standardize sequence lengths. The make_std_mask() function, on the other hand, constructs a standard mask for the target sequence. This standard mask has the dual role of concealing both the padded zeros and the future tokens in the target sequence. We will explore the concept of target masking further when discussing masked multi-head self-attention later in this chapter.

Below, we import the Batch() class from the local module and use it to create batches of training data:

```
from utils.ch09util import Batch

batches=[]
for b in batch_indexs:
    batch_en=[out_en_ids[x] for x in b]
    batch_fr=[out_fr_ids[x] for x in b]
    batch_en=seq_padding(batch_en)
    batch_fr=seq_padding(batch_fr)
    batches.append(Batch(batch_en,batch_fr))
```

The `batches` list comprises data batches intended for training. Each batch in this list contains 128 pairs, where each pair contains numerical representations of an English phrase and its corresponding French translation.

9.2.3 Word embedding and positional encoding

The numerical representations of the English and French phrases involve a large number of indexes. To determine the exact number of distinct indexes required for each language, we can count the number of unique elements in the 'en_word_dict' and 'fr_word_dict' dictionaries. Doing so will reveal the total number of unique tokens in each language's vocabulary (we'll use them as inputs to the Transformer later):

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

The output is as follows:

```
there are 11055 distinct English tokens
there are 11239 distinct French tokens
```

In our dataset, there are 11,055 unique English tokens and 11,239 unique French tokens. Utilizing one-hot encoding for these would result in an excessively high number of parameters to train. To address this, we will employ word embeddings, which compress the numerical representations into continuous vectors, each with a length of d_model=256.

This is achieved through the use of the Embeddings() class, which is defined in the local module ch09util, as follows:

```
import math

class Embeddings(nn.Module):
    def __init__(self, d_model, vocab):
        super().__init__()
        self.lut = nn.Embedding(vocab, d_model)
        self.d_model = d_model

    def forward(self, x):
        out = self.lut(x) * math.sqrt(self.d_model)
        return out
```

The Embeddings() class defined above is the same as PyTorch's Embedding() class except that it multiplies the output by the square root of d_model, which is 256. This multiplication is intended to counterbalance the division by the square root of d_model that occurs later during the computation of attention scores. The Embeddings() class decreases the dimensionality of the numerical representations of English and French phrases. We discussed in detail how PyTorch's Embedding() class works in Chapter 8.

To accurately represent the sequence order of elements in both input and output, we introduce the PositionalEncoding() class in the local module, as outlined below:

Listing 9.5 A class to calculate positional encoding

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

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

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

In our example, the indexes for the six tokens ['BOS', 'how', 'are', 'you', '?', 'EOS'] from the English phrase are first processed through a word embedding layer. This step transforms these indexes into a tensor with the dimensions of (1, 6, 256): 1 means there is only one sequence in the batch; 6 means there are 6 tokens in the sequence; 256 means each token is represented by a 256-value vector. After this word embedding process, the PositionalEncoding() class is employed to calculate the positional encodings for the indexes corresponding to the tokens ['BOS', 'how', 'are', 'you', '?', 'EOS']. This is done to provide the model with information about the position of each token in the sequence. Better yet, we can tell you the exact values of the positional encodings for the above six tokens by using the following code block:

```
from utils.ch09util import PositionalEncoding
```

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

pe = PositionalEncoding(256, 0.1)    #A
x = torch.zeros(1, 8, 256).to(DEVICE)    #B
y = pe.forward(x)    #C
print(f"the shape of positional encoding is {y.shape}")
print(y)    #D
```

We first create an instance, `pe`, of the PositionalEncoding() class by setting the model dimension to 256 and the dropout rate to 0.1. Since the output from this class is the sum of word embedding and positional encoding, we create a word embedding filled with zeros and feed it to `pe`: this way, the output is the same as the positional encoding.

After running the above code block, you'll see the following output:

```
the shape of positional encoding is torch.Size([1, 8, 256])
tensor([[[ 0.0000e+00,  1.1111e+00,  0.0000e+00,  ...,  0.0000e+0
           0.0000e+00,  1.1111e+00],
         [ 9.3497e-01,  6.0034e-01,  8.9107e-01,  ...,  1.1111e+0
           1.1940e-04,  1.1111e+00],
         [ 0.0000e+00, -4.6239e-01,  1.0646e+00,  ...,  1.1111e+0
           2.3880e-04,  1.1111e+00],
         ...,
         [-1.0655e+00,  3.1518e-01, -1.1091e+00,  ...,  1.1111e+0
           5.9700e-04,  1.1111e+00],
         [-3.1046e-01,  1.0669e+00, -0.0000e+00,  ...,  0.0000e+0
           7.1640e-04,  1.1111e+00],
         [ 7.2999e-01,  8.3767e-01,  2.5419e-01,  ...,  1.1111e+0
           8.3581e-04,  1.1111e+00]]], device='cuda:0')
```

The above tensor represents the positional encoding for the English phrase "How are you?" It's important to note that this positional encoding also has the dimensions of (1, 6, 256), which matches the size of the word embedding for "How are you?". The next step involves combining the word embedding and positional encoding into a single tensor.

An essential characteristic of positional encodings is that their values are the same no matter what the input sequences are. This means that regardless of the specific input sequence, the positional encoding for the first token will always be the same 256-value vector, [0.0000e+00, 1.1111e+00, ...,

1.1111e+00], as shown in the above output. Similarly, the positional encoding for the second token will always be [9.3497e-01, 6.0034e-01, ..., 1.1111e+00], and so on. Their values don’t change during the training process, either.

9.3 Build a Transformer for English-French translation

We’ll develop and train an encoder-decoder Transformer designed for translating English into French. The coding in this project is adapted from the work of Chris Cui in translating Chinese to English (https://github.com/cuicaihao/Annotated-Transformer-English-to-Chinese-Translator) and Alexander Rush's German-to-English translation project (https://github.com/harvardnlp/annotated-transformer).

This section discusses how to construct a Transformer to translate English to French. Specifically, we’ll delve into the process of building the encoder and decoder, including the various sublayers within each component and how to implement the attention mechanism.

9.3.1 The attention mechanism

While there are different attention mechanisms, we'll use the scaled dot product attention (SDPA) because it’s widely used and effective. The SDPA attention mechanism uses query, key, and value to calculate the relationships among elements in a sequence. It assigns scores to show how an element in a sequence is related to all other elements.

Instead of using one set of query, key, and value vectors, the Transformer model uses a concept called multi-head attention. Our 256-dimensional query, key, and value vectors are split into 8 heads, and each head has a set of query, key, and value vectors with dimensions of 32 (because 256/8=32). Each head pays attention to different parts or aspects of the input, enabling the model to capture a broader range of information and form a more detailed and contextual understanding of the input data. For example, multi-head attention allows the model to capture the multiple meanings of the word

“bank” in the pun joke, “Why is the river so rich? Because it has two banks.”

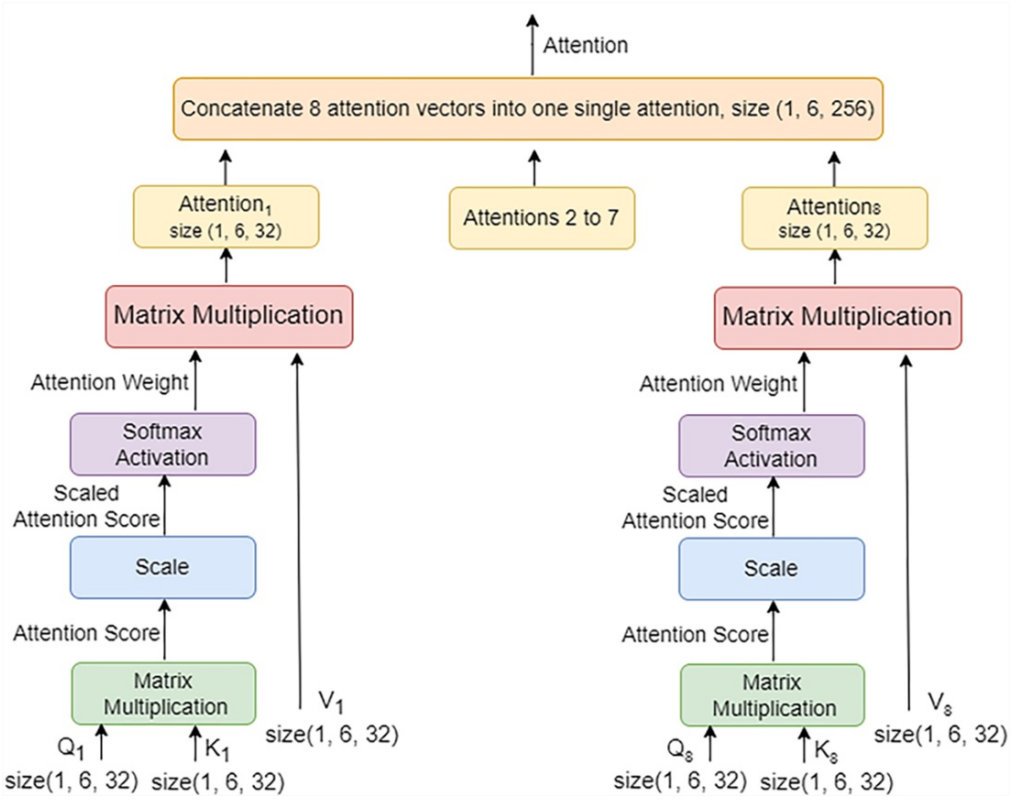
To implement this, we define the attention() function in the local module ch09util as follows:

Listing 9.6 How to calculate attention based on query, key, and value

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

The attention() function takes query, key, and value as inputs and calculates attention and attention weights as we discussed in the introduction of this chapter. The scaled attention score is the dot product of query and key, scaled by the square root of the dimension of the key, d_k. We apply the softmax function on the scaled attention score to obtain attention weights. Finally, attention is calculated as the dot product of attention weights and value.

Figure 9.8 An example of multi-head attention. This diagram uses the calculation of the multi-head self attention for the phrase “How are you?” as an example. We first pass the embedding through three neural networks to obtain query Q, key K, and value V, each with a size of (1, 6, 256). We split them into 8 heads, each with a set of Q, k, and V, with a size of (1, 6, 32). We calculate the attention in each head. The attention vectors from the 8 heads are then joined back into one single attention vector, with a size of (1, 6, 256).



Let’s use our running example to show how multi-head attention works. The embedding for “How are you?” is a tensor with a size of (1, 6, 256), as we explained in the last section (after we add positional encoding to word embedding). This embedding is passed through three linear layers to obtain query Q, key K, and value V, each of the same size (1, 6, 256). These are divided into eight heads, resulting in eight distinct sets of Q, K, and V, now sized (1, 6, 256/8=32) each. The attention function, as defined earlier, is applied to each of these sets, yielding eight attention outputs, each also sized (1, 6, 32). We then concatenate the eight attention outputs into one single attention, and the result is a tensor with a size of (1, 6, 32x8=256). Finally, this combined attention passes through another linear layer sized 256 by 256, leading to the output from the MultiHeadAttention() class. This output maintains the original input's dimensions, which are (1, 6, 256).

This is implemented in the following code listing.

Listing 9.7 Calculating multi-head attention

```
from copy import deepcopy
class MultiHeadedAttention(nn.Module):
```

```
    def __init__(self, h, d_model, dropout=0.1):
        super().__init__()
        assert d_model % h == 0
        self.d_k = d_model // h
        self.h = h
        self.linears = nn.ModuleList([deepcopy(
            nn.Linear(d_model, d_model)) for i in range(4)])
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        if mask is not None:
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)
        query, key, value = [l(x).view(nbatches, -1, self.h,
           self.d_k).transpose(1, 2)
         for l, x in zip(self.linears, (query, key, value))]    #
        x, self.attn = attention(
            query, key, value, mask=mask, dropout=self.dropout)
        x = x.transpose(1, 2).contiguous().view(
            nbatches, -1, self.h * self.d_k)    #C
        output = self.linears[-1](x)    #D
        return output
```

Each encoder layer and decoder layer also contain a feed-forward sublayer, which is a two-layer fully connected neural network, with the purpose of enhancing the model's ability to capture and learn intricate features in the training dataset. Further, the neural network processes each embedding independently. It doesn't treat the sequence of embeddings as a single vector. Therefore, we often call it a position-wide feed-forward network (or a 1-D convolutional network). For that purpose, we define a PositionwiseFeedForward() class in the local module as follows:

```
class PositionwiseFeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super().__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)
    def forward(self, x):
        h1 = self.w_1(x)
        h2 = self.dropout(h1)
        return self.w_2(h2)
```

The PositionwiseFeedForward() class is defined with two key parameters: `d_ff`, the dimensionality of the feed-forward layer, and `d_model`, representing the model's dimension size. Typically, `d_ff` is chosen to be four times the size of `d_model`. In our example, `d_model` is 256 and we therefore set `d_ff` to 256*4=1024. This practice of enlarging the hidden layer in comparison to the model size is a standard approach in Transformer architectures. It enhances the network's ability to capture and learn intricate features in the training dataset.

9.3.2 Create an encoder-decoder Transformer

To create an encoder-decoder Transformer, we first define a Transformer() class in the local module.

Open the file ch09util.py, and you'll see the definition of the class as follows:

Listing 9.8 A class to represent an encoder-decoder Transformer

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

The Transformer() class is constructed with five key components: encoder, decoder, src_embed, tgt_embed, and generator, which we'll explain in detail below. At its core, the Transformer architecture comprises an encoder and a decoder. The encoder's role is to transform the source language's embedding,

like that for an English phrase "How are you?", into a continuous vector representation, referred to as `memory` in the Transformer() class. The decoder then takes the output from the encoder and generates the translation in the target language.

To create an encoder for the Transformer() class, we stack N=6 identical encoder layers together to increase representation capacity and enable hierarchical feature extraction. Therefore, we first define the following EncoderLayer() class and *SublayerConnection()* class:

Listing 9.9 A class to define an encoder layer

```
class EncoderLayer(nn.Module):
    def __init__(self, size, self_attn, feed_forward, dropout):
        super().__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = nn.ModuleList([deepcopy(
        SublayerConnection(size, dropout)) for i in range(2)])
        self.size = size
    def forward(self, x, mask):
        x = self.sublayer[0](
            x, lambda x: self.self_attn(x, x, x, mask))    #A
        output = self.sublayer[1](x, self.feed_forward)    #B
        return output
class SublayerConnection(nn.Module):
    def __init__(self, size, dropout):
        super().__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)
    def forward(self, x, sublayer):
        output = x + self.dropout(sublayer(self.norm(x)))    #C
        return output
```

Each encoder layer is composed of two distinct sublayers: one is a multi-head self-attention layer, as outlined in the MultiHeadAttention() class, and the other is a straightforward, position-wise, fully connected feed-forward network, as specified in the PositionwiseFeedForward() class. Additionally, both of these sublayers incorporate layer normalization and residual connections. As explained in Chapter 6, a residual connection involves passing the input through a sequence of transformations (either the attention or the feed-forward layer in this context) and then adding the input back to

these transformations' output. The method of residual connection is employed to combat the issue of vanishing gradients, which is a common challenge in very deep networks. Another benefit of residual connections in Transformers is to provide a passage to pass the positional encodings (which are calculated only before the first layer) to subsequent layers.

Layer normalization is somewhat similar to the batch normalization we have implemented in Chapter 4. It standardizes the observations in a layer to have a zero mean and a unit standard deviation. To achieve this within the local module, we define the LayerNorm() class, which executes layer normalization as follows:

```
class LayerNorm(nn.Module):
    def __init__(self, features, eps=1e-6):
        super().__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps
    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        x_zscore = (x - mean) / torch.sqrt(std ** 2 + self.eps)
        output = self.a_2*x_zscore+self.b_2
        return output
```

The `mean` and `std` values in the above LayerNorm() class are the mean and standard deviation of the inputs in each layer. The a_2 and b_2 layers in the LayerNorm() class broadcast `x_zscore` back to the shape of the input `x`.

With that, we have finished discussing the architecture of the encoder layer. We can now create an encoder by stacking six encoder layers together. For that purpose, we define the *Encoder()* class in the local module as follows:

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

```
            x = layer(x, mask)
            output = self.norm(x)
        return output
```

9.3.3 Build decoder layers

Now that you understand how to create an encoder in the Transformer, let's move on to the decoder. The decoder consists of N=6 identical decoder layers. Each decoder layer consists of three sublayers: a multi-head self-attention layer, a second sub-layer that applies multi-head attention over the encoder stack's output, and a basic, position-wise, fully connected feed-forward network. Each of these three sublayers incorporates residual connections and layer normalization, similar to what we have done in encoder layers. Furthermore, the decoder stack's multi-head self-attention sub-layer is masked to prevent positions from attending to subsequent positions. The mask forces the model to use previous elements in a sequence to predict later elements. We'll explain how masked multi-head self-attention works in a moment.

Now we can stack six decoder layers together to form a decoder. Specifically, the *Decoder()* class and the *DecoderLayer()* class are defined in the local module as follows:

Listing 9.10 A decoder with six decoder layers

```
class Decoder(nn.Module):
    def __init__(self, layer, N):
        super().__init__()
        self.layers = nn.ModuleList(
            [deepcopy(layer) for i in range(N)])
        self.norm = LayerNorm(layer.size)
    def forward(self, x, memory, src_mask, tgt_mask):
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)    #A
        output = self.norm(x)
        return output
class DecoderLayer(nn.Module):
    def __init__(self, size, self_attn, src_attn,
                 feed_forward, dropout):
        super().__init__()
        self.size = size
```

```
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = nn.ModuleList([deepcopy(
        SublayerConnection(size, dropout)) for i in range(3)])
    def forward(self, x, memory, src_mask, tgt_mask):
        x = self.sublayer[0](x, lambda x:
                 self.self_attn(x, x, x, tgt_mask))    #B
        x = self.sublayer[1](x, lambda x:
                 self.src_attn(x, memory, memory, src_mask))    #
        output = self.sublayer[2](x, self.feed_forward)    #D
        return output
```

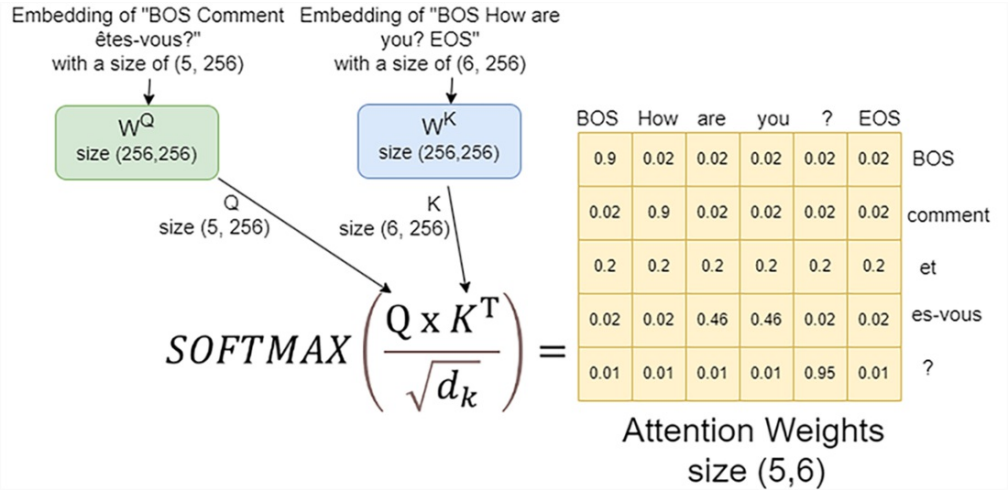
To illustrate the operation of a decoder layer, let's consider our ongoing example. The decoder takes in tokens ['BOS', 'comment', 'et', 'es-vous', '?'], along with the output from the encoder (referred to as `memory` in the above code block), to predict the sequence ['comment', 'et', 'es-vous', '?', 'EOS']. The embedding of ['BOS', 'comment', 'et', 'es-vous', '?'] is a tensor of size (1, 5, 256): 1 is the number of sequences in the batch, 5 is the number of tokens in the sequence, and 256 means each token is represented by a 256-value vector. We pass this embedding through the first sublayer, a masked multi-head self-attention layer. This process is similar to the multi-head self-attention calculation you saw earlier in the encoder layer. However, the process utilizes a mask, designated as `tgt_mask` in the above code block, which is a 5x5 tensor with the following values in the ongoing example:

```
tensor([[ True, False, False, False, False],
        [ True,  True, False, False, False],
        [ True,  True,  True, False, False],
        [ True,  True,  True,  True, False],
        [ True,  True,  True,  True,  True]], device='cuda:0')
```

As you may have noticed, the lower half of the mask (values below the main diagonal in the tensor) is turned on as `True`, and the upper half of the mask (values above the main diagonal) is turned off as `False`. When this mask is applied to the attention scores, it results in the first token attending only to itself during the first time step. In the second time step, attention scores are calculated exclusively between the first two tokens. As the process continues, for example, in the third time step, the decoder uses tokens ['BOS', 'comment', 'et'] to predict the token 'es-vous', and the attention scores are

computed only among these three tokens, effectively hiding the future tokens ['es-vous', '?'].

Figure 9.9 An example of how cross-attention weights are calculated between the input to the decoder and the output from the encoder. The input to the decoder is passed through a neural network to obtain query Q. The output from the encoder is passed through a different neural network to obtain key K. The scaled cross attention scores are calculated as the dot product of Q and K divided by the square root of the dimension of K. Finally, we apply the softmax function on the scaled cross-attention scores to obtain the cross-attention weights, which demonstrate how each element in Q is related to all elements in K.



Following this process, the output generated from the first sublayer, which is a tensor of size (1, 5, 256), matches the input's size. This output, which we can refer to as x, is then fed into the second sublayer. Here, cross attention is computed between x and the output of the encoder stack, termed `memory`. You may remember that `memory` has a dimension of (1, 6, 256) since the English phrase “How are you?” is converted to six tokens ['BOS', 'how', 'are', 'you', '?', 'EOS'].

To calculate the cross attention between x and `memory`, we first pass x through a neural network to obtain query, which has a dimension of (1, 5, 256). We then pass *memory* through two neural networks to obtain key and value, each having a dimension of (1, 6, 256). The scaled attention score is calculated using the formula as specified in equation 9.1. This scaled attention score has a dimension of (1, 5, 6): the query Q has a dimension of (1, 5, 256) and the transposed key K has a dimension of (1, 256, 6). Therefore, the scaled attention score, which is the dot product of the two scaled by $\sqrt{d_k}$, has a size

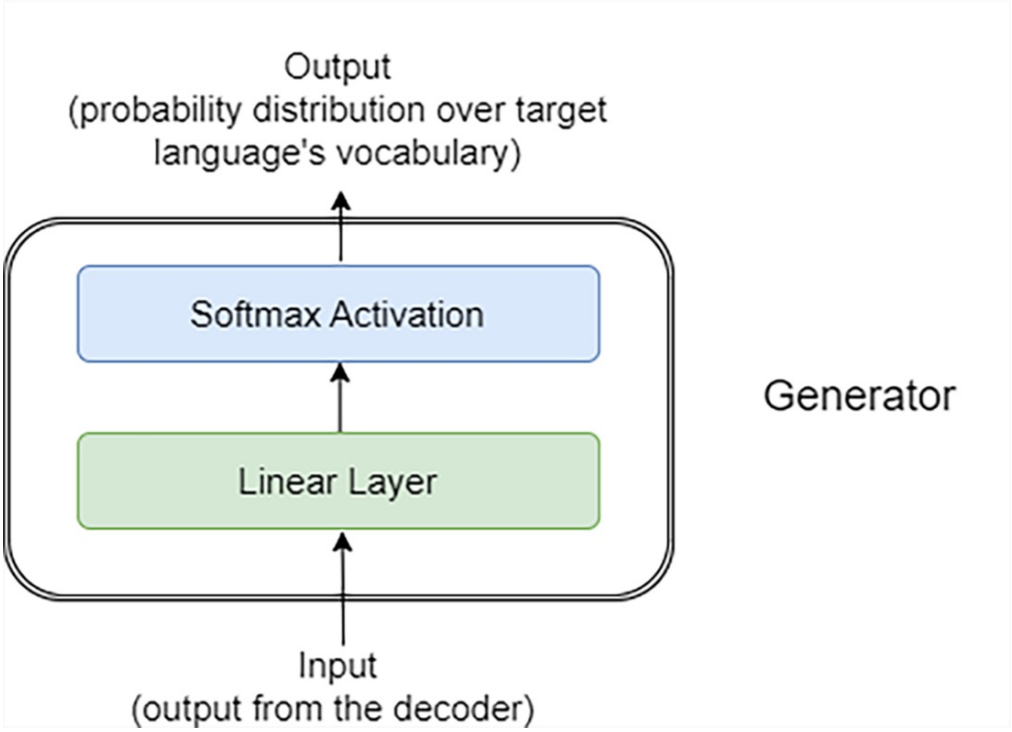
of (1, 5, 6). After applying the softmax function to the scaled attention score, we obtain attention weights, which is a 5 by 6 matrix. This matrix tells us how the five tokens in the French input ['BOS', 'comment', 'et', 'es-vous', '?'] attend to the six tokens in the English phrase ['BOS', 'how', 'are', 'you', '?', 'EOS']. This is how the decoder captures the meaning of the English phrase when translating.

The final cross attention in the second sublayer is then calculated as the dot product of attention weights and the value vector V. The attention weights have a dimension of (1, 5, 6) and the value vector has a dimension of (1, 6, 256), so the final cross attention, which is the dot product of the two, has a size of (1, 5, 256). Therefore, the input and output of the second sublayer have the same dimension of (1, 5, 256). After processing through this second sublayer, the output is then directed through the third sublayer, which is a feed-forward network.

9.3.4 Put all pieces together

Before we put all the pieces together, we define a Generator() class in the local module to generate the probability distribution of the next token. The idea is to attach a head to the decoder for downstream tasks. In our case, the downstream task is to predict the next token in the French translation.

Figure 9.10 The structure of the generator in the Transformer. The generator converts the output from the decoder stack to a probability distribution over the targe language's vocabulary, so that the Transformer can use the distribution to predict the next token in the French translation of an English phrase. The generator contains a linear layer so that the number of outputs is the same as the number of tokens in the French vocabulary. The generator also applies a softmax activation to the output so that the output is a probability distribution.



The class is defined as follows:

```
class Generator(nn.Module):
    def __init__(self, d_model, vocab):
        super().__init__()
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        out = self.proj(x)
        probs = nn.functional.log_softmax(out, dim=-1)
        return probs
```

The Generator() class produces predicted probabilities for each index that corresponds to the tokens in the target language. This enables the model to sequentially predict tokens in an autoregressive manner, utilizing previously generated tokens and the encoder's output.

Now we are ready to create a Transformer model to translate English to French. The create_model() function defined in the local module accomplishes that:

Listing 9.11 Create a Transformer to translate English to French

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

The primary element of the create_model() function is the Transformer() class, which was previously defined. Recall that the Transformer() class is built with five essential elements: encoder, decoder, src_embed, tgt_embed, and generator. Within the create_model() function, we sequentially construct these five components, leveraging the recently defined Encoder(), Decoder(), and Generator() classes. For generating the source language embedding, we process numerical representations of English phrases using word embedding and positional encoding, combining the results to form the src_embed component. Similarly, for the target language, we process numerical representations of French phrases in the same manner, using the combined output as the tgt_embed component.

Finally, we import the *create_model()* function from the local module and construct a Transformer so that we can train it to translate English to French:

```
from utils.ch09util import create_model

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

The paper *Attention Is All You Need* uses various combinations of

hyperparameters when constructing the model. Here we choose a model dimension of 256 with 8 heads because we find this combination does a good job translating English to French in our setting. Interested readers could potentially use a validation set to tune hyperparameters to select the best model in their own projects.

9.4 Train and use the Transformer

Our constructed English to French Transformer model can be viewed as a multi-category classifier. The core objective is to predict the next token in the French vocabulary when translating an English sentence. This is somewhat similar to the image classification project we discussed in Chapter 2, though this model is significantly more complex. This complexity necessitates careful selection of the loss function, optimizer, and training loop parameters.

In this section, we will detail the process of selecting an appropriate loss function and optimizer. We will train the Transformer using batches of English-to-French translations, prepared earlier in this chapter, as our training dataset. After the model is trained, we will guide you through the application of the trained model for translating English into French.

9.4.1 Loss function, optimizer, and the training loop

We'll follow the original paper *Attention Is All You Need* and use label smoothing during training.

Label smoothing is commonly used in training deep neural networks to improve the generalization of the model. It is used to address overconfidence problems (the predicted probability is greater than the true probability) and overfitting in classifications. Specifically, it modifies the way the model learns by adjusting the target labels, aiming to reduce the model's confidence in the training data, which can lead to better performance on unseen data.

In a typical classification task, the target labels are represented in a one-hot encoding format. This representation implies absolute certainty about the correctness of the label for each training sample. Training with absolute certainty can lead to two main issues. The first is overfitting: the model

becomes overly confident in its predictions, fitting too closely to the training data, which can harm its performance on new, unseen data. The second issue is poor calibration: models trained this way often output overconfident probabilities. For instance, they might output a probability of 99% for a correct class when, realistically, the confidence should be lower.

Label smoothing adjusts the target labels to be less confident. Instead of having a target label of [1, 0, 0] for a 3-class problem, you might have something like [0.9, 0.05, 0.05]. This approach encourages the model not to be too confident about its predictions by penalizing overconfident outputs. The smoothed labels are a mixture of the original label and some distribution over the other labels (usually the uniform distribution).

We define the following LabelSmoothing() class in the local module ch09util as follows:

Listing 9.12 A class to conduct label smoothing

```
class LabelSmoothing(nn.Module):
    def __init__(self, size, padding_idx, smoothing=0.1):
        super().__init__()
        self.criterion = nn.KLDivLoss(reduction='sum')
        self.padding_idx = padding_idx
        self.confidence = 1.0 - smoothing
        self.smoothing = smoothing
        self.size = size
        self.true_dist = None
    def forward(self, x, target):
        assert x.size(1) == self.size
        true_dist = x.data.clone()    #A
        true_dist.fill_(self.smoothing / (self.size - 2))
        true_dist.scatter_(1,
               target.data.unsqueeze(1), self.confidence)    #B
        true_dist[:, self.padding_idx] = 0
        mask = torch.nonzero(target.data == self.padding_idx)
        if mask.dim() > 0:
            true_dist.index_fill_(0, mask.squeeze(), 0.0)
        self.true_dist = true_dist
        output = self.criterion(x, true_dist.clone().detach())
        return output
```

The LabelSmoothing() class first extracts the predictions from the model. It

then smoothes the actual labels in the training dataset by adding noise to it. The parameter `smoothing` controls how much noise we inject into the actual label. The label [1, 0, 0] is smoothed to [0.9, 0.05, 0.05] if you set `smoothing=0.1`; and it is smoothed to [0.95, 0.025, 0.025] if you set `smoothing=0.05`, for example. The class then calculates the loss by comparing the predictions with the smoothed labels.

As in previous chapters, the optimizer we use is the Adam optimizer. However, instead of using a constant learning rate throughout training, we define the NoamOpt() class in the local module to change the learning rate during the training process, as follows:

```
class NoamOpt:
    def __init__(self, model_size, factor, warmup, optimizer):
        self.optimizer = optimizer
        self._step = 0
        self.warmup = warmup    #A
        self.factor = factor
        self.model_size = model_size
        self._rate = 0
    def step(self):    #B
        self._step += 1
        rate = self.rate()
        for p in self.optimizer.param_groups:
            p['lr'] = rate
        self._rate = rate
        self.optimizer.step()
    def rate(self, step=None):
        if step is None:
            step = self._step
        output = self.factor * (self.model_size ** (-0.5) *
        min(step ** (-0.5), step * self.warmup ** (-1.5)))    #C
        return output
```

The NoamOpt() class, as defined above, implements a warm-up learning rate strategy. First, it increases the learning rate linearly during the initial warmup steps of training. Following this warm-up period, the class then decreases the learning rate, adjusting it in proportion to the inverse square root of the training step number.

Next, we create the optimizer for training as follows:

```
from utils.ch09util import NoamOpt
```

```
optimizer = NoamOpt(256, 1, 2000, torch.optim.Adam(
    model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))
```

To define the loss function for training, we first create the following SimpleLossCompute() class in the local module:

Listing 9.13 A class to compute loss

```
class SimpleLossCompute:
    def __init__(self, generator, criterion, opt=None):
        self.generator = generator
        self.criterion = criterion
        self.opt = opt
    def __call__(self, x, y, norm):
        x = self.generator(x)    #A
        loss = self.criterion(x.contiguous().view(-1, x.size(-1))
                              y.contiguous().view(-1)) / norm
        loss.backward()    #C
        if self.opt is not None:
            self.opt.step()    #D
            self.opt.optimizer.zero_grad()
        return loss.data.item() * norm.float()
```

The SimpleLossCompute() class is designed with three key elements: the 'generator', serving as the prediction model, the 'criterion', which is a function to calculate loss, and 'opt', the optimizer. This class processes a batch of training data, denoted as (x, y), by utilizing the generator for predictions. It subsequently evaluates the loss by comparing these predictions with the actual labels y (which is handled by the LabelSmoothing() class defined earlier; the actual labels y will be smoothed in the process). The class computes gradients relative to the model parameters and utilizes the optimizer to update these parameters accordingly.

We are now ready to define the loss function as follows:

```
from utils.ch09util import (LabelSmoothing,
       SimpleLossCompute)

criterion = LabelSmoothing(tgt_vocab,
                           padding_idx=0, smoothing=0.1)
loss_func = SimpleLossCompute(
```

```
                model.generator, criterion, optimizer)
```

Next, we'll train the Transformer by using the data we prepared earlier in the Chapter.

We could potentially divide the training data into a train set and a validation set and train the model until the performance of the model doesn't improve on the validation set, similar to what we have done in Chapter 2. However, to save space, we'll train the model for 100 epochs. We'll calculate the loss and the number of tokens from each batch. After each epoch, we calculate the average loss in the epoch as the ratio between the total loss and the total number of tokens:

Listing 9.14 Train a Transformer to translate English to French

```
for epoch in range(100):
    model.train()
    tloss=0
    tokens=0
    for batch in batches:
        out = model(batch.src, batch.trg,
                    batch.src_mask, batch.trg_mask)    #A
        loss = loss_func(out, batch.trg_y, batch.ntokens)    #B
        tloss += loss
        tokens += batch.ntokens    #C
    print(f"Epoch {epoch}, average loss: {tloss/tokens}")
torch.save(model.state_dict(),"files/en2fr.pth")    #D
```

The above training process takes a couple of hours if you are using a CUDA-enabled GPU. It may take a full day if you are using CPU training. Once the training is done, the model weights are saved as en2fr.pth on your computer. Alternatively, you can download the trained weights from my website (https://gattonweb.uky.edu/faculty/lium/gai/ch9.zip).

9.4.2 Translate English to French with the trained model

Now that you have trained the Transformer, you can use it to translate any English sentence to French. We define a function translate() as follows:

Listing 9.15 Define a translate() function to translate English to French

```
def translate(eng):
    tokenized_en=tokenizer.tokenize(eng)
    tokenized_en=["BOS"]+tokenized_en+["EOS"]
    enidx=[en_word_dict.get(i,UNK) for i in tokenized_en]
    src=torch.tensor(enidx).long().to(DEVICE).unsqueeze(0)
    src_mask=(src!=0).unsqueeze(-2)
    memory=model.encode(src,src_mask)    #A
    start_symbol=fr_word_dict["BOS"]
    ys = torch.ones(1, 1).fill_(start_symbol).type_as(src.data)
    translation=[]
    for i in range(100):
        out = model.decode(memory,src_mask,ys,
        subsequent_mask(ys.size(1)).type_as(src.data))    #B
        prob = model.generator(out[:, -1])
        _, next_word = torch.max(prob, dim=1)
        next_word = next_word.data[0]
        ys = torch.cat([ys, torch.ones(1, 1).type_as(
            src.data).fill_(next_word)], dim=1)
        sym = fr_idx_dict[ys[0, -1].item()]
        if sym != 'EOS':
            translation.append(sym)
        else:
            break    #C
    trans="".join(translation)
    trans=trans.replace("</w>"," ")
    for x in '''?:;.,'("-!&)%''':
        trans=trans.replace(f" {x}",f"{x}")    #D
    print(trans)
    return trans
```

To translate an English phrase to French, we first use the tokenizer to convert the English sentence to tokens. We then add "BOS" and "EOS" at the beginning and the end of the phrase. We use the dictionary en_word_dict we created earlier in the chapter to convert tokens to indexes. We feed the sequence of indexes to the encoder in the trained model. The encoder produces an abstract vector representation and passes it on to the decoder.

Based on the abstract vector representation of the English sentence produced by the encoder, the decoder in the trained model starts translating in an autoregressive manner, starting with the beginning token "BOS". In each time step, the decoder generates the most likely next token based on previously generated tokens, until the predicted token is "EOS", which signals the end of the sentence. Note this is slightly different from the text generation approach discussed in Chapter 8, where the next token is chosen randomly in

accordance with its predicted probabilities. Here, the method for selecting the next token is deterministic, meaning the token with the highest probability is chosen with certainty because we mainly care about accuracy. However, you can use top-K sampling and temperature if you want your translation to be creative.

Finally, we change the token separator to a space and remove the space before the punctuation marks. The output is the French translation in a clean format.

Let's try the translate() function with the English phrase "Today is a beautiful day!", like so:

```
from utils.ch09util import subsequent_mask

with open("files/dict.p","rb") as fb:
    en_word_dict,en_idx_dict,\
    fr_word_dict,fr_idx_dict=pickle.load(fb)
trained_weights=torch.load("files/en2fr.pth",
                           map_location=DEVICE)
model.load_state_dict(trained_weights)
model.eval()
eng = "Today is a beautiful day!"
translated_fr = translate(eng)
```

And the output is:

```
aujourd'hui est une belle journee!
```

You can verify that the French translation indeed means "Today is a beautiful day!" by using, say, Google Translate.

Let’s try a longer sentence and see if the trained model can successfully translate, like so:

```
eng = "A little boy in jeans climbs a small tree while another ch
translated_fr = translate(eng)
```

The output is:

```
un petit garcon en jeans grimpe un petit arbre tandis qu'un autre
```

When I translate the above output back to English using Google Translate, it says, "a little boy in jeans climbs a small tree while another child watches." Not exactly the same as the original English sentence but the meaning is the same.

Next, we'll test if the trained model generates the same translation for the two English sentences "I don't speak French." and "I do not speak French." First, let's try the sentence "I don't speak French.", like this:

```
eng = "I don't speak French."
translated_fr = translate(eng)
```

The output is:

```
je ne parle pas francais.
```

Now let's try the sentence "I do not speak French.", like so:

```
eng = "I do not speak French."
translated_fr = translate(eng)
```

The output this time is:

```
je ne parle pas francais.
```

The results indicate that French translations for the two sentences are exactly the same. This suggests that the encoder component of the Transformer successfully grasps the semantic essence of the two phrases. It then represents them as similar abstract continuous vector forms, which are subsequently passed on to the decoder. The decoder then generates translations based on these vectors and produces identical results.

Exercise 9.3

Use the translate() function to translate the following two English sentences to French. Compare the results with those from Google Translate and see if they are the same: (i) I love skiing in the winter! (ii) How are you?

In this chapter, you learned the architecture of Transformers and the key ingredient in them: the attention mechanism. You built an encoder-decoder Transformer from scratch for the purpose of translating English to French and trained it using more than 47,000 pairs of English-to-French translations. The trained model works well, translating common English phrases correctly!

In the following chapters, you'll explore the other two types of Transformers: encoder-only Transformers and decoder-only Transformers. You'll also learn to build them from scratch and use them to generate coherent text, better than the text you generated in Chapter 8 using LSTMs.

9.5 Summary

- Transformers are advanced deep-learning models that excel in handling sequence-to-sequence prediction challenges. Their strength lies in effectively understanding the relationships between elements in input and output sequences over long distances.
- The revolutionary aspect of the Transformer architecture is its attention mechanism. This mechanism assesses the relationship between words in a sequence by assigning weights, determining how closely words are related based on the training data. This enables Transformer models like ChatGPT to comprehend relationships between words, thus understanding human language more effectively.
- A common approach to calculate attention is scaled dot product attention (SDPA). This method is also called self-attention because the algorithm calculates how a word attends to all words in the sequence, including the word itself.
- To calculate SDPA, the input embedding X is processed through three distinct neural network layers, query (Q), key (K), and value (V). The corresponding weights for these layers are W^Q, W^K, and W^V. WE can calculate Q, K, and V as $Q=X*W^Q$, $K=X*Q^K$, and $V=X*W^V$. SDPA first calculates the attention score as follows:

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

where d_k represents the dimension of the key vector K. The next step is to apply the softmax function to these attention scores, converting them into attention weights. This ensures that the total attention a word gives to all words in the sentence sums to 100%. The final attention is then calculated as the dot product of these attention weights and the value vector V:

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

- Furthermore, instead of using one set of query, key, and value vectors, Transformer models use a concept called multi-head attention. The query, key, and value vectors are split into multiple heads. Each head pays attention to different parts or aspects of the input, enabling the model to capture a broader range of information and form a more detailed and contextual understanding of the input data. Multi-head attention is especially useful when a word has multiple meanings in a sentence.

[1] Vaswani et al, “Attention is all you need,” 2017, https://arxiv.org/abs/1706.03762.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio, 2014, Neural Machine Translation by Jointly Learning to Align and Translate. https://arxiv.org/abs/1409.0473.

Appendix. Installing Python, Jupyter Notebook, and PyTorch

This appendix covers

- How to install Anaconda on your computer based on your operating system
- Creating a Python virtual environment for projects in this book
- Installing Jupyter Notebook in the virtual environment
- Installing PyTorch based on whether you have a CUDA-enabled GPU

There exist various ways of installing Python and managing libraries and packages on your computer. This book uses Anaconda, an open-source Python distribution, package manager, and environment management tool. Anaconda stands out for its user-friendly nature and capacity to facilitate the effortless installation of numerous libraries and packages, which could be painful or downright impossible to install otherwise.

Specifically, Anaconda allows users to install packages through both 'conda install' and 'pip install,' broadening the spectrum of available resources. Subsequently, you will create a dedicated Python virtual environment for all the projects in this book. This segregation ensures that the libraries and packages used in this book remain isolated from any libraries utilized in other, unrelated projects, thus eliminating any potential interference.

We will use Jupyter Notebook as our integrated development environment (IDE). I will guide you through the installation of Jupyter Notebook in the Python virtual environment you just created. Finally, I will guide you through the process of installing PyTorch, Torchvision, and Torchaudio, based on whether your computer is equipped with a Compute Unified Device Architecture (CUDA)-enabled GPU.

A.1 Install Python and Set Up a Virtual

Environment

In this section, I'll guide you through the process of installing Anaconda on your computer based on your operating system. After that, you'll create a Python virtual environment for all projects in this book. Finally, you'll install Jupyter Notebook as your IDE to run Python programs in this book.

A.1.1 Install Anaconda

To install Python through the Anaconda distribution, follow the following steps.

First, go to the official Anaconda website at https://www.anaconda.com/products/individual and scroll to the bottom of the webpage. Locate and download the most recent Python 3 version tailored to your specific operating system (be it Windows, macOS, or Linux).

If you are using Windows, download the latest Python 3 graphical installer from the above link. Click on the installer and follow the provided instructions to install. To confirm the successful installation of Anaconda on your computer, you can search for the "Anaconda Navigator" application on your computer. If you can launch the application, Anaconda has been successfully installed.

For macOS users, the latest Python 3 graphical installer for Mac is recommended, although a command line installer option is also available. Execute the installer and comply with the provided instructions. Verify the successful installation of Anaconda by searching for the "Anaconda Navigator" application on your computer. If you can launch the application, Anaconda has been successfully installed.

The installation process for Linux is more complex than for other operating systems, as there is no graphical installer. Begin by identifying the latest Linux version. Select the appropriate x86 or Power8 and Power9 package. Click to download the latest installer bash script. The installer bash script is typically saved to your computer's "Downloads" folder by default. Install Anaconda by executing the bash script within a terminal. Upon completing

the installation, activate it by running the following command:

```
source ~/.bashrc
```

To access Anaconda Navigator, enter the following command in a terminal:

```
anaconda-navigator
```

If you can successfully launch the Anaconda Navigator on your Linux system, your installation of Anaconda is complete.

Exercise A.1

Install Anaconda on your computer based on your operating system. After that, open the Anaconda Navigator app on your computer to confirm the installation.

A.1.2 Set Up a Python Virtual Environment

It's highly recommended that you create a separate virtual environment for this book. Let’s name the virtual environment *dgai*. Execute the following command in the Anaconda prompt (Windows) or a terminal (Mac and Linux):

```
conda create -n dgai
```

After pressing the ENTER key on your keyboard, follow the instructions on the screen and press y when the prompt asks you y/n. To activate the virtual environment, run the following command in the same Anaconda prompt (Windows) or terminal (Mac and Linux):

```
conda activate dgai
```

The virtual environment isolates the Python packages and libraries that you use for this book from other packages and libraries that you use for other purposes. This prevents any undesired interference.

Exercise A.2

Create a Python virtual environment *dgai* on your computer. After that, activate the virtual environment.

A.1.3 Install Jupyter Notebook

Now, let's install Jupyter Notebook in the newly created virtual environment on your computer.

First, activate the virtual environment by running the following line of code in the Anaconda prompt (in Windows) or a terminal (in Mac or Linux):

```
conda activate dgai
```

To install Jupyter Notebook in the virtual environment, run the command:

```
conda install notebook
```

Follow the instructions all the way through to install the app.

To launch Jupyter Notebook, execute the following command in the same terminal with the virtual environment activated:

```
jupyter notebook
```

The Jupyter Notebook app will open in your default browser.

Exercise A.3

Install Jupyter Notebook in the Python virtual environment *dgai*. After that, open the Jupter Notebook app on your computer to confirm the installation.

A.2 Install PyTorch

In this section, I'll guide you through the installation of PyTorch, based on whether you have a Compute Unified Device Architecture (CUDA)-enabled GPU on your computer. The official PyTorch website, https://pytorch.org/get-started/locally/, provides updates on PyTorch installation with or without CUDA. I encourage you to check the website for

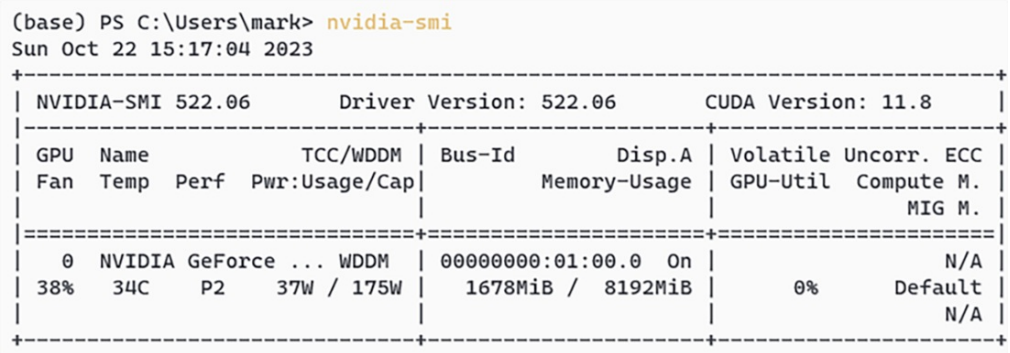
any updates.

CUDA is only available on Windows or Linux, not on Mac. To find out if your computer is equipped with a CUDA-enabled GPU, open the Windows PowerShell (in Windows) or a terminal (in Linux) and issue the following command:

```
nvidia-smi
```

If your computer has a CUDA-enabled GPU, you should see an output similar to figure Appendix.1. Further, make a note of the CUDA version as shown at the top right corner of the figure because you'll need this piece of information later when you install PyTorch. Figure Appendix.1 shows that the CUDA version is 11.8 on my computer. The version may be different on your computer.

Figure A.1 Checking if your computer has a CUDA-enabled GPU.



If you see an error message after running the command `nvidia-smi`, your computer doesn't have a CUDA-enabled GPU.

In the first subsection, I'll discuss how to install PyTorch if you don't have a CUDA-enabled GPU on your computer. You can use the CPU to train all generative AI models in this book. It just takes much longer. However, I'll provide you with the pre-trained models so that you can witness generative AI in action.

On the other hand, if you are using Windows or Linux operating system, and you do have a CUDA-enabled GPU on your computer, I'll guide you through

the installation of PyTorch with CUDA in the second subsection.

A.2.1 Install PyTorch without CUDA

To install PyTorch with CPU training, first activate the virtual environment *dgai* by running the following line of code in the Anaconda prompt (in Windows) or a terminal (in Mac or Linux):

```
conda activate dgai
```

You should be able to see *(dgai)* in front of your prompt, which indicates that you are now in the *dgai* virtual environment. To install PyTorch, issue the following line of command:

```
conda install pytorch torchvision torchaudio cpuonly -c pytorch
```

Follow the on-screen instructions to finish the installation. Here, we install three libraries together: PyTorch, Torchaudio, and Torchvision. Torchaudio is a library to process audio and signals and we need it to generate music later in this book. We'll also use the Torchvision library extensively in the book to process images.

If your Mac computer has an Apple silicon or AMD GPU with macOS 12.3 or later, you can potentially use the new Metal Performance Shaders (MPS) backend for GPU training acceleration. More information is available here https://developer.apple.com/metal/pytorch/ and here https://pytorch.org/get-started/locally/.

To check if the three libraries are successfully installed on your computer, run the following lines of code:

```
import torch, torchvision, torchaudio

print(torch.__version__)
print(torchvision.__version__)
print(torchaudio.__version__)
```

The output on my computer says:

```
2.0.1
```

```
0.15.2
2.0.2
```

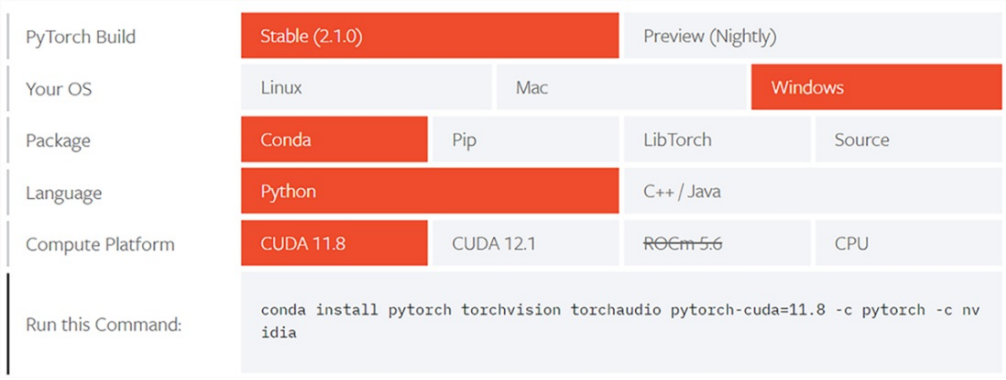
If you don't see an error message, you have successfully installed PyTorch on your computer.

A.2.2 Install PyTorch with CUDA

To install PyTorch with CUDA, first find out the CUDA version of your GPU, as shown at the top right corner of figure Appendix.1. My CUDA version is 11.8 so I'll use it as an example in the installation below.

If you go to the PyTorch website here, https://pytorch.org/get-started/locally/, you'll see an interactive interface as shown in figure Appendix.2.

Figure A.2 The interactive interface on how to install PyTorch



Once there, choose your operating system, select Conda as the Package, Python as the Language, and either CUDA 11.8 or CUDA 12.1 as your Computer Platform (based on what you have found out in the previous step). The command you need to run will be shown at the bottom panel. For example, I am using the Windows operating system, and I have CUDA 11.8 on my GPU. Therefore, the command for me is shown at the bottom panel of figure Appendix.2.

Once you know what command to run to install PyTorch with CUDA, activate the virtual environment by running the following line of code in the Anaconda prompt (Windows) or a terminal (Linux):

```
conda activate dgai
```

Then, issue the line of command you have found out in the last step. For me, the command line is:

```
conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c
```

Follow the on-screen instructions to finish the installation. Here, we install three libraries together: PyTorch, Torchaudio, and Torchvision. Torchaudio is a library to process audio and signals and we need it to generate music later in this book. We'll also use the Torchvision library extensively in the book to process images.

To make sure you have PyTorch correctly installed, run the following lines of code in a new cell in Jupyter Notebook:

```
import torch, torchvision, torchaudio

print(torch.__version__)
print(torchvision.__version__)
print(torchaudio.__version__)
device="cuda" if torch.cuda.is_available() else "cpu"
print(device)
```

The output is as follows on my computer:

```
2.0.1
0.15.2
2.0.2
cuda
```

The last line of the output above says `cuda`, indicating that I have installed PyTorch with CUDA. If you have installed PyTorch without CUDA on your computer, the output is `cpu`.

Exercise A.4

Install PyTorch, Torchvision, and Torchaudio on your computer based on your operating system and on whether your computer has GPU training acceleration. After that, print out the versions of the three libraries you just installed.