Part I is a general introduction to generative modeling and deep learning—the two fields that we need to understand in order to get started with generative deep learning!

In Chapter 1 we will define generative modeling and consider a toy example that we can use to understand some of the key concepts that are important to all generative models. We will also lay out the taxonomy of generative model families that we will explore in Part II of this book.

Chapter 2 provides a guide to the deep learning tools and techniques that we will need to start building more complex generative models. In particular, we will build our first example of a deep neural network—a multilayer perceptron (MLP)—using Keras. We will then adapt this to include convolutional layers and other improvements, to observe the difference in performance.

By the end of Part I you will have a good understanding of the core concepts that underpin all of the techniques in later parts of the book.

CHAPTER 1

Generative Modeling

Chapter Goals

In this chapter you will:

- Learn the key differences between generative and discriminative models.
- Understand the desirable properties of a generative model through a simple example.
- Learn about the core probabilistic concepts that underpin generative models.
- Explore the different families of generative models.
- Clone the codebase that accompanies this book, so that you can get started building generative models!

This chapter is a general introduction to the field of generative modeling.

We will start with a gentle theoretical introduction to generative modeling and see how it is the natural counterpart to the more widely studied discriminative modeling. We will then establish a framework that describes the desirable properties that a good generative model should have. We will also lay out the core probabilistic concepts that are important to know, in order to fully appreciate how different approaches tackle the challenge of generative modeling.

This will lead us naturally to the penultimate section, which lays out the six broad families of generative models that dominate the field today. The final section explains how to get started with the codebase that accompanies this book.

What Is Generative Modeling?

Generative modeling can be broadly defined as follows:

> Generative modeling is a branch of machine learning that involves training a model to produce new data that is similar to a given dataset.

What does this mean in practice? Suppose we have a dataset containing photos of horses. We can *train* a generative model on this dataset to capture the rules that govern the complex relationships between pixels in images of horses. Then we can *sample* from this model to create novel, realistic images of horses that did not exist in the original dataset. This process is illustrated in Figure 1-1.

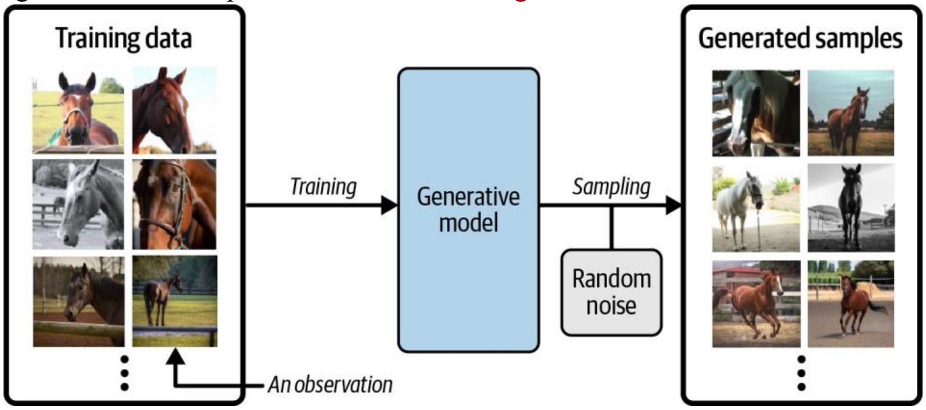


Figure 1-1. A generative model trained to generate realistic photos of horses

In order to build a generative model, we require a dataset consisting of many examples of the entity we are trying to generate. This is known as the *training data*, and one such data point is called an *observation*.

Each observation consists of many *features*. For an image generation problem, the features are usually the individual pixel values; for a text generation problem, the features could be individual words or groups of letters. It is our goal to build a model that can generate new sets of features that look as if they have been created using the same rules as the original data. Conceptually, for image generation this is an incredibly difficult task, considering the vast number of ways that individual pixel values can be assigned and the relatively tiny number of such arrangements that constitute an image of the entity we are trying to generate.

A generative model must also be *probabilistic* rather than *deterministic*, because we want to be able to sample many different variations of the output, rather than get the same output every time. If our model is merely a fixed calculation, such as taking the average value of each pixel in the training dataset, it is not generative. A generative model must include a random component that influences the individual samples generated by the model.

In other words, we can imagine that there is some unknown probabilistic distribution that explains why some images are likely to be found in the training dataset and other images are not. It is our job to build a model that mimics this distribution as closely as possible and then sample from it to generate new, distinct observations that look as if they could have been included in the original training set.

Generative Versus Discriminative Modeling

In order to truly understand what generative modeling aims to achieve and why this is important, it is useful to compare it to its counterpart, *discriminative modeling*. If you have studied machine learning, most problems you will have faced will have most likely been discriminative in nature. To understand the difference, let's look at an example.

Suppose we have a dataset of paintings, some painted by Van Gogh and some by other artists. With enough data, we could train a discriminative model to predict if a given painting was painted by Van Gogh. Our model would learn that certain colors, shapes, and textures are more likely to indicate that a painting is by the Dutch master, and for paintings with these features, the model would upweight its prediction accordingly. Figure 1-2 shows the discriminative modeling process—note how it differs from the generative modeling process shown in Figure 1-1.

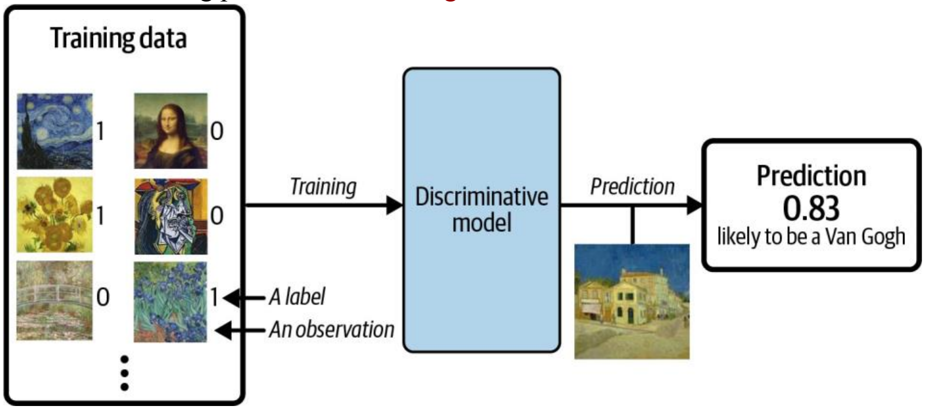


Figure 1-2. A discriminative model trained to predict if a given image is painted by Van Gogh

When performing discriminative modeling, each observation in the training data has a *label*. For a binary classification problem such as our artist discriminator, Van Gogh paintings would be labeled 1 and non–Van Gogh paintings labeled 0. Our model then

learns how to discriminate between these two groups and outputs the probability that a new observation has label 1—i.e., that it was painted by Van Gogh.

In contrast, generative modeling doesn't require the dataset to be labeled because it concerns itself with generating entirely new images, rather than trying to predict a label of a given image.

Let's define these types of modeling formally, using mathematical notation:

Discriminative modeling estimates $p\,y(\ .|\ )$

That is, discriminative modeling aims to model the probability of a label y given some observation .

Generative modeling estimates $p\ (.\ )$

That is, generative modeling aims to model the probability of observing an observation . Sampling from this distribution allows us to generate new observations.

Conditional Generative Models

Note that we can also build a generative model to model the conditional probabil(ty|p)y —the probability of seeing an observation with a specific label y.

For example, if our dataset contains different types of fruit, we could tell our generative model to specifically generate an image of an apple.

An important point to note is that even if we were able to build a perfect discriminative model to identify Van Gogh paintings, it would still have no idea how to create a painting that looks like a Van Gogh. It can only output probabilities against existing images, as this is what it has been trained to do. We would instead need to train a generative model and sample from this model to generate images that have a high chance of belonging to the original training dataset.

The Rise of Generative Modeling

Until recently, discriminative modeling has been the driving force behind most progress in machine learning. This is because for any discriminative problem, the corresponding generative modeling problem is typically much more difficult to tackle. For example, it is much easier to train a model to predict if a painting is by Van Gogh than it is to train a model to generate a Van Gogh–style painting from scratch.

Similarly, it is much easier to train a model to predict if a page of text was written by Charles Dickens than it is to build a model to generate a set of paragraphs in the style of Dickens. Until recently, most generative challenges were simply out of reach and many doubted that they could ever be solved. Creativity was considered a purely human capability that couldn't be rivaled by AI.

However, as machine learning technologies have matured, this assumption has gradually weakened. In the last 10 years many of the most interesting advancements in the field have come through novel applications of machine learning to generative modeling tasks. For example, Figure 1-3 shows the striking progress that has already been made in facial image generation since 2014.

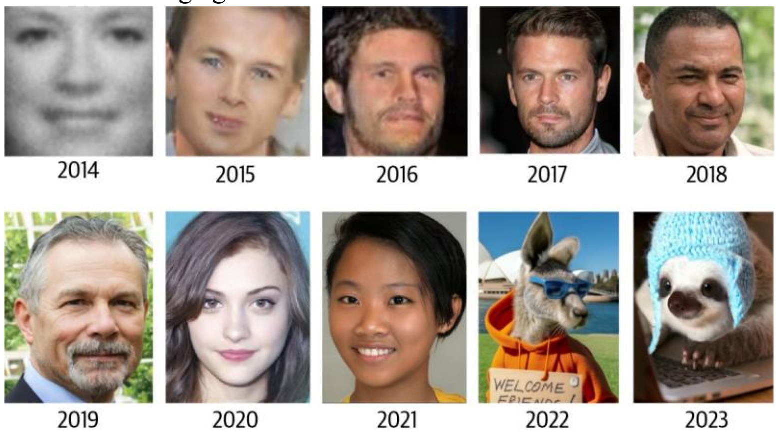


Figure 1-3. Face generation using generative modeling has improved significantly over the last decade (adapted from Brundage et al., 2018)[1]

As well as being easier to tackle, discriminative modeling has historically been more readily applicable to practical problems across industry than generative modeling. For example, a doctor may benefit from a model that predicts if a given retinal image shows signs of glaucoma, but wouldn't necessarily benefit from a model that can generate novel pictures of the back of an eye.

However, this is also starting to change, with the proliferation of companies offering generative services that target specific business problems. For example, it is now possible to access APIs that generate original blog posts given a particular subject matter, produce a variety of images of your product in any setting you desire, or write social media content and ad copy to match your brand and target message. There are also clear positive applications of generative AI for industries such as game design and cinematography, where models trained to output video and music are beginning to add value.

Generative Modeling and AI

As well as the practical uses of generative modeling (many of which are yet to be discovered), there are three deeper reasons why generative modeling can be considered the key to unlocking a far more sophisticated form of artificial intelligence that goes beyond what discriminative modeling alone can achieve.

Firstly, purely from a theoretical point of view, we shouldn't limit our machine training to simply categorizing data. For completeness, we should also be concerned with training models that capture a more complete understanding of the data distribution, beyond any particular label. This is undoubtedly a more difficult problem to solve, due to the high dimensionality of the space of feasible outputs and the relatively small number of creations that we would class as belonging to the dataset. However, as we shall see, many of the same techniques that have driven development in discriminative modeling, such as deep learning, can be utilized by generative models too.

Secondly, as we shall see in Chapter 12, generative modeling is now being used to drive progress in other fields of AI, such as reinforcement learning (the study of teaching agents to optimize a goal in an environment through trial and error). Suppose we want to train a robot to walk across a given terrain. A traditional approach would be to run many experiments where the agent tries out different strategies in the terrain, or a computer simulation of the terrain. Over time the agent would learn which strategies are more successful than others and therefore gradually improve. A challenge with this approach is that it is fairly inflexible because it is trained to optimize the policy for one particular task. An alternative approach that has recently gained traction is to instead train the agent to learn a *world model* of the environment using a generative model, independent of any particular task. The agent can quickly adapt to new tasks by testing strategies in its own world model, rather than in the real environment, which is often computationally more efficient and does not require retraining from scratch for each new task.

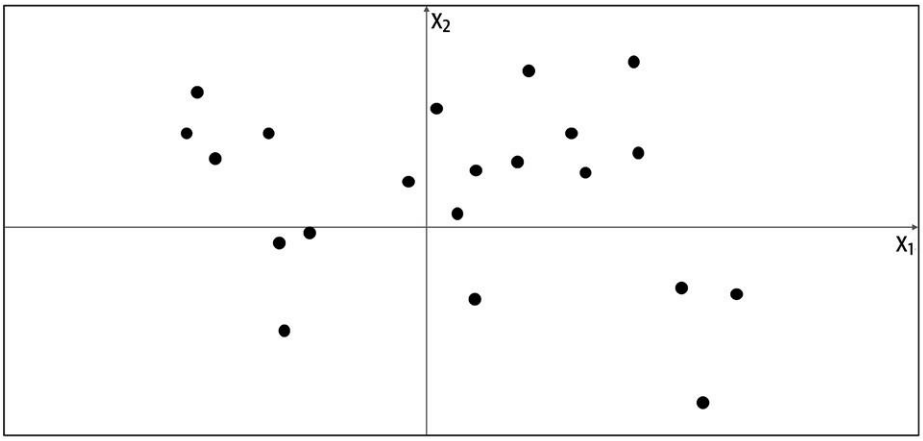
Finally, if we are to truly say that we have built a machine that has acquired a form of intelligence that is comparable to a human's, generative modeling must surely be part of the solution. One of the finest examples of a generative model in the natural world is the person reading this book. Take a moment to consider what an incredible generative model you are. You can close your eyes and imagine what an elephant would look like from any possible angle. You can imagine a number of plausible different endings to your favorite TV show, and you can plan your week ahead by working through various futures in your mind's eye and taking action accordingly. Current neuroscientific theory suggests that our perception of reality is not a highly complex discriminative model operating on our sensory input to produce predictions of what we are experiencing, but is instead a generative model that is trained from birth to produce simulations of our surroundings that accurately match the future. Some theories even suggest that the output from this generative model is what we directly perceive as reality. Clearly, a deep understanding of how we can build machines to acquire this ability will be central to our continued understanding of the workings of the brain and general artificial intelligence.

Our First Generative Model

With this in mind, let's begin our journey into the exciting world of generative modeling. To begin with, we'll look at a toy example of a generative model and introduce some of the ideas that will help us to work through the more complex architectures that we will encounter later in the book.

Hello World!

Let's start by playing a generative modeling game in just two dimensions. I have chosen a rule that has been used to generate the set of points in Figure 1-4. Let's call this rule p_{data} $(\ _1 \quad _2)$ in the space that looks like it has been generated by the same rule.



. Your challenge is to choose a different point $= x , x$

Figure 1-4. A set of points in two dimensions, generated by an unknown rule p_{data}

Where did you choose? You probably used your knowledge of the existing data points to construct a mental model, p_{model}, of whereabouts in the space the point is more likely to be found. In this respect, p_{model} is an *estimate* of p_{data}. Perhaps you decided that p_{model} should look like Figure 1-5—a rectangular box where points may be found, and an area outside of the box where there is no chance of finding any points.

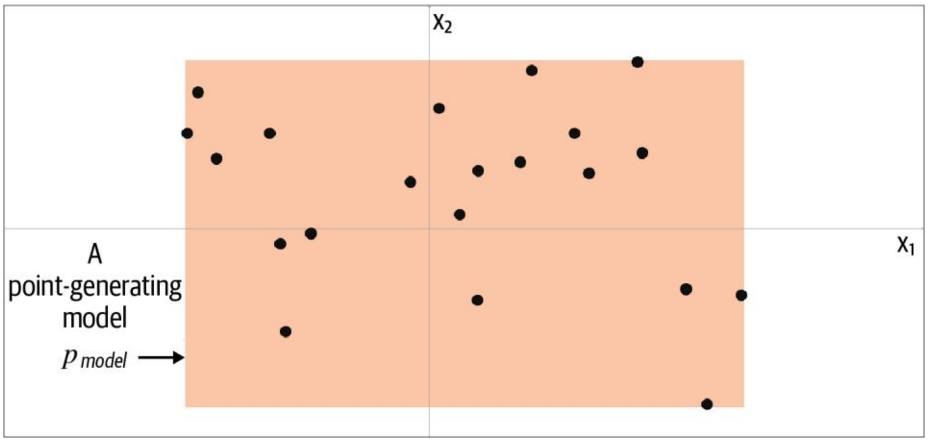


Figure 1-5. The orange box, p_{model}, is an estimate of the true data-generating distribution, p_{data}

To generate a new observation, you can simply choose a point at random within the box, or more formally, *sample* from the distribution p_{model}. Congratulations, you have just built your first generative model! You have used the training data (the black points) to construct a model (the orange region) that you can easily sample from to generate other points that appear to belong to the training set.

Let's now formalize this thinking into a framework that can help us understand what generative modeling is trying to achieve.

The Generative Modeling Framework

We can capture our motivations and goals for building a generative model in the following framework.

The Generative Modeling Framework

- We have a dataset of observations .
- We assume that the observations have been generated according to some unknown distribution, p_{data}.
- We want to build a generative model p_{model} that mimics p_{data}. If we achieve this goal, we can sample from p_{model} to generate observations that appear to have been drawn from p_{data}.

- Therefore, the desirable properties of p_{model} are:

Accuracy

If p_{model} is high for a generated observation, it should look like it has been drawn from p_{data}. If \(p_{model}\) is low for a generated observation, it should *not* look like it has been drawn from p_{data}.

Generation

It should be possible to easily sample a new observation from p_{model}.

Representation

It should be possible to understand how different high-level features in the data are represented by p_{model}.

Let's now reveal the true data-generating distribution, p_{data}, and see how the framework applies to this example. As we can see from Figure 1-6, the data-generating rule is simply a uniform distribution over the land mass of the world, with no chance of finding a point in the sea.

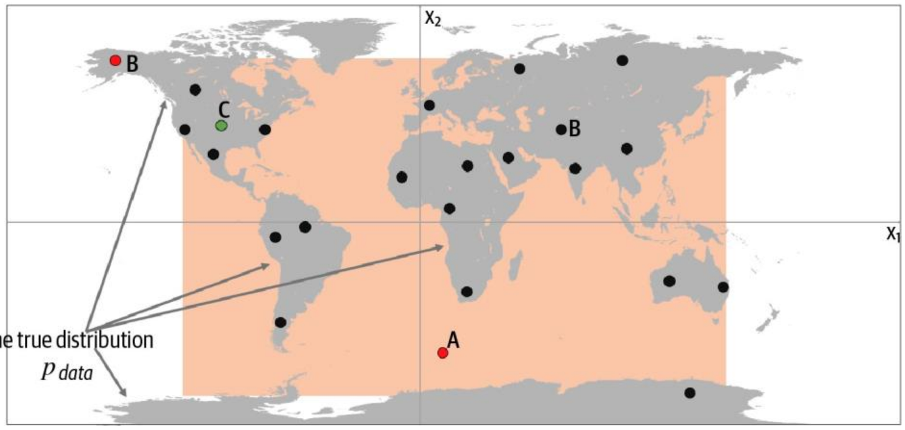


Figure 1-6. The orange box, p_{model}, is an estimate of the true data-generating distribution, p_{data} (the gray area)

Clearly, our model, p_{model}, is an oversimplification of p_{data}. We can inspect points A, B, and C to understand the successes and failures of our model in terms of how accurately it mimics p_{data}:

- Point A is an observation that is generated by our model but does not appear to have been generated by p_{data} as it's in the middle of the sea.

- Point B could never have been generated by p_{model} as it sits outside the orange box. Therefore, our model has some gaps in its ability to produce observations across the entire range of potential possibilities.
- Point C is an observation that could be generated by p_{model} and also by p_{data}.

Despite its shortcomings, the model is easy to sample from, because it is simply a uniform distribution over the orange box. We can easily choose a point at random from inside this box, in order to sample from it.

Also, we can certainly say that our model is a simple representation of the underlying complex distribution that captures some of the underlying high-level features. The true distribution is separated into areas with lots of land mass (continents) and those with no land mass (the sea). This is a high-level feature that is also true of our model, except we have one large continent, rather than many.

This example has demonstrated the fundamental concepts behind generative modeling. The problems we will be tackling in this book will be far more complex and highdimensional, but the underlying framework through which we approach the problem will be the same.

Representation Learning

It is worth delving a little deeper into what we mean by learning a *representation* of the high-dimensional data, as it is a topic that will recur throughout this book.

Suppose you wanted to describe your appearance to someone who was looking for you in a crowd of people and didn't know what you looked like. You wouldn't start by stating the color of pixel 1 of a photo of you, then pixel 2, then pixel 3, etc. Instead, you would make the reasonable assumption that the other person has a general idea of what an average human looks like, then amend this baseline with features that describe groups of pixels, such as *I have very blond hair* or *I wear glasses*. With no more than 10 or so of these statements, the person would be able to map the description back into pixels to generate an image of you in their head. The image wouldn't be perfect, but it would be a close enough likeness to your actual appearance for them to find you among possibly hundreds of other people, even if they've never seen you before.

This is the core idea behind *representation learning*. Instead of trying to model the high-dimensional sample space directly, we describe each observation in the training set using some lower-dimensional *latent space* and then learn a mapping function that can take a point in the latent space and map it to a point in the original domain. In other words, each point in the latent space is a *representation* of some highdimensional observation.

What does this mean in practice? Let's suppose we have a training set consisting of grayscale images of biscuit tins (Figure 1-7).

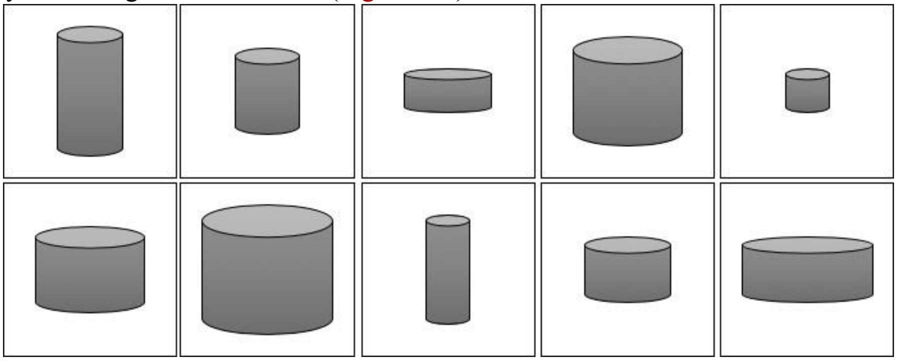


Figure 1-7. The biscuit tin dataset

To us, it is obvious that there are two features that can uniquely represent each of these tins: the height and width of the tin. That is, we can convert each image of a tin to a point in a latent space of just two dimensions, even though the training set of images is provided in high-dimensional pixel space. Notably, this means that we can also produce images of tins that do not exist in the training set, by applying a suitable mapping function *f* to a new point in the latent space, as shown in Figure 1-8. Realizing that the original dataset can be described by the simpler latent space is not so easy for a machine—it would first need to establish that height and width are the two latent space dimensions that best describe this dataset, then learn the mapping function *f* that can take a point in this space and map it to a grayscale biscuit tin image. Machine learning (and specifically, deep learning) gives us the ability to train machines that can find these complex relationships without human guidance.

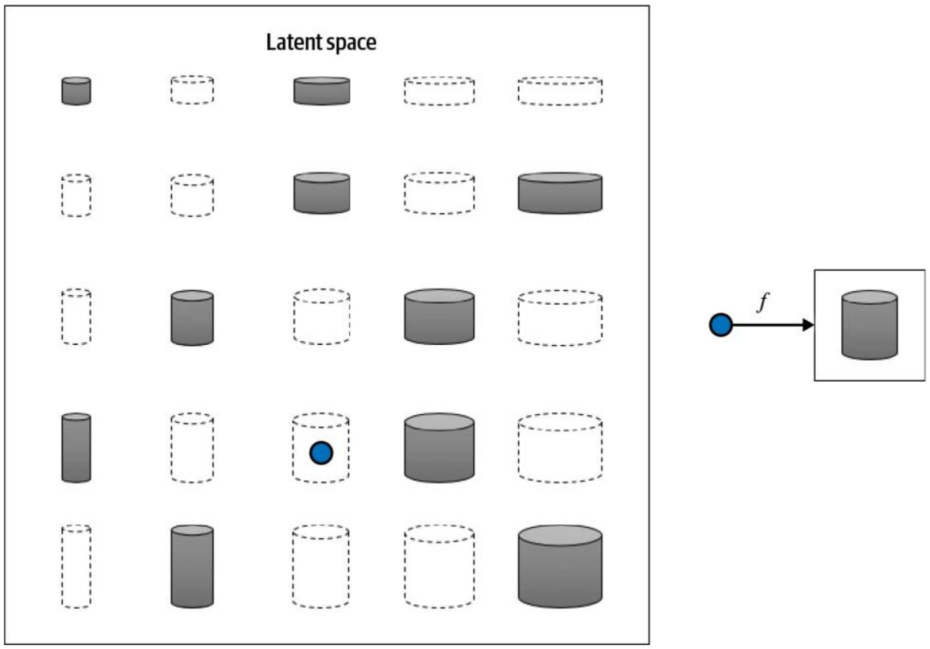


Figure 1-8. The 2D latent space of biscuit tins and the function f that maps a point in the latent space back to the original image domain

One of the benefits of training models that utilize a latent space is that we can perform operations that affect high-level properties of the image by manipulating its representation vector within the more manageable latent space. For example, it is not obvious how to adjust the shading of every single pixel to make an image of a biscuit tin *taller*. However, in the latent space, it's simply a case of increasing the *height* latent dimension, then applying the mapping function to return to the image domain. We shall see an explicit example of this in the next chapter, applied not to biscuit tins but to faces.

The concept of encoding the training dataset into a latent space so that we can sample from it and decode the point back to the original domain is common to many generative modeling techniques, as we shall see in later chapters of this book. Mathematically speaking, *encoder-decoder* techniques try to transform the highly nonlinear *manifold* on which the data lies (e.g., in pixel space) into a simpler latent space that can be sampled from, so that it is likely that any point in the latent space is the representation of a well-formed image, as shown in Figure 1-9.

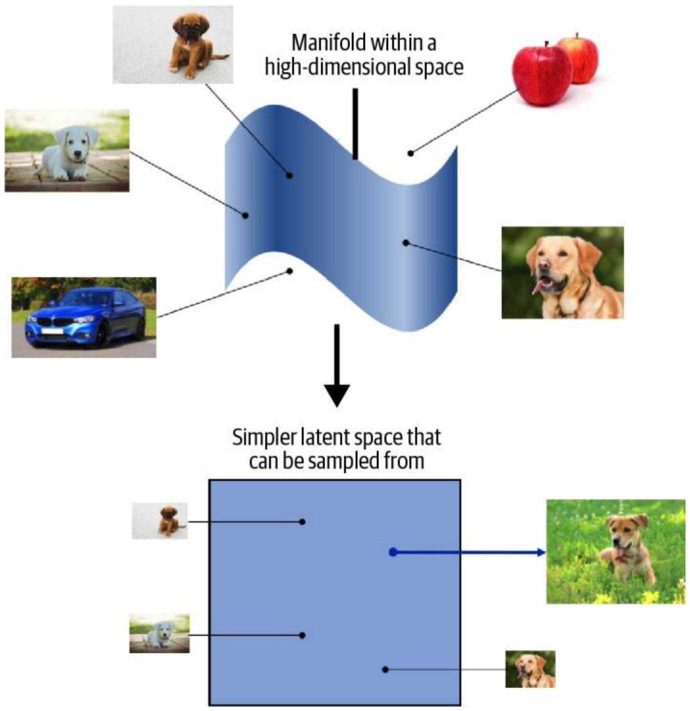

Manifold within a high-dimensional space
Simpler latent space that can be sampled from

Figure 1-9. The dog manifold in high-dimensional pixel space is mapped to a simpler latent space that can be sampled from

Core Probability Theory

We have already seen that generative modeling is closely connected to statistical modeling of probability distributions. Therefore, it now makes sense to introduce some core probabilistic and statistical concepts that will be used throughout this book to explain the theoretical background of each model.

If you have never studied probability or statistics, don't worry. To build many of the deep learning models that we shall see later in this book, it is not essential to have a deep understanding of statistical theory. However, to gain a full appreciation of the task that we are trying to tackle, it's worth trying to build up a solid understanding of basic probabilistic theory. This way, you will have the foundations in place to understand the different families of generative models that will be introduced later in this chapter.

As a first step, we shall define five key terms, linking each one back to our earlier example of a generative model that models the world map in two dimensions:

Sample space

The *sample space* is the complete set of all values an observation can take.

In our previous example, the sample space consists of all points of latitude and longitude $= (x_1, x_2)$ on the world map. For example, $= (40.7306, –73.9352)$ is a point in the sample space (New York City) that belongs to the true data-generating distribution. $= (11.3493, 142.1996)$ is a point in the sample space that does not belong to the true data-generating distribution (it's in the sea).

Probability density function

A *probability density function* (or simply *density function*) is a function(p) that maps a point in the sample space to a number between 0 and 1. The integral of the density function over all points in the sample space must equal 1, so that it is a well-defined probability distribution.

In the world map example, the density function of our generative model is 0 outside of the orange box and constant inside of the box, so that the integral of the density function over the entire sample space equals 1.

While there is only one true density function p_{data}( )that is assumed to have generated the observable dataset, there are infinitely many density functions p_{model} (that) we can use to estimate p_{data} . ( )

Parametric modeling

Parametric modeling is a technique that we can use to structure our approach to finding a suitable p_{model} (A) *parametric model* is a family of density functions ( ) p_θ that can be described using a finite number of parameters, θ.

If we assume a uniform distribution as our model family, then the set all possible boxes we could draw on Figure 1-5 is an example of a parametric model. In this case, there are four parameters: the coordinates of the bottom-left (θ_1, θ_2) and top-right (θ_3, θ_4) corners of the box.

Thus, each density function $p\theta$ (in) this parametric model (i.e., each box) can be uniquely represented by four numbers, $\theta = (\theta_1, \theta_2, \theta_3, \theta_4)$.

Likelihood

The *likelihood* $\mathcal{L}(\theta |$ of$)$ a parameter set θ is a function that measures the plausibility of θ, given some observed point . It is defined as follows:

$$\mathcal{L}(\theta |) = p_\theta \quad ( )$$

That is, the likelihood of θ given some observed point is defined to be the value of the density function parameterized by θ, at the point . If we have a whole dataset of independent observations, then we can write:

$$(|) \quad \mathcal{L}\ \theta = \prod_{\in} p\theta$$

In the world map example, an orange box that only covered the left half of the map would have a likelihood of 0—it couldn't possibly have generated the dataset, as we have observed points in the right half of the map. The orange box in Figure 1-5 has a positive likelihood, as the density function is positive for all data points under this model.

Since the product of a large number of terms between 0 and 1 can be quite computationally difficult to work with, we often use the *log-likelihood* ℓ instead:

$$\ell(\theta |) = \sum_{\in} \log p_\theta \quad ( )$$

There are statistical reasons why the likelihood is defined in this way, but we can also see that this definition intuitively makes sense. The likelihood of a set of parameters θ is defined to be the probability of seeing the data if the true datagenerating distribution was the model parameterized by θ.

Note that the likelihood is a function of the *parameters*, not the data. It should *not* be interpreted as the probability that a given parameter set is correct—in other words, it is not a probability distribution over the parameter space (i.e., it doesn't sum/integrate to 1, with respect to the parameters).

It makes intuitive sense that the focus of parametric modeling should be to find the optimal value $\hat{\theta}$ of the parameter set that maximizes the likelihood of observing the dataset .

Maximum likelihood estimation

Maximum likelihood estimation is the technique that allows us to estimate $\hat{\theta}$—the set of parameters θ of a density function p_θ ( )that is most likely to explain some observed data . More formally:

$$\hat{\theta} = \text{arg max} \ell \; \theta (\; | \;)$$

$\hat{\theta}$ is also called the *maximum likelihood estimate* (MLE).

In the world map example, the MLE is the smallest rectangle that still contains all of the points in the training set.

Neural networks typically *minimize* a loss function, so we can equivalently talk about finding the set of parameters that *minimize the negative log-likelihood*:

$$\hat{\theta} = \text{arg min}_\theta \; (\quad (\; | \;)) \qquad - \ell \; \theta \qquad = \quad (\text{arg}) \quad \text{min}\big(- \log p_\theta \;\; _\theta$$

Generative modeling can be thought of as a form of maximum likelihood estimation, where the parameters θ are the weights of the neural networks contained in the model. We are trying to find the values of these parameters that maximize the likelihood of observing the given data (or equivalently, minimize the negative log-

likelihood).

However, for high-dimensional problems, it is generally not possible to directly calculate(p_θ) —it is *intractable*. As we shall see in the next section, different families of generative models take different approaches to tackling this problem.

Generative Model Taxonomy

While all types of generative models ultimately aim to solve the same task, they all take slightly different approaches to modeling the density function $p(x)$. Broadly speaking, there are three possible approaches:

1. Explicitly model the density function, but constrain the model in some way, so that the density function is tractable (i.e., it can be calculated).
2. Explicitly model a tractable approximation of the density function.
3. Implicitly model the density function, through a stochastic process that directly generates data.

These are shown in Figure 1-10 as a taxonomy, alongside the six families of generative models that we will explore in Part II of this book. Note that these families are not mutually exclusive—there are many examples of models that are hybrids between two different kinds of approaches. You should think of the families as different general approaches to generative modeling, rather than explicit model architectures.

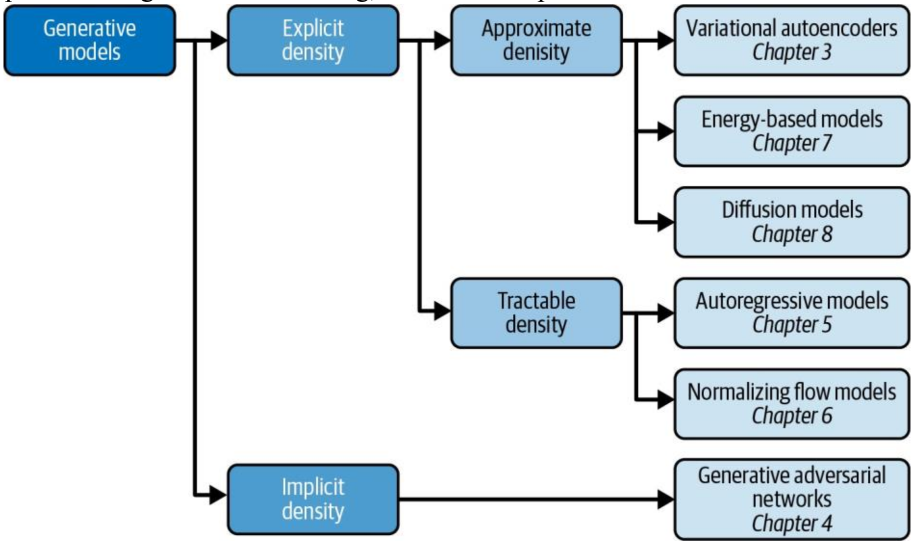

Generative models
Explicit density
Approximate denisity
Variational autoencoders Chapter 3
Energy-based models Chapter 7
Diffusion models Chapter 8
Tractable density
Autoregressive models Chapter 5
Normalizing flow models Chapter 6
Implicit density
Generative adversarial networks Chapter 4

Figure 1-10. A taxonomy of generative modeling approaches

The first split that we can make is between models where the probability density function $p(x)$ is modeled *explicitly* and those where it is modeled *implicitly*.

Implicit density models do not aim to estimate the probability density at all, but instead focus solely on producing a stochastic process that directly generates data. The best-known example of an implicit generative model is a *generative adversarial network*.

We can further split *explicit density models* into those that directly optimize the density function (tractable models) and those that only optimize an approximation of it.

Tractable models place constraints on the model architecture, so that the density function has a form that makes it easy to calculate. For example, *autoregressive models* impose an ordering on the input features, so that the output can be generated sequentially—e.g., word by word, or pixel by pixel. *Normalizing flow models* apply a series of tractable, invertible functions to a simple distribution, in order to generate more complex distributions.

Approximate density models include *variational autoencoders*, which introduce a latent variable and optimize an approximation of the joint density function. *Energy-based models* also utilize approximate methods, but do so via Markov chain sampling, rather than variational methods. *Diffusion models* approximate the density function by training a model to gradually denoise a given image that has been previously corrupted.

A common thread that runs through all of the generative model family types is *deep learning*. Almost all sophisticated generative models have a deep neural network at their core, because they can be trained from scratch to learn the complex relationships that govern the structure of the data, rather than having to be hardcoded with information a priori. We'll explore deep learning in Chapter 2, with practical examples of how to get started building your own deep neural networks.

The Generative Deep Learning Codebase

The final section of this chapter will get you set up to start building generative deep learning models by introducing the codebase that accompanies this book.

Many of the examples in this book are adapted from the excellent open source implementations that are available through the Keras website. I highly recommend you check out this resource, as new models and examples are constantly being added.

Cloning the Repository

To get started, you'll first need to clone the Git repository. *Git* is an open source version control system and will allow you to copy the code locally so that you can run the notebooks on your own machine, or in a cloud-based environment. You may already have this installed, but if not, follow the instructions relevant to your operating system.

To clone the repository for this book, navigate to the folder where you would like to store the files and type the following into your terminal:

```
    git clone
https://github.com/davidADSP/Generative_Deep_Learning_2nd_Edition.git
```

You should now be able to see the files in a folder on your machine.

Using Docker

The codebase for this book is intended to be used with *Docker*, a free containerization technology that makes getting started with a new codebase extremely easy, regardless of your architecture or operating system. If you have never used Docker, don't worry—there is a description of how to get started in the *README* file in the book repository.

Running on a GPU

If you don't have access to your own GPU, that's also no problem! All of the examples in this book will train on a CPU, though this will take longer than if you use a GPUenabled machine. There is also a section in the *README* about setting up a Google Cloud environment that gives you access to a GPU on a pay-as-you-go basis.

Summary

This chapter introduced the field of generative modeling, an important branch of machine learning that complements the more widely studied discriminative modeling. We discussed how generative modeling is currently one of the most active and exciting areas of AI research, with many recent advances in both theory and applications.

We started with a simple toy example and saw how generative modeling ultimately focuses on modeling the underlying distribution of the data. This presents many complex and interesting challenges, which we summarized into a framework for understanding the desirable properties of any generative model.

We then walked through the key probabilistic concepts that will help to fully understand the theoretical foundations of each approach to generative modeling and laid out the six different families of generative models that we will explore in Part II of this book. We also saw how to get started with the *Generative Deep Learning* codebase, by cloning the repository.

In Chapter 2, we will begin our exploration of deep learning and see how to use Keras to build models that can perform discriminative modeling tasks. This will give us the necessary foundation to tackle generative deep learning problems in later chapters.

References

1. Miles Brundage et al., “The Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation,” February 20, 2018, *https://www.eff.org/files/2018/02/20/malicious_ai_report_final.pdf*.

Summary

CHAPTER 2

Deep Learning

Chapter Goals

In this chapter you will:

- Learn about the different types of unstructured data that can be modeled using deep learning.
- Define a deep neural network and understand how it can be used to model complex datasets.
- Build a multilayer perceptron to predict the content of an image.
- Improve the performance of the model by using convolutional layers, dropout, and batch normalization layers.

Let's start with a basic definition of deep learning:

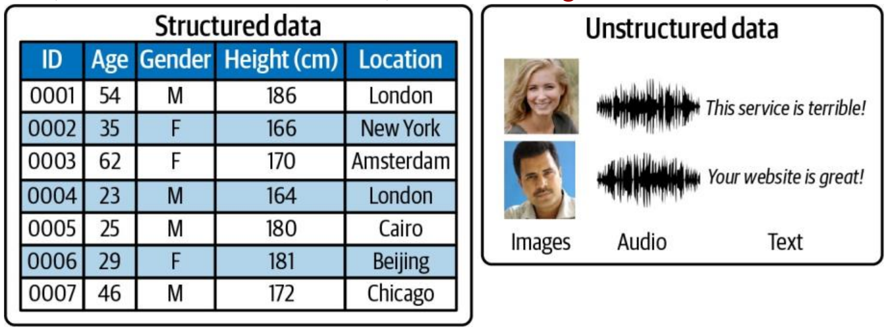
> Deep learning is a class of machine learning algorithms that uses *multiple stacked layers of processing units* to learn high-level representations from *unstructured* data.

To understand deep learning fully, we need to delve into this definition a bit further. First, we'll take a look at the different types of unstructured data that deep learning can be used to model, then we'll dive into the mechanics of building multiple stacked layers of processing units to solve classification tasks. This will provide the foundation for future chapters where we focus on deep learning for generative tasks.

Data for Deep Learning

Many types of machine learning algorithms require *structured*, tabular data as input, arranged into columns of features that describe each observation. For example, a person's age, income, and number of website visits in the last month are all features that could help to predict if the person will subscribe to a particular online service in the coming month. We could use a structured table of these features to train a logistic regression, random forest, or XGBoost model to predict the binary response variable—did the person subscribe (1) or not (0)? Here, each individual feature contains a nugget of information about the observation, and the model would learn how these features interact to influence the response.

Unstructured data refers to any data that is not naturally arranged into columns of features, such as images, audio, and text. There is of course spatial structure to an image, temporal structure to a recording or passage of text, and both spatial and temporal structure to video data, but since the data does not arrive in columns of features, it is considered unstructured, as shown in Figure 2-1.



Structured data

ID	Age	Gender	Height (cm)	Location
0001	54	M	186	London
0002	35	F	166	New York
0003	62	F	170	Amsterdam
0004	23	M	164	London
0005	25	M	180	Cairo
0006	29	F	181	Beijing
0007	46	M	172	Chicago

Figure 2-1. The difference between structured and unstructured data

When our data is unstructured, individual pixels, frequencies, or characters are almost entirely uninformative. For example, knowing that pixel 234 of an image is a muddy shade of brown doesn't really help identify if the image is of a house or a dog, and knowing that character 24 of a sentence is an *e* doesn't help predict if the text is about football or politics.

Pixels or characters are really just the dimples of the canvas into which higher-level informative features, such as an image of a chimney or the word *striker*, are embedded. If the chimney in the image were placed on the other side of the house, the image would still contain a chimney, but this information would now be carried by completely different pixels. If the word *striker* appeared slightly earlier or later in the text, the text would still be about football, but different character positions would provide this information. The granularity of the data combined with the high degree of spatial dependence destroys the concept of the pixel or character as an informative feature in its own right.

For this reason, if we train logistic regression, random forest, or XGBoost models on raw pixel values, the trained model will often perform poorly for all but the simplest of classification tasks. These models rely on the input features to be informative and not spatially dependent. A deep learning model, on the other hand, can learn how to build high-level informative features by itself, directly from the unstructured data.

Deep learning can be applied to structured data, but its real power, especially with regard to generative modeling, comes from its ability to work with unstructured data. Most often, we want to generate unstructured data such as new images or original strings of text, which is why deep learning has had such a profound impact on the field of generative modeling.

Deep Neural Networks

The majority of deep learning systems are *artificial neural networks* (ANNs, or just *neural networks* for short) with multiple stacked hidden layers. For this reason, *deep learning* has now almost become synonymous with *deep neural networks*. However, any system that employs many layers to learn high-level representations of the input data is also a form of deep learning (e.g., deep belief networks).

Let's start by breaking down exactly what we mean by a neural network and then see how they can be used to learn high-level features from unstructured data.

What Is a Neural Network?

A neural network consists of a series of stacked *layers*. Each layer contains *units* that are connected to the previous layer's units through a set of *weights*. As we shall see, there are many different types of layers, but one of the most common is the *fully connected* (or *dense*) layer that connects all units in the layer directly to every unit in the previous layer.

Neural networks where all adjacent layers are fully connected are called *multilayer perceptrons* (MLPs). This is the first type of neural network that we will study. An example of an MLP is shown in Figure 2-2.

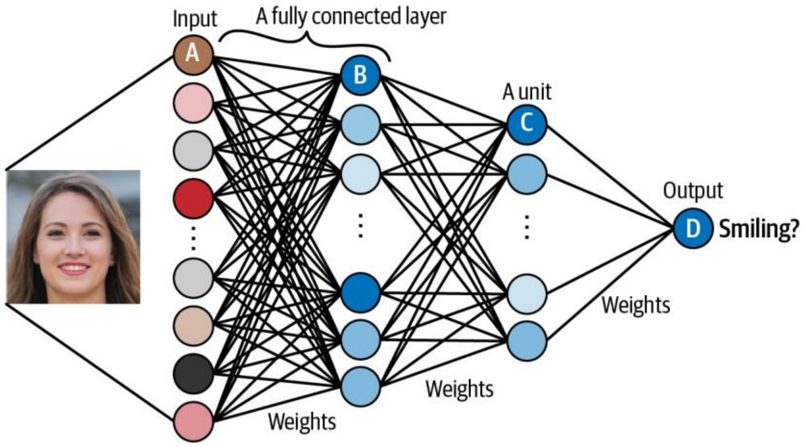


Figure 2-2. An example of a multilayer perceptron that predicts if a face is smiling

The input (e.g., an image) is transformed by each layer in turn, in what is known as a *forward pass* through the network, until it reaches the output layer. Specifically, each unit applies a nonlinear transformation to a weighted sum of its inputs and passes the output through to the subsequent layer. The final output layer is the culmination of this process, where the single unit outputs a probability that the original input belongs to a particular category (e.g., *smiling*).

The magic of deep neural networks lies in finding the set of weights for each layer that results in the most accurate predictions. The process of finding these weights is what we mean by *training* the network.

During the training process, batches of images are passed through the network and the predicted outputs are compared to the ground truth. For example, the network might output a probability of 80% for an image of someone who really is smiling and a probability of 23% for an image of someone who really isn't smiling. A perfect prediction would output 100% and 0% for these examples, so there is a small amount of error. The error in the prediction is then propagated backward through the network, adjusting each set of weights a small amount in the direction that improves the prediction most significantly. This process is appropriately called *backpropagation*. Gradually, each unit becomes skilled at identifying a particular feature that ultimately helps the network to make better predictions.

Learning High-Level Features

The critical property that makes neural networks so powerful is their ability to learn features from the input data, without human guidance. In other words, we do not need to do any feature engineering, which is why neural networks are so useful! We can let the model decide how it wants to arrange its weights, guided only by its desire to minimize the error in its predictions.

For example, let’s walk through the network shown in Figure 2-2, assuming it has already been trained to accurately predict if a given input face is smiling:

1. Unit A receives the value for an individual channel of an input pixel.
2. Unit B combines its input values so that it fires strongest when a particular lowlevel feature such as an edge is present.
3. Unit C combines the low-level features so that it fires strongest when a higherlevel feature such as *teeth* are seen in the image.
4. Unit D combines the high-level features so that it fires strongest when the person in the original image is smiling.

Units in each subsequent layer are able to represent increasingly sophisticated aspects of the original input, by combining lower-level features from the previous layer. Amazingly, this arises naturally out of the training process—we do not need to *tell* each unit what to look for, or whether it should look for high-level features or lowlevel features.

The layers between the input and output layers are called *hidden* layers. While our example only has two hidden layers, deep neural networks can have many more. Stacking large numbers of layers allows the neural network to learn progressively higher-level features by gradually building up information from the lower-level features in previous layers. For example, ResNet,[1] designed for image recognition, contains 152 layers.

Next, we’ll dive straight into the practical side of deep learning and get set up with TensorFlow and Keras so that you can start building your own deep neural networks.

TensorFlow and Keras

TensorFlow is an open source Python library for machine learning, developed by Google. TensorFlow is one of the most utilized frameworks for building machine learning solutions, with particular emphasis on the manipulation of tensors (hence the name). It provides the low-level functionality required to train neural networks, such as computing the gradient of arbitrary differentiable expressions and efficiently executing tensor operations.

Keras is a high-level API for building neural networks, built on top of TensorFlow (Figure 2-3). It is extremely flexible and very user-friendly, making it an ideal choice for getting started with deep learning. Moreover, Keras provides numerous useful building blocks that can be plugged together to create highly complex deep learning architectures through its functional API.

Figure 2-3. TensorFlow and Keras are excellent tools for building deep learning solutions

If you are just getting started with deep learning, I can highly recommend using TensorFlow and Keras. This setup will allow you to build any network that you can think of in a production environment, while also giving you an easy-to-learn API that enables rapid development of new ideas and concepts. Let's start by seeing how easy it is to build a multilayer perceptron using Keras.

Multilayer Perceptron (MLP)

In this section, we will train an MLP to classify a given image using *supervised learning*. Supervised learning is a type of machine learning algorithm in which the computer is trained on a labeled dataset. In other words, the dataset used for training includes input data with corresponding output labels. The goal of the algorithm is to learn a mapping between the input data and the output labels, so that it can make predictions on new, unseen data.

The MLP is a discriminative (rather than generative) model, but supervised learning will still play a role in many types of generative models that we will explore in later chapters of this book, so it is a good place to start our journey.

Running the Code for This Example

The code for this example can be found in the Jupyter notebook located at *notebooks/02_deeplearning/01_mlp/mlp.ipynb* in the book repository.

Preparing the Data

For this example we will be using the CIFAR-10 dataset, a collection of 60,000 32 × 32–pixel color images that comes bundled with Keras out of the box. Each image is classified into exactly one of 10 classes, as shown in Figure 2-4.

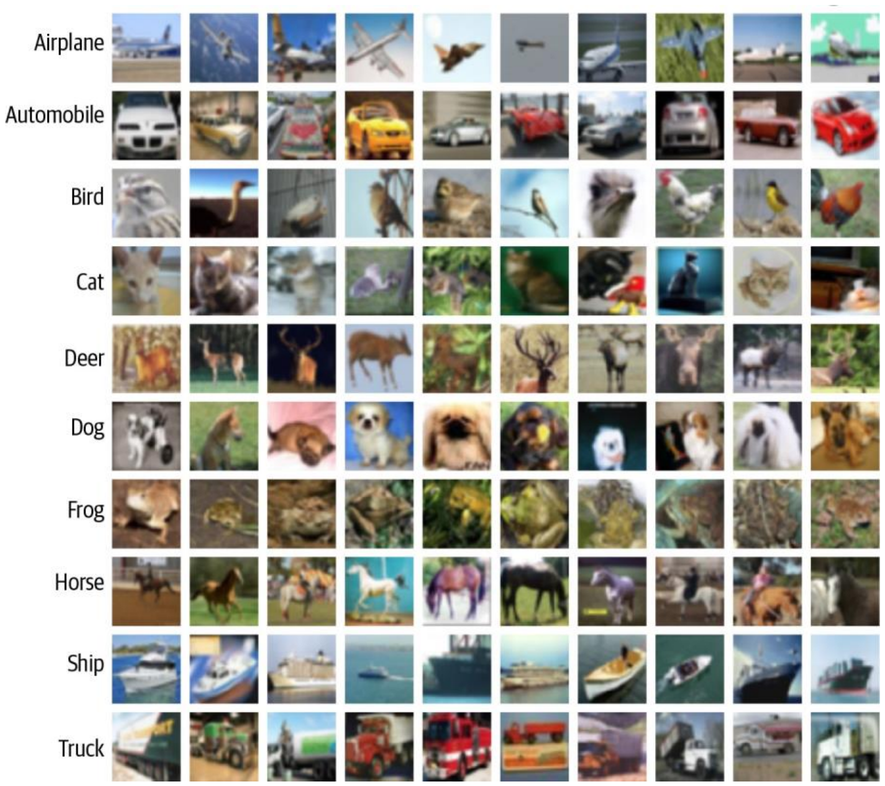


Figure 2-4. Example images from the CIFAR-10 dataset (source: Krizhevsky, 2009)[2]

By default, the image data consists of integers between 0 and 255 for each pixel channel. We first need to preprocess the images by scaling these values to lie between 0 and 1, as neural networks work best when the absolute value of each input is less than 1.

We also need to change the integer labeling of the images to one-hot encoded vectors, because the neural network output will be a probability that the image belongs to each class. If the class integer label of an image is *i*, then its one-hot encoding is a vector of length 10 (the number of classes) that has 0s in all but the *i*th element, which is 1. These steps are shown in Example 2-1. *Example 2-1. Preprocessing the CIFAR-10 dataset*

```
import numpy as np
from tensorflow.keras import datasets, utils

(x_train, y_train), (x_test, y_test) = datasets.cifar10.load_data() ❶
```

```
NUM_CLASSES = 10

x_train = x_train.astype('float32') / 255.0
❷x_test = x_test.astype('float32') / 255.0

y_train = utils.to_categorical(y_train,
NUM_CLASSES) ❸y_test =
utils.to_categorical(y_test, NUM_CLASSES)
```

❶ Load the CIFAR-10 dataset. `x_train` and `x_test` are `numpy` arrays of shape `[50000, 32, 32, 3]` and `[10000, 32, 32, 3]`, respectively. `y_train` and `y_test` are `numpy` arrays of shape `[50000, 1]` and `[10000, 1]`, respectively, containing the integer labels in the range 0 to 9 for the class of each image.

❷ Scale each image so that the pixel channel values lie between 0 and 1.

❸ One-hot encode the labels—the new shapes of `y_train` and `y_test` are `[50000, 10]` and `[10000, 10]`, respectively.

We can see that the training image data (`x_train`) is stored in a *tensor* of shape `[50000, 32, 32, 3]`. There are no *columns* or *rows* in this dataset; instead, this is a tensor with four dimensions. A tensor is just a multidimensional array—it is the natural extension of a matrix to more than two dimensions. The first dimension of this tensor references the index of the image in the dataset, the second and third relate to the size of the image, and the last is the channel (i.e., red, green, or blue, since these are RGB images).

For example, Example 2-2 shows how we can find the channel value of a specific pixel in an image.

Example 2-2. The green channel (1) value of the pixel in the (12,13) position of image 54

```
x_train[54, 12, 13,
1] # 0.36862746
```

Building the Model

In Keras you can either define the structure of a neural network as a `Sequential` model or using the functional API.

A `Sequential` model is useful for quickly defining a linear stack of layers (i.e., where one layer follows on directly from the previous layer without any branching).

We can define our MLP model using the `Sequential` class as shown in Example 2-3.

Example 2-3. Building our MLP using a `Sequential` model

```
from tensorflow.keras import layers, models

model = models.Sequential([
    layers.Flatten(input_shape=(32, 32,
3)),     layers.Dense(200, activation =
'relu'),     layers.Dense(150,
activation = 'relu'),
layers.Dense(10, activation =
'softmax'), ])
```

Many of the models in this book require that the output from a layer is passed to multiple subsequent layers, or conversely, that a layer receives input from multiple preceding layers. For these models, the `Sequential` class is not suitable and we would need to use the functional API instead, which is a lot more flexible.

I recommend that even if you are just starting out building linear models with Keras, you still use the functional API rather than `Sequential` models, since it will serve you better in the long run as your neural networks become more architecturally complex. The functional API will give you complete freedom over the design of your deep neural network.

Example 2-4 shows the same MLP coded using the functional API. When using the functional API, we use the `Model` class to define the overall input and output layers of the model.

Example 2-4. Building our MLP using the functional API

```
from tensorflow.keras import layers, models

input_layer = layers.Input(shape=(32, 32, 3))
x = layers.Flatten()(input_layer)
x = layers.Dense(units=200, activation = 'relu')(x) x
= layers.Dense(units=150, activation = 'relu')(x)
output_layer = layers.Dense(units=10, activation =
'softmax')(x) model = models.Model(input_layer,
output_layer)
```

Both methods give identical models—a diagram of the architecture is shown in Figure 2-5.

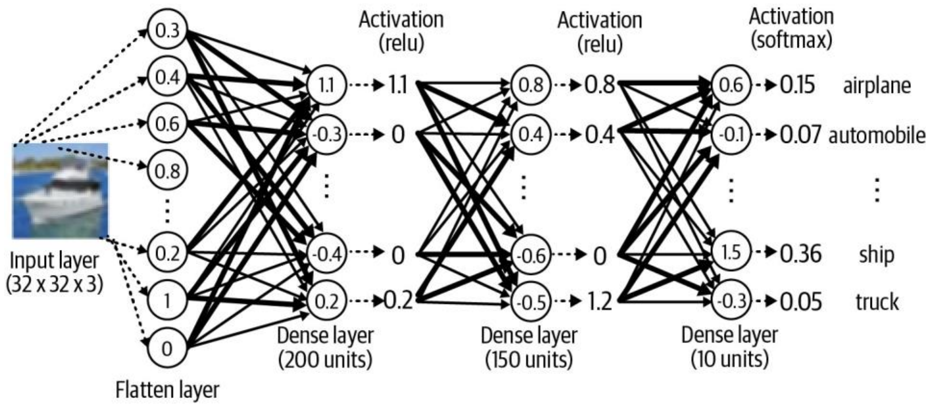


Figure 2-5. A diagram of the MLP architecture

Let's now look in more detail at the different layers and activation functions used within the MLP.

Layers

To build our MLP, we used three different types of layers: `Input`, `Flatten`, and `Dense`.

The `Input` layer is an entry point into the network. We tell the network the shape of each data element to expect as a tuple. Notice that we do not specify the batch size; this isn't necessary as we can pass any number of images into the `Input` layer simultaneously. We do not need to explicitly state the batch size in the `Input` layer definition.

Next we flatten this input into a vector, using a `Flatten` layer. This results in a vector of length 3,072 (= 32 × 32 × 3). The reason we do this is because the subsequent `Dense` layer requires that its input is flat, rather than a multidimensional array. As we shall see later, other layer types require multidimensional arrays as input, so you need to be aware of the required input and output shape of each layer type to understand when it is necessary to use `Flatten`.

The `Dense` layer is one of the most fundamental building blocks of a neural network. It contains a given number of units that are densely connected to the previous layer—that is, every unit in the layer is connected to every unit in the previous layer, through a single connection that carries a weight (which can be positive or negative). The output from a given unit is the weighted sum of the inputs it receives from the previous layer, which is then passed through a nonlinear *activation function* before being sent to the following layer. The activation function is critical to ensure the neural network is able to learn complex functions and doesn't just output a linear combination of its inputs.

Activation functions

There are many kinds of activation function, but three of the most important are ReLU, sigmoid, and softmax.

The *ReLU* (rectified linear unit) activation function is defined to be 0 if the input is negative and is otherwise equal to the input. The *LeakyReLU* activation function is very similar to ReLU, with one key difference: whereas the ReLU activation function returns 0 for input values less than 0, the LeakyReLU function returns a small negative number proportional to the input. ReLU units can sometimes die if they always output 0, because of a large bias toward negative values pre-activation. In this case, the gradient is 0 and therefore no error is propagated back through this unit. LeakyReLU activations fix this issue by always ensuring the gradient is nonzero. ReLU-based functions are among the most reliable activations to use between the layers of a deep network to encourage stable training.

The *sigmoid* activation is useful if you wish the output from the layer to be scaled between 0 and 1—for example, for binary classification problems with one output unit or multilabel classification problems, where each observation can belong to more than one class. Figure 2-6 shows ReLU, LeakyReLU, and sigmoid activation functions side by side for comparison.

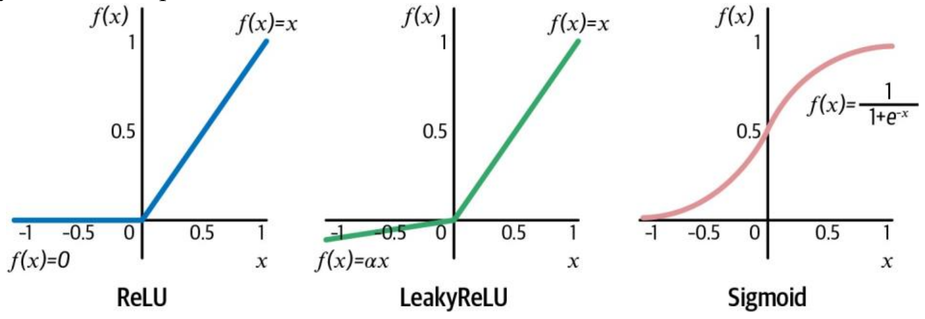


Figure 2-6. The ReLU, LeakyReLU, and sigmoid activation functions

The *softmax* activation function is useful if you want the total sum of the output from the layer to equal 1; for example, for multiclass classification problems where each observation only belongs to exactly one class. It is defined as:

xi

$$y_i = \overline{\sum J\ e\ exj\ j}$$

$$= 1$$

Here, J is the total number of units in the layer. In our neural network, we use a softmax activation in the final layer to ensure that the output is a set of 10 probabilities that sum to 1, which can be interpreted as the likelihood that the image belongs to each class.

In Keras, activation functions can be defined within a layer (Example 2-5) or as a separate layer (Example 2-6).

Example 2-5. A ReLU activation function defined as part of a `Dense` layer

```
x = layers.Dense(units=200, activation = 'relu')(x)
```

Example 2-6. A ReLU activation function defined as its own layer

```
x =
layers.Dense(units=200)(x)
x =
layers.Activation('relu')(
x)
```

In our example, we pass the input through two `Dense` layers, the first with 200 units and the second with 150, both with ReLU activation functions.

Inspecting the model

We can use the `model.summary()` method to inspect the shape of the network at each layer, as shown in Table 2-1.

Table 2-1. Output from the `model.summary()` method

Layer (type)	Output shape	Param #
InputLayer	(None, 32, 32, 3)	0
Flatten	(None, 3072)	0
Dense	(None, 200)	614,600
Dense	(None, 150)	30,150
Dense	(None, 10)	1,510
Total params	646,260	

Trainable params	646,260
Non-trainable params	0

Notice how the shape of our `Input` layer matches the shape of `x_train` and the shape of our `Dense` output layer matches the shape of `y_train`. Keras uses `None` as a marker for the first dimension to show that it doesn't yet know the number of observations that will be passed into the network. In fact, it doesn't need to; we could just as easily pass 1 observation through the network at a time as 1,000. That's because tensor operations are conducted across all observations simultaneously using linear algebra—this is the part handled by TensorFlow. It is also the reason why you get a performance increase when training deep neural networks on GPUs instead of CPUs: GPUs are optimized for large tensor operations since these calculations are also necessary for complex graphics manipulation.

The `summary` method also gives the number of parameters (weights) that will be trained at each layer. If ever you find that your model is training too slowly, check the summary to see if there are any layers that contain a huge number of weights. If so, you should consider whether the number of units in the layer could be reduced to speed up training.

Make sure you understand how the number of parameters is calculated in each layer! It's important to remember that by default, each unit within a given layer is also connected to one additional *bias* unit that always outputs 1. This ensures that the output from the unit can still be nonzero even when all inputs from the previous layer are 0.

Therefore, the number of parameters in the 200-unit `Dense` layer is 200 * (3,072 + 1) = 614,600.

Compiling the Model

In this step, we compile the model with an optimizer and a loss function, as shown in Example 2-7.

Example 2-7. Defining the optimizer and the loss function

```
from tensorflow.keras import optimizers

opt = optimizers.Adam(learning_rate=0.0005)
model.compile(loss='categorical_crossentropy', optimizer=opt,
metrics=['accuracy'])
```

Let's now look in more detail at what we mean by loss functions and optimizers.

Loss functions

The *loss function* is used by the neural network to compare its predicted output to the ground truth. It returns a single number for each observation; the greater this number, the worse the network has performed for this observation.

Keras provides many built-in loss functions to choose from, or you can create your own. Three of the most commonly used are mean squared error, categorical crossentropy, and binary cross-entropy. It is important to understand when it is appropriate to use each.

If your neural network is designed to solve a regression problem (i.e., the output is continuous), then you might use the *mean squared error* loss. This is the mean of the squared difference between the ground truth y_i and predicted value p_i of each output unit, where the mean is taken over all n output units:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - p_i\right)^2$$

If you are working on a classification problem where each observation only belongs to one class, then *categorical cross-entropy* is the correct loss function. This is defined as follows:

$$-\sum_{i=1}^{n} y_i \log\left(p_i\right)$$

Finally, if you are working on a binary classification problem with one output unit, or a multilabel problem where each observation can belong to multiple classes simultaneously, you should use *binary cross-entropy*:

$$-\frac{1}{n}\sum_{i=1}^{n}\left(y_i \log\left(p_i\right) + \left(1 - y_i\right)\log\left(1 - p_i\right)\right)$$

Optimizers

The *optimizer* is the algorithm that will be used to update the weights in the neural network based on the gradient of the loss function. One of the most commonly used and stable optimizers is *Adam* (Adaptive Moment Estimation).[3] In most cases, you shouldn't need to tweak the default parameters of the Adam optimizer, except the *learning rate*. The greater the learning rate, the larger the change in weights at each training step. While training is initially faster with a large learning rate, the downside is that it may result in less stable training and may not find the global minimum of the loss function. This is a parameter that you may want to tune or adjust during

training.

Another common optimizer that you may come across is *RMSProp* (Root Mean Squared Propagation). Again, you shouldn't need to adjust the parameters of this optimizer too much, but it is worth reading the Keras documentation to understand the role of each parameter.

We pass both the loss function and the optimizer into the `compile` method of the model, as well as a `metrics` parameter where we can specify any additional metrics that we would like to report on during training, such as accuracy.

Training the Model

Thus far, we haven't shown the model any data. We have just set up the architecture and compiled the model with a loss function and optimizer.

To train the model against the data, we simply call the `fit` method, as shown in Example 2-8.

Example 2-8. Calling the `fit` method to train the model

```
model.fit(x_tra
in            , ❶
y_train          ❷
          , batch_size =❸32
          , epochs =❹10
          , shuffle = ❺True
          )
```

❶ The raw image data.

❷ The one-hot encoded class labels.

❸ The `batch_size` determines how many observations will be passed to the network at each training step.

❹ The `epochs` determine how many times the network will be shown the full training data.

❺

If `shuffle = True`, the batches will be drawn randomly without replacement from the training data at each training step.

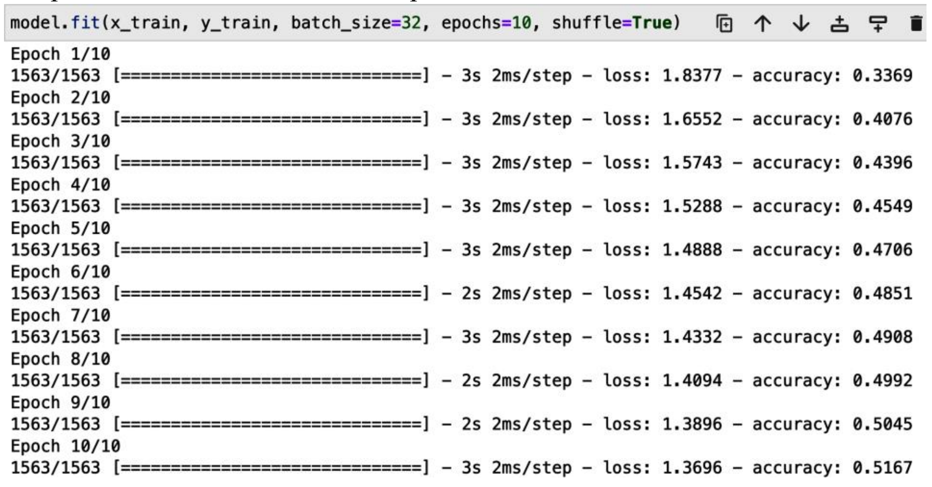
This will start training a deep neural network to predict the category of an image from the CIFAR-10 dataset. The training process works as follows.

First, the weights of the network are initialized to small random values. Then the network performs a series of training steps. At each training step, one *batch* of images is passed through the network and the errors are backpropagated to update the weights. The `batch_size` determines how many images are in each training step batch. The larger the batch size, the more stable the gradient calculation, but the slower each training step.

It would be far too time-consuming and computationally intensive to use the entire dataset to calculate the gradient at each training step, so generally a batch size between 32 and 256 is used. It is also now recommended practice to increase the batch size as training progresses.4

This continues until all observations in the dataset have been seen once. This completes the first *epoch*. The data is then passed through the network again in batches as part of the second epoch. This process repeats until the specified number of epochs have elapsed.

During training, Keras outputs the progress of the procedure, as shown in Figure 2-7. We can see that the training dataset has been split into 1,563 batches (each containing 32 images) and it has been shown to the network 10 times (i.e., over 10 epochs), at a rate of approximately 2 milliseconds per batch. The categorical cross-entropy loss has fallen from 1.8377 to 1.3696, resulting in an accuracy increase from 33.69% after the first epoch to 51.67% after the tenth epoch.



```
model.fit(x_train, y_train, batch_size=32, epochs=10, shuffle=True)

Epoch 1/10
1563/1563 [==============================] - 3s 2ms/step - loss: 1.8377 - accuracy: 0.3369
Epoch 2/10
1563/1563 [==============================] - 3s 2ms/step - loss: 1.6552 - accuracy: 0.4076
Epoch 3/10
1563/1563 [==============================] - 3s 2ms/step - loss: 1.5743 - accuracy: 0.4396
Epoch 4/10
1563/1563 [==============================] - 3s 2ms/step - loss: 1.5288 - accuracy: 0.4549
Epoch 5/10
1563/1563 [==============================] - 3s 2ms/step - loss: 1.4888 - accuracy: 0.4706
Epoch 6/10
1563/1563 [==============================] - 2s 2ms/step - loss: 1.4542 - accuracy: 0.4851
Epoch 7/10
1563/1563 [==============================] - 3s 2ms/step - loss: 1.4332 - accuracy: 0.4908
Epoch 8/10
1563/1563 [==============================] - 2s 2ms/step - loss: 1.4094 - accuracy: 0.4992
Epoch 9/10
1563/1563 [==============================] - 2s 2ms/step - loss: 1.3896 - accuracy: 0.5045
Epoch 10/10
1563/1563 [==============================] - 3s 2ms/step - loss: 1.3696 - accuracy: 0.5167
```

Figure 2-7. The output from the `fit` method

Evaluating the Model

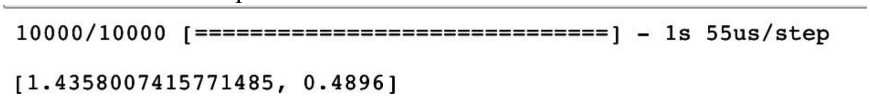
We know the model achieves an accuracy of 51.9% on the training set, but how does it perform on data it has never seen?

To answer this question we can use the `evaluate` method provided by Keras, as shown in Example 2-9.

Example 2-9. Evaluating the model performance on the test set

```
model.evaluate(x_test, y_test)
```

Figure 2-8 shows the output from this method.



```
10000/10000 [==============================] - 1s 55us/step

[1.4358007415771485, 0.4896]
```

Figure 2-8. The output from the `evaluate` method

The output is a list of the metrics we are monitoring: categorical cross-entropy and accuracy. We can see that model accuracy is still 49.0% even on images that it has never seen before. Note that if the model were guessing randomly, it would achieve approximately 10% accuracy (because there are 10 classes), so 49.0% is a good result, given that we have used a very basic neural network.

We can view some of the predictions on the test set using the `predict` method, as shown in Example 2-10.

Example 2-10. Viewing predictions on the test set using the `predict` method

```
CLASSES = np.array(['airplane', 'automobile', 'bird', 'cat', 'deer',
'dog'
                 , 'frog', 'horse', 'ship', 'truck'])

preds = model.predict(x_test) ❶
preds_single = CLASSES[np.argmax(preds, axis = -1)] ❷
actual_single = CLASSES[np.argmax(y_test, axis = -1)]
```

❶ `preds` is an array of shape `[10000, 10]`—i.e., a vector of 10 class probabilities for each observation.

❷ We convert this array of probabilities back into a single prediction using `numpy`'s `argmax` function. Here, `axis = -1` tells the function to collapse the array over the last dimension (the classes dimension), so that the shape of `preds_single` is then `[10000, 1]`.

We can view some of the images alongside their labels and predictions with the code in Example 2-11. As expected, around half are correct.

Example 2-11. Displaying predictions of the MLP against the actual labels

```
import matplotlib.pyplot as plt

n_to_show = 10
indices = np.random.choice(range(len(x_test)), n_to_show)

fig = plt.figure(figsize=(15, 3))
fig.subplots_adjust(hspace=0.4, wspace=0.4) for i,
idx in enumerate(indices):     img = x_test[idx]
    ax = fig.add_subplot(1, n_to_show, i+1)
    ax.axis('off')
    ax.text(0.5, -0.35, 'pred = ' + str(preds_single[idx]),
fontsize=10
        , ha='center', transform=ax.transAxes)
    ax.text(0.5, -0.7, 'act = ' + str(actual_single[idx]), fontsize=10
         , ha='center', transform=ax.transAxes)
ax.imshow(img)
```

Figure 2-9 shows a randomly chosen selection of predictions made by the model, alongside the true labels.

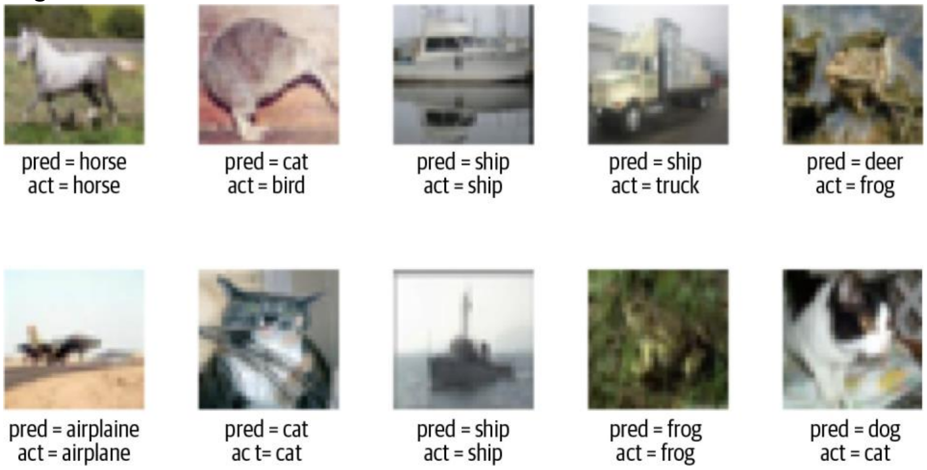


Figure 2-9. Some predictions made by the model, alongside the actual labels

Congratulations! You've just built a multilayer perceptron using Keras and used it to make predictions on new data. Even though this is a supervised learning problem, when we come to building generative models in future chapters many of the core ideas from this chapter (such as loss functions, activation functions, and understanding layer shapes) will still be extremely important. Next we’ll look at ways of improving this model, by introducing a few new layer types.

Convolutional Neural Network (CNN)

One of the reasons our network isn’t yet performing as well as it might is because there isn’t anything in the network that takes into account the spatial structure of the input images. In fact, our first step is to flatten the image into a single vector, so that we can pass it to the first `Dense` layer!

To achieve this we need to use a *convolutional layer*.

Convolutional Layers

First, we need to understand what is meant by a *convolution* in the context of deep learning.

Figure 2-10 shows two different 3 × 3 × 1 portions of a grayscale image being convoluted with a 3 × 3 × 1 *filter* (or *kernel*). The convolution is performed by multiplying the filter pixelwise with the portion of the image, and summing the results. The output is more positive when the portion of the image closely matches the filter and more negative when the portion of the image is the inverse of the filter. The top example resonates strongly with the filter, so it produces a large positive value. The bottom example does not resonate much with the filter, so it produces a value near zero.

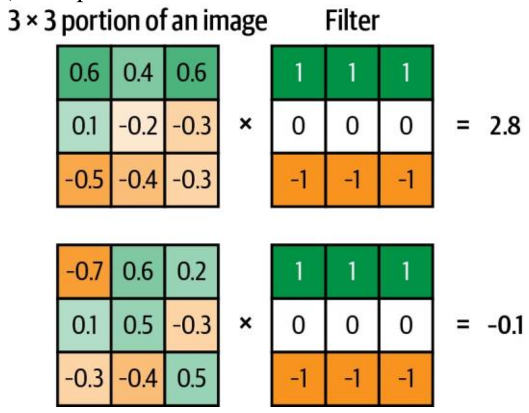


Figure 2-10. A 3 × 3 convolutional filter applied to two portions of a grayscale image

If we move the filter across the entire image from left to right and top to bottom, recording the convolutional output as we go, we obtain a new array that picks out a particular feature of the input, depending on the values in the filter. For example, Figure 2-11 shows two different filters that highlight horizontal and vertical edges.

Running the Code for This Example

You can see this convolutional process worked through manually in the Jupyter notebook located at *notebooks/02_deeplearning/ 02_cnn/convolutions.ipynb* in the book repository.

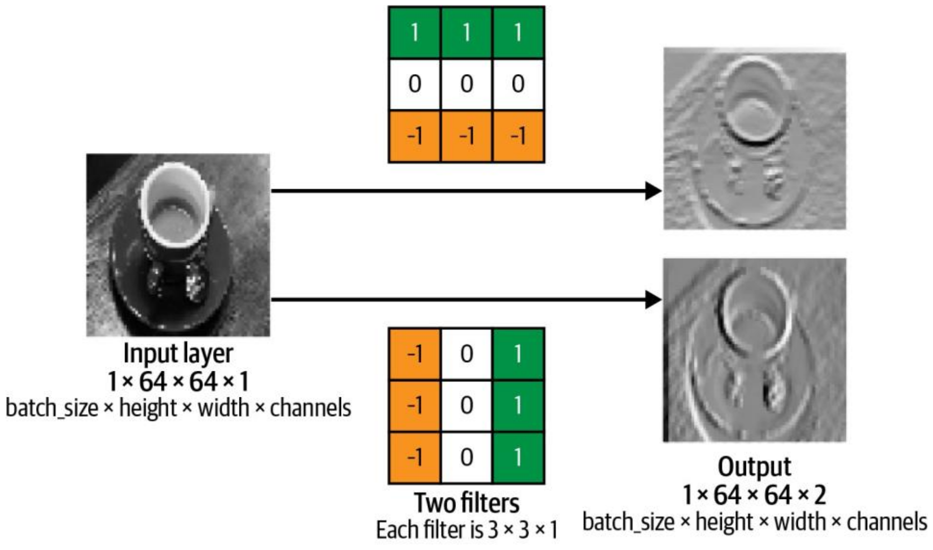


Figure 2-11. Two convolutional filters applied to a grayscale image

A convolutional layer is simply a collection of filters, where the values stored in the filters are the weights that are learned by the neural network through training. Initially these are random, but gradually the filters adapt their weights to start picking out interesting features such as edges or particular color combinations.

In Keras, the `Conv2D` layer applies convolutions to an input tensor with two spatial dimensions (such as an image). For example, the code shown in Example 2-12 builds a convolutional layer with two filters, to match the example in Figure 2-11. *Example 2-12. A `Conv2D` layer applied to grayscale input images*

```
from tensorflow.keras import layers

input_layer = layers.Input(shape=(64,64,1))
conv_layer_1 = layers.Conv2D(
    filters = 2
    , kernel_size = (3,3)
    , strides = 1
    , padding = "same"
    )(input_layer)
```

Next, let's look at two of the arguments to the `Conv2D` layer in more detail—`strides` and `padding`.

Stride

The `strides` parameter is the step size used by the layer to move the filters across the input. Increasing the stride therefore reduces the size of the output tensor. For example, when `strides = 2`, the height and width of the output tensor will be half the size of the input tensor. This is useful for reducing the spatial size of the tensor as it passes through the network, while increasing the number of channels.

Padding

The `padding = "same"` input parameter pads the input data with zeros so that the output size from the layer is exactly the same as the input size when `strides = 1`. Figure 2-12 shows a 3 × 3 kernel being passed over a 5 × 5 input image, with `padding = "same"` and `strides = 1`. The output size from this convolutional layer would also be 5 × 5, as the padding allows the kernel to extend over the edge of the image, so that it fits five times in both directions. Without padding, the kernel could only fit three times along each direction, giving an output size of 3 × 3.

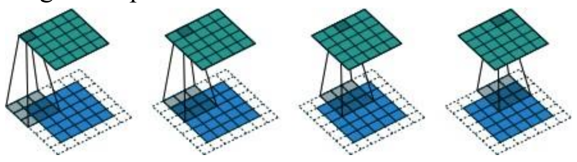


Figure 2-12. A 3 × 3 × 1 kernel (gray) being passed over a 5 × 5 × 1 input image (blue), with `padding = "same"` *and* `strides = 1`*, to generate the 5 × 5 × 1 output (green) (source: Dumoulin and Visin, 2018)5*

Setting `padding = "same"` is a good way to ensure that you are able to easily keep track of the size of the tensor as it passes through many convolutional layers. The shape of the output from a convolutional layer with `padding = "same"` is: *input height*, *input widthstride* , *filters stride*

(———)

Stacking convolutional layers

The output of a `Conv2D` layer is another four-dimensional tensor, now of shape `(batch_size, height, width, filters)`, so we can stack `Conv2D` layers on top of each other to grow the depth of our neural network and make it more powerful. To demonstrate this, let's imagine we are applying `Conv2D` layers to the CIFAR-10 dataset and wish to predict the label of a given image. Note that this time, instead of one input channel (grayscale) we have three (red, green, and blue).

Example 2-13 shows how to build a simple convolutional neural network that we could train to succeed at this task.

Example 2-13. Code to build a convolutional neural network model using Keras

```
from tensorflow.keras import layers, models

input_layer = layers.Input(shape=(32,32,3))
conv_layer_1 = layers.Conv2D(
    filters = 10
    , kernel_size = (4,4)
    , strides = 2
    , padding = 'same'
    )(input_layer) conv_layer_2 =
layers.Conv2D(
    filters = 20
    , kernel_size = (3,3)
    , strides = 2
    , padding = 'same'
    )(conv_layer_1)
flatten_layer = layers.Flatten()(conv_layer_2)
output_layer = layers.Dense(units=10, activation = 'softmax')(flatten_layer) model =
models.Model(input_layer, output_layer)
```

This code corresponds to the diagram shown in Figure 2-13.

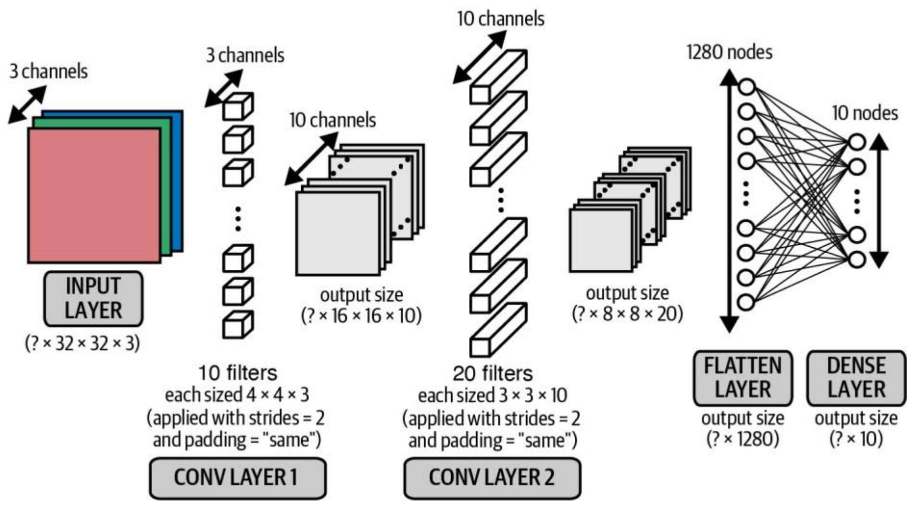


Figure 2-13. A diagram of a convolutional neural network

Note that now that we are working with color images, each filter in the first convolutional layer has a depth of 3 rather than 1 (i.e., each filter has shape 4 × 4 × 3, rather than 4 × 4 × 1). This is to match the three channels (red, green, blue) of the input image. The same idea applies to the filters in the second convolutional layer that have a depth of 10, to match the 10 channels output by the first convolutional layer.

In general, the depth of the filters in a layer is always equal to the number of channels output by the preceding layer.

Inspecting the model

It’s really informative to look at how the shape of the tensor changes as data flows through from one convolutional layer to the next. We can use the `model.summary()` method to inspect the shape of the tensor as it passes through the network (Table 2-2).

Table 2-2. CNN model summary

Layer (type)	Output shape	Param #
InputLayer	(None, 32, 32, 3)	0
Conv2D	(None, 16, 16, 10)	490
Conv2D	(None, 8, 8, 20)	1,820
Flatten	(None, 1280)	0
Dense	(None, 10)	12,810

Total params 15,120

Trainable params 15,120 Non-trainable params 0

Let's walk through our network layer by layer, noting the shape of the tensor as we go:

1. The input shape is `(None, 32, 32, 3)`—Keras uses `None` to represent the fact that we can pass any number of images through the network simultaneously. Since the network is just performing tensor algebra, we don't need to pass images through the network individually, but instead can pass them through together as a batch.
2. The shape of each of the 10 filters in the first convolutional layer is 4 × 4 × 3. This is because we have chosen each filter to have a height and width of 4 (`ker nel_size = (4,4)`) and there are three channels in the preceding layer (red, green, and blue). Therefore, the number of parameters (or weights) in the layer is (4 × 4 × 3 + 1) × 10 = 490, where the + 1 is due to the inclusion of a bias term attached to each of the filters. The output from each filter will be the pixelwise multiplication of the filter weights and the 4 × 4 × 3 section of the image it is covering. As `strides = 2` and `padding = "same"`, the width and height of the output are both halved to 16, and since there are 10 filters the output of the first layer is a batch of tensors each having shape `[16, 16, 10]`.
3. In the second convolutional layer, we choose the filters to be 3 × 3 and they now have depth 10, to match the number of channels in the previous layer. Since there are 20 filters in this layer, this gives a total number of parameters (weights) of (3 × 3 × 10 + 1) × 20 = 1,820. Again, we use `strides = 2 and padding = "same"`, so the width and height both halve. This gives us an overall output shape of `(None, 8, 8, 20)`.
4. We now flatten the tensor using the Keras `Flatten` layer. This results in a set of 8 × 8 × 20 = 1,280 units. Note that there are no parameters to learn in a `Flatten` layer as the operation is just a restructuring of the tensor.
5. We finally connect these units to a 10-unit `Dense` layer with softmax activation, which represents the probability of each category in a 10-category classification task. This creates an extra 1,280 × 10 = 12,810 parameters (weights) to learn.

This example demonstrates how we can chain convolutional layers together to create a convolutional neural network. Before we see how this compares in accuracy to our densely connected neural network, we'll examine two more techniques that can also improve performance: batch normalization and dropout.

Batch Normalization

One common problem when training a deep neural network is ensuring that the weights of the network remain within a reasonable range of values—if they start to become too large, this is a sign that your network is suffering from what is known as the *exploding gradient* problem. As errors are propagated backward through the network, the calculation of the gradient in the earlier layers can sometimes grow exponentially large, causing wild fluctuations in the weight values.

If your loss function starts to return `NaN`, chances are that your weights have grown large enough to cause an overflow error.

This doesn't necessarily happen immediately as you start training the network. Sometimes it can be happily training for hours when suddenly the loss function returns `NaN` and your network has exploded. This can be incredibly annoying. To prevent it from happening, you need to understand the root cause of the exploding gradient problem.

Covariate shift

One of the reasons for scaling input data to a neural network is to ensure a stable start to training over the first few iterations. Since the weights of the network are initially randomized, unscaled input could potentially create huge activation values that immediately lead to exploding gradients. For example, instead of passing pixel values from 0–255 into the input layer, we usually scale these values to between –1 and 1.

Because the input is scaled, it's natural to expect the activations from all future layers to be relatively well scaled as well. Initially this may be true, but as the network trains and the weights move further away from their random initial values, this assumption can start to break down. This phenomenon is known as *covariate shift*.

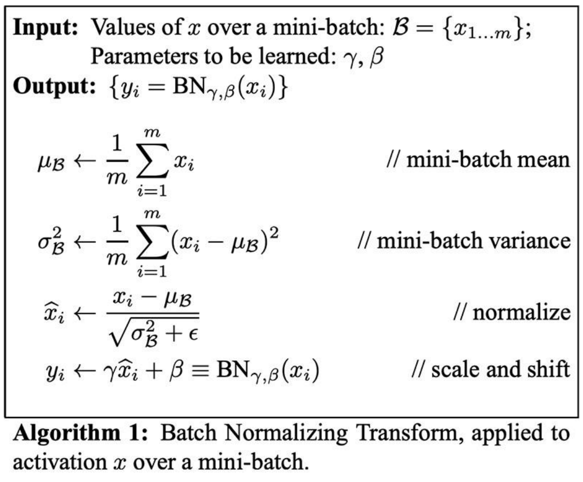
Covariate Shift Analogy

Imagine you're carrying a tall pile of books, and you get hit by a gust of wind. You move the books in a direction opposite to the wind to compensate, but as you do so, some of the books shift, so that the tower is slightly more unstable than before. Initially, this is OK, but with every gust the pile becomes more and more unstable, until eventually the books have shifted so much that the pile collapses. This is covariate shift.

Relating this to neural networks, each layer is like a book in the pile. To remain stable, when the network updates the weights, each layer implicitly assumes that the distribution of its input from the layer beneath is approximately consistent across iterations. However, since there is nothing to stop any of the activation distributions shifting significantly in a certain direction, this can sometimes lead to runaway weight values and an overall collapse of the network.

Training using batch normalization

Batch normalization is a technique that drastically reduces this problem. The solution is surprisingly simple. During training, a batch normalization layer calculates the mean and standard deviation of each of its input channels across the batch and normalizes by subtracting the mean and dividing by the standard deviation. There are then two learned parameters for each channel, the scale (gamma) and shift (beta). The output is simply the normalized input, scaled by gamma and shifted by beta.
Figure 2-14 shows the whole process.



```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                          // mini-batch mean
$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$        // mini-batch variance
$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                       // normalize
$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$          // scale and shift
```

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Figure 2-14. The batch normalization process (source: Ioffe and Szegedy, 2015)[6]

We can place batch normalization layers after dense or convolutional layers to normalize the output.

Referring to our previous example, it's a bit like connecting the layers of books with small sets of adjustable springs that ensure there aren't any overall huge shifts in their positions over time.

Prediction using batch normalization

You might be wondering how this layer works at prediction time. When it comes to prediction, we may only want to predict a single observation, so there is no *batch* over which to calculate the mean and standard deviation. To get around this problem, during training a batch normalization layer also calculates the moving average of the mean and standard deviation of each channel and stores this value as part of the layer to use at test time.

How many parameters are contained within a batch normalization layer? For every channel in the preceding layer, two weights need to be learned: the scale (gamma) and shift (beta). These are the *trainable* parameters. The moving average and standard deviation also need to be calculated for each channel, but since they are derived from the data passing through the layer rather than trained through backpropagation, they are called *nontrainable* parameters. In total, this gives four parameters for each channel in the preceding layer, where two are trainable and two are nontrainable.

In Keras, the `BatchNormalization` layer implements the batch normalization functionality, as shown in Example 2-14.

Example 2-14. A `BatchNormalization` layer in Keras

```
from tensorflow.keras import layers layers.BatchNormalization(momentum =
0.9)
```

The `momentum` parameter is the weight given to the previous value when calculating the moving average and moving standard deviation.

Dropout

When studying for an exam, it is common practice for students to use past papers and sample questions to improve their knowledge of the subject material. Some students try to memorize the answers to these questions, but then come unstuck in the exam because they haven't truly understood the subject matter. The best students use the practice material to further their general understanding, so that they are still able to answer correctly when faced with new questions that they haven't seen before.

The same principle holds for machine learning. Any successful machine learning algorithm must ensure that it generalizes to unseen data, rather than simply *remembering* the training dataset. If an algorithm performs well on the training dataset, but not the test dataset, we say that it is suffering from *overfitting*. To counteract this problem, we use *regularization* techniques, which ensure that the model is penalized if it starts to overfit.

There are many ways to regularize a machine learning algorithm, but for deep learning, one of the most common is by using *dropout* layers. This idea was introduced by Hinton et al. in 2012[7] and presented in a 2014 paper by Srivastava et al.[8]

Dropout layers are very simple. During training, each dropout layer chooses a random set of units from the preceding layer and sets their output to 0, as shown in Figure 2-15.

Incredibly, this simple addition drastically reduces overfitting by ensuring that the network doesn't become overdependent on certain units or groups of units that, in effect, just remember observations from the training set. If we use dropout layers, the network cannot rely too much on any one unit and therefore knowledge is more evenly spread across the whole network.

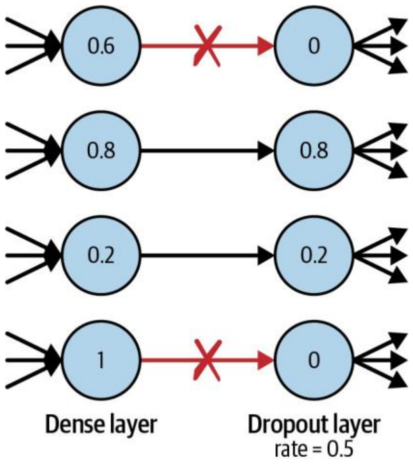


Figure 2-15. A dropout layer

This makes the model much better at generalizing to unseen data, because the network has been trained to produce accurate predictions even under unfamiliar conditions, such as those caused by dropping random units. There are no weights to learn within a dropout layer, as the units to drop are decided stochastically. At prediction time, the dropout layer doesn't drop any units, so that the full network is used to make predictions.

Dropout Analogy

Returning to our analogy, it's a bit like a math student practicing past papers with a random selection of key formulae missing from their formula book. This way, they learn how to answer questions through an understanding of the core principles, rather than always looking up the formulae in the same places in the book. When it comes to test time, they will find it much easier to answer questions that they have never seen before, due to their ability to generalize beyond the training material.

The `Dropout` layer in Keras implements this functionality, with the `rate` parameter specifying the proportion of units to drop from the preceding layer, as shown in Example 2-15.

Example 2-15. A `Dropout` layer in Keras

```
from tensorflow.keras import layers layers.Dropout(rate = 0.25)
```

Dropout layers are used most commonly after dense layers since these are the most prone to overfitting due to the higher number of weights, though you can also use them after convolutional layers.

Batch normalization also has been shown to reduce overfitting, and therefore many modern deep learning architectures don't use dropout at all, relying solely on batch normalization for regularization. As with most deep learning principles, there is no golden rule that applies in every situation—the only way to know for sure what's best is to test different architectures and see which performs best on a holdout set of data.

Building the CNN

You've now seen three new Keras layer types: `Conv2D`, `BatchNormalization`, and `Dropout`. Let's put these pieces together into a CNN model and see how it performs on the CIFAR-10 dataset.

Running the Code for This Example

You can run the following example in the Jupyter notebook in the book repository called *notebooks/02_deeplearning/02_cnn/ cnn.ipynb*.

The model architecture we shall test is shown in Example 2-16. *Example 2-16. Code to build*

a CNN model using Keras

```
from tensorflow.keras import layers, models input_layer =

layers.Input((32,32,3))

x = layers.Conv2D(filters = 32, kernel_size = 3
   , strides = 1, padding = 'same')(input_layer)
x = layers.BatchNormalization()(x) x =
layers.LeakyReLU()(x)

x = layers.Conv2D(filters = 32, kernel_size = 3, strides = 2, padding = 'same')(x)
x = layers.BatchNormalization()(x) x =
layers.LeakyReLU()(x)

x = layers.Conv2D(filters = 64, kernel_size = 3, strides = 1, padding = 'same')(x)
x = layers.BatchNormalization()(x) x =
layers.LeakyReLU()(x)

x = layers.Conv2D(filters = 64, kernel_size = 3, strides = 2, padding = 'same')(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU()(x) x =

layers.Flatten()(x)

x = layers.Dense(128)(x) x = layers.BatchNormalization()(x) x =
layers.LeakyReLU()(x) x = layers.Dropout(rate = 0.5)(x)
output_layer = layers.Dense(10, activation = 'softmax')(x) model =
models.Model(input_layer, output_layer)
```

We use four stacked `Conv2D` layers, each followed by a `BatchNormalization` and a `LeakyReLU` layer. After flattening the resulting tensor, we pass the data through a `Dense` layer of size 128, again followed by a `BatchNormalization` and a `LeakyReLU` layer. This is immediately followed by a `Dropout` layer for regularization, and the network is concluded with an output `Dense` layer of size 10.

The order in which to use the batch normalization and activation layers is a matter of preference. Usually batch normalization layers are placed before the activation, but some successful architectures use these layers the other way around. If you do choose to use batch normalization before activation, you can remember the order using the acronym *BAD* (batch normalization, activation, then dropout)!

The model summary is shown in Table 2-3.

Table 2-3. Model summary of the CNN for CIFAR-10

Layer (type)	Output shape	Param #
InputLayer	(None, 32, 32, 3)	0
Conv2D	(None, 32, 32, 32)	896
BatchNormalization	(None, 32, 32, 32)	128

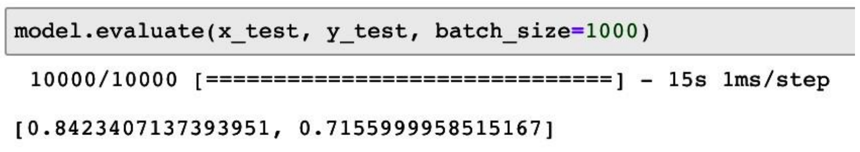
Layer (type)	Output shape	Param #
LeakyReLU	(None, 32, 32, 32)	0
Conv2D	(None, 16, 16, 32)	9,248
BatchNormalization	(None, 16, 16, 32)	128
LeakyReLU	(None, 16, 16, 32)	0
Conv2D	(None, 16, 16, 64)	18,496
BatchNormalization	(None, 16, 16, 64)	256
LeakyReLU	(None, 16, 16, 64)	0
Conv2D	(None, 8, 8, 64)	36,928
BatchNormalization	(None, 8, 8, 64)	256
LeakyReLU	(None, 8, 8, 64)	0
Flatten	(None, 4096)	0
Dense	(None, 128)	524,416
BatchNormalization	(None, 128)	512
LeakyReLU	(None, 128)	0
Dropout	(None, 128)	0
Dense	(None, 10)	1290

Total params	592,554
Trainable params	591,914
Non-trainable params	640

Before moving on, make sure you are able to calculate the output shape and number of parameters for each layer by hand. It's a good exercise to prove to yourself that you have fully understood how each layer is constructed and how it is connected to the preceding layer! Don't forget to include the bias weights that are included as part of the `Conv2D` and `Dense` layers.

Training and Evaluating the CNN

We compile and train the model in exactly the same way as before and call the `evaluate` method to determine its accuracy on the holdout set (Figure 2-16).



```
model.evaluate(x_test, y_test, batch_size=1000)

 10000/10000 [==============================] - 15s 1ms/step

[0.8423407137393951, 0.7155999958515167]
```

Figure 2-16. CNN performance

As you can see, this model is now achieving 71.5% accuracy, up from 49.0% previously. Much better! Figure 2-17 shows some predictions from our new convolutional model.

This improvement has been achieved simply by changing the architecture of the model to include convolutional, batch normalization, and dropout layers. Notice that the number of parameters is actually fewer in our new model than the previous model, even though the number of layers is far greater. This demonstrates the importance of being experimental with your model design and being comfortable with how the different layer types can be used to your advantage. When building generative models, it becomes even more important to understand the inner workings of your model since it is the middle layers of your network that capture the high-level features that you are most interested in.

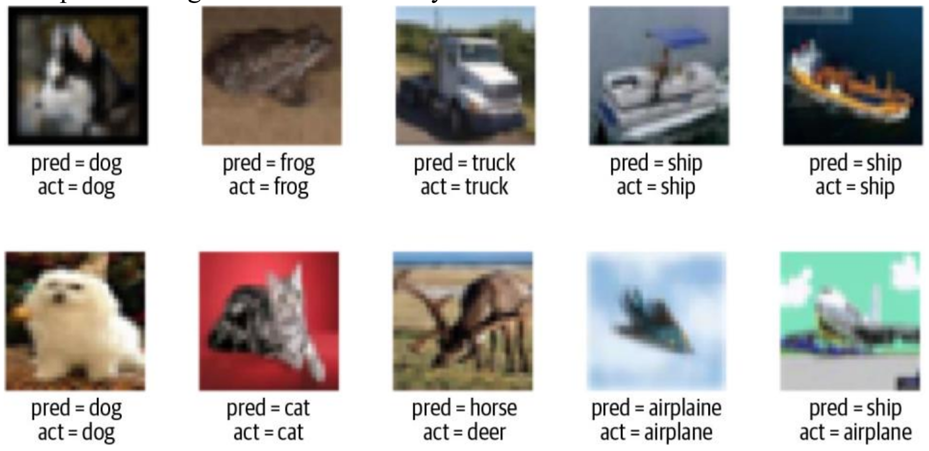


Figure 2-17. CNN predictions

Summary

This chapter introduced the core deep learning concepts that you will need to start building deep generative models. We started by building a multilayer perceptron (MLP) using Keras and trained the model to predict the category of a given image from the CIFAR-10 dataset. Then, we improved upon this architecture by introducing convolutional, batch normalization, and dropout layers to create a convolutional neural network (CNN).

A really important point to take away from this chapter is that deep neural networks are completely flexible by design, and there really are no fixed rules when it comes to model architecture. There are guidelines and best practices, but you should feel free to experiment with layers and the order in which they appear. Don't feel constrained to only use the architectures that you have read about in this book or elsewhere! Like a child with a set of building blocks, the design of your neural network is only limited by your own imagination.

In the next chapter, we shall see how we can use these building blocks to design a network that can generate images. **References**