As you can see, the smaller model starts overfitting later than the reference model (after six epochs rather than four), and its performance degrades more slowly once it starts overfitting.

Now, let's add to our benchmark a model that has much more capacity—far more than the problem warrants. While it is standard to work with models that are significantly overparameterized for what they're trying to learn, there can definitely be such a thing as *too much* memorization capacity. You'll know your model is too large if it starts overfitting right away and if its validation loss curve looks choppy with high-variance (although choppy validation metrics could also be a symptom of using an unreliable validation process, such as a validation split that's too small).

Listing 5.12 Version of the model with higher capacity

```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(512, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_larger_model = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)
```

Figure 5.18 shows how the bigger model fares compared with the reference model.

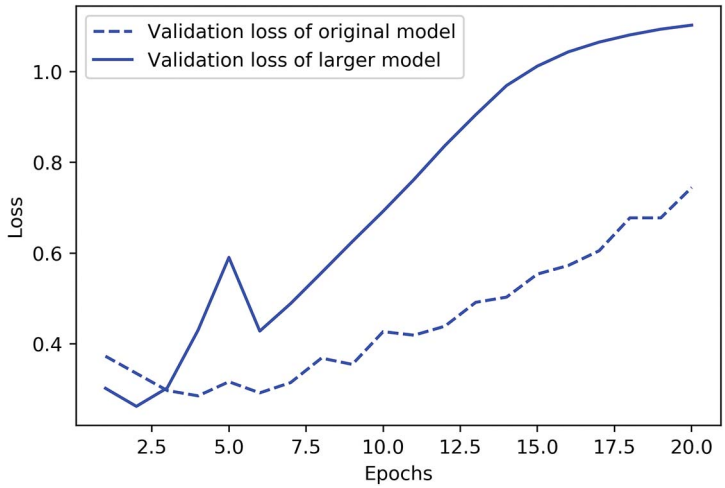

Figure 5.18 Original model vs. much larger model on IMDB review classification

The bigger model starts overfitting almost immediately, after just one epoch, and it overfits much more severely. Its validation loss is also noisier. It gets training loss near zero very quickly. The more capacity the model has, the more quickly it can model the training data (resulting in a low training loss), but the more susceptible it is to overfitting (resulting in a large difference between the training and validation loss).

ADDING WEIGHT REGULARIZATION

You may be familiar with the principle of *Occam's razor*: given two explanations for something, the explanation most likely to be correct is the simplest one—the one that makes fewer assumptions. This idea also applies to the models learned by neural networks: given some training data and a network architecture, multiple sets of weight values (multiple *models*) could explain the data. Simpler models are less likely to overfit than complex ones.

A *simple model* in this context is a model where the distribution of parameter values has less entropy (or a model with fewer parameters, as you saw in the previous section). Thus, a common way to mitigate overfitting is to put constraints on the complexity of a model by forcing its weights to take only small values, which makes the distribution of weight values more *regular*. This is called *weight regularization*, and it's done by adding to the loss function of the model a cost associated with having large weights. This cost comes in two flavors:

- *L1 regularization*—The cost added is proportional to the *absolute value of the weight coefficients* (the *L1 norm* of the weights).
- *L2 regularization*—The cost added is proportional to the *square of the value of the weight coefficients* (the *L2 norm* of the weights). L2 regularization is also called *weight decay* in the context of neural networks. Don't let the different name confuse you: weight decay is mathematically the same as L2 regularization.

In Keras, weight regularization is added by passing *weight regularizer instances* to layers as keyword arguments. Let's add L2 weight regularization to our initial movie-review classification model.

Listing 5.13 Adding L2 weight regularization to the model

```
from tensorflow.keras import regularizers
model = keras.Sequential([
    layers.Dense(16,
                 kernel_regularizer=regularizers.l2(0.002),
                 activation="relu"),
    layers.Dense(16,
                 kernel_regularizer=regularizers.l2(0.002),
                 activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
```

```
history_l2_reg = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)
```

In the preceding listing, `l2(0.002)` means every coefficient in the weight matrix of the layer will add `0.002 * weight_coefficient_value ** 2` to the total loss of the model. Note that because this penalty is *only added at training time*, the loss for this model will be much higher at training than at test time.

Figure 5.19 shows the impact of the L2 regularization penalty. As you can see, the model with L2 regularization has become much more resistant to overfitting than the reference model, even though both models have the same number of parameters.

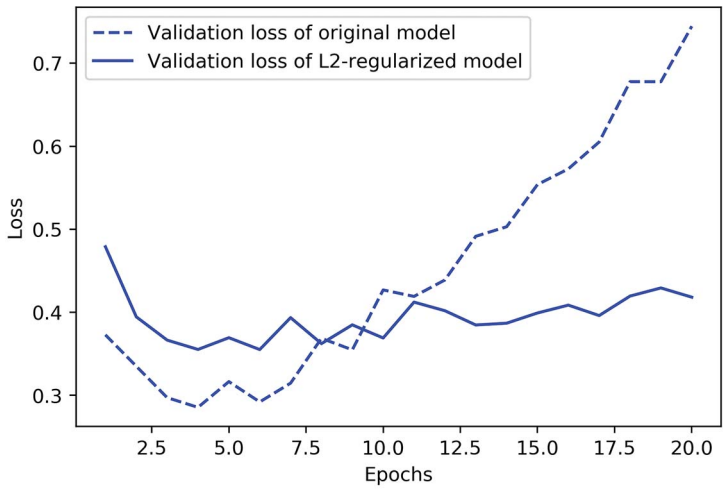

Figure 5.19 Effect of L2 weight regularization on validation loss

As an alternative to L2 regularization, you can use one of the following Keras weight regularizers.

Listing 5.14 Different weight regularizers available in Keras

```
from tensorflow.keras import regularizers
regularizers.l1(0.001)
regularizers.l1_l2(l1=0.001, l2=0.001)
```

L1 regularization

Simultaneous L1 and L2 regularization

Note that weight regularization is more typically used for smaller deep learning models. Large deep learning models tend to be so overparameterized that imposing constraints on weight values hasn't much impact on model capacity and generalization. In these cases, a different regularization technique is preferred: dropout.

ADDING DROPOUT

Dropout is one of the most effective and most commonly used regularization techniques for neural networks; it was developed by Geoff Hinton and his students at the University of Toronto. Dropout, applied to a layer, consists of randomly *dropping out* (setting to zero) a number of output features of the layer during training. Let's say a given layer would normally return a vector `[0.2, 0.5, 1.3, 0.8, 1.1]` for a given input sample during training. After applying dropout, this vector will have a few zero entries distributed at random: for example, `[0, 0.5, 1.3, 0, 1.1]`. The *dropout rate* is the fraction of the features that are zeroed out; it's usually set between 0.2 and 0.5. At test time, no units are dropped out; instead, the layer's output values are scaled down by a factor equal to the dropout rate, to balance for the fact that more units are active than at training time.

Consider a NumPy matrix containing the output of a layer, `layer_output`, of shape `(batch_size, features)`. At training time, we zero out at random a fraction of the values in the matrix:

```
layer_output *= np.random.randint(0, high=2, size=layer_output.shape)
```

At training time, drops out 50% of the units in the output

At test time, we scale down the output by the dropout rate. Here, we scale by 0.5 (because we previously dropped half the units):

```
layer_output *= 0.5
```

At test time

Note that this process can be implemented by doing both operations at training time and leaving the output unchanged at test time, which is often the way it's implemented in practice (see figure 5.20):

At training time

```
layer_output *= np.random.randint(0, high=2, size=layer_output.shape)
layer_output /= 0.5
```

Note that we're scaling up rather than scaling down in this case.

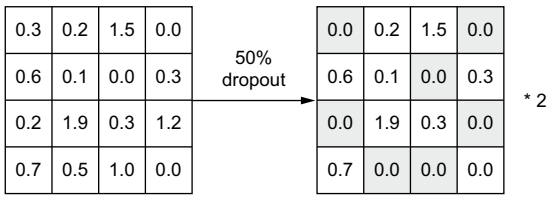

Figure 5.20 Dropout applied to an activation matrix at training time, with rescaling happening during training. At test time the activation matrix is unchanged.

This technique may seem strange and arbitrary. Why would this help reduce overfitting? Hinton says he was inspired by, among other things, a fraud-prevention mechanism used by banks. In his own words, "I went to my bank. The tellers kept changing and I asked one of them why. He said he didn't know but they got moved around a lot.

I figured it must be because it would require cooperation between employees to successfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting." The core idea is that introducing noise in the output values of a layer can break up happenstance patterns that aren't significant (what Hinton refers to as *conspiracies*), which the model will start memorizing if no noise is present.

In Keras, you can introduce dropout in a model via the `Dropout` layer, which is applied to the output of the layer right before it. Let's add two `Dropout` layers in the IMDB model to see how well they do at reducing overfitting.

Listing 5.15 Adding dropout to the IMDB model

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(16, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_dropout = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)
```

Figure 5.21 shows a plot of the results. This is a clear improvement over the reference model—it also seems to be working much better than L2 regularization, since the lowest validation loss reached has improved.

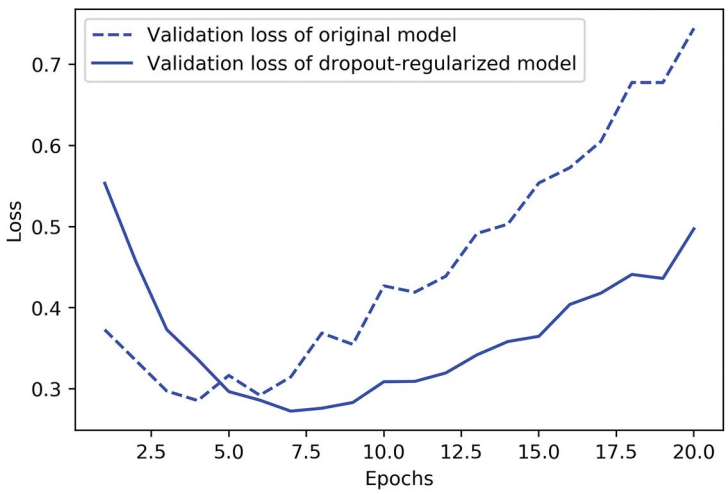

Figure 5.21 Effect of dropout on validation loss

To recap, these are the most common ways to maximize generalization and prevent overfitting in neural networks:

- Get more training data, or better training data.
- Develop better features.
- Reduce the capacity of the model.
- Add weight regularization (for smaller models).
- Add dropout.

Summary

- The purpose of a machine learning model is to *generalize*: to perform accurately on never-before-seen inputs. It's harder than it seems.
- A deep neural network achieves generalization by learning a parametric model that can successfully *interpolate* between training samples—such a model can be said to have learned the "latent manifold" of the training data. This is why deep learning models can only make sense of inputs that are very close to what they've seen during training.
- The fundamental problem in machine learning is *the tension between optimization and generalization*: to attain generalization, you must first achieve a good fit to the training data, but improving your model's fit to the training data will inevitably start hurting generalization after a while. Every single deep learning best practice deals with managing this tension.
- The ability of deep learning models to generalize comes from the fact that they manage to learn to approximate the *latent manifold* of their data, and can thus make sense of new inputs via interpolation.
- It's essential to be able to accurately evaluate the generalization power of your model while you're developing it. You have at your disposal an array of evaluation methods, from simple holdout validation to K-fold cross-validation and iterated K-fold cross-validation with shuffling. Remember to always keep a completely separate test set for final model evaluation, since information leaks from your validation data to your model may have occurred.
- When you start working on a model, your goal is first to achieve a model that has some generalization power and that can overfit. Best practices for doing this include tuning your learning rate and batch size, leveraging better architecture priors, increasing model capacity, or simply training longer.
- As your model starts overfitting, your goal switches to improving generalization through *model regularization*. You can reduce your model's capacity, add dropout or weight regularization, and use early stopping. And naturally, a larger or better dataset is always the number one way to help a model generalize.

6 The universal workflow of machine learning

This chapter covers

- Steps for framing a machine learning problem
- Steps for developing a working model
- Steps for deploying your model in production and maintaining it

Our previous examples have assumed that we already had a labeled dataset to start from, and that we could immediately start training a model. In the real world, this is often not the case. You don't start from a dataset, you start from a problem.

Imagine that you're starting your own machine learning consulting shop. You incorporate, you put up a fancy website, you notify your network. The projects start rolling in:

- A personalized photo search engine for a picture-sharing social network—type in "wedding" and retrieve all the pictures you took at weddings, without any manual tagging needed.
- Flagging spam and offensive text content among the posts of a budding chat app.
- Building a music recommendation system for users of an online radio.
- Detecting credit card fraud for an e-commerce website.

- Predicting display ad click-through rate to decide which ad to serve to a given user at a given time.
- Flagging anomalous cookies on the conveyor belt of a cookie-manufacturing line.
- Using satellite images to predict the location of as-yet unknown archeological sites.

Note on ethics

You may sometimes be offered ethically dubious projects, such as "building an AI that rates the trustworthiness of someone from a picture of their face." First of all, the validity of the project is in doubt: it isn't clear why trustworthiness would be reflected on someone's face. Second, such a task opens the door to all kinds of ethical problems. Collecting a dataset for this task would amount to recording the biases and prejudices of the people who label the pictures. The models you would train on such data would merely encode these same biases into a black-box algorithm that would give them a thin veneer of legitimacy. In a largely tech-illiterate society like ours, "the AI algorithm said this person cannot be trusted" strangely appears to carry more weight and objectivity than "John Smith said this person cannot be trusted," despite the former being a learned approximation of the latter. Your model would be laundering and operationalizing at scale the worst aspects of human judgement, with negative effects on the lives of real people.

Technology is never neutral. If your work has any impact on the world, this impact has a moral direction: technical choices are also ethical choices. Always be deliberate about the values you want your work to support.

It would be very convenient if you could import the correct dataset from `keras.datasets` and start fitting some deep learning models. Unfortunately, in the real world you'll have to start from scratch.

In this chapter, you'll learn about a universal step-by-step blueprint that you can use to approach and solve any machine learning problem, like those in the previous list. This template will bring together and consolidate everything you've learned in chapters 4 and 5, and will give you the wider context that should anchor what you'll learn in the next chapters.

The universal workflow of machine learning is broadly structured in three parts:

1. *Define the task*—Understand the problem domain and the business logic underlying what the customer asked for. Collect a dataset, understand what the data represents, and choose how you will measure success on the task.
2. *Develop a model*—Prepare your data so that it can be processed by a machine learning model, select a model evaluation protocol and a simple baseline to beat, train a first model that has generalization power and that can overfit, and then regularize and tune your model until you achieve the best possible generalization performance.
3. *Deploy the model*—Present your work to stakeholders, ship the model to a web server, a mobile app, a web page, or an embedded device, monitor the model's

performance in the wild, and start collecting the data you'll need to build the next-generation model.

Let's dive in.

6.1 Define the task

You can't do good work without a deep understanding of the context of what you're doing. Why is your customer trying to solve this particular problem? What value will they derive from the solution—how will your model be used, and how will it fit into your customer's business processes? What kind of data is available, or could be collected? What kind of machine learning task can be mapped to the business problem?

6.1.1 Frame the problem

Framing a machine learning problem usually involves many detailed discussions with stakeholders. Here are the questions that should be on the top of your mind:

- What will your input data be? What are you trying to predict? You can only learn to predict something if you have training data available: for example, you can only learn to classify the sentiment of movie reviews if you have both movie reviews and sentiment annotations available. As such, data availability is usually the limiting factor at this stage. In many cases, you will have to resort to collecting and annotating new datasets yourself (which we'll cover in the next section).
- What type of machine learning task are you facing? Is it binary classification? Multiclass classification? Scalar regression? Vector regression? Multiclass, multilabel classification? Image segmentation? Ranking? Something else, like clustering, generation, or reinforcement learning? In some cases, it may be that machine learning isn't even the best way to make sense of the data, and you should use something else, such as plain old-school statistical analysis.
  - The photo search engine project is a multiclass, multilabel classification task.
  - The spam detection project is a binary classification task. If you set "offensive content" as a separate class, it's a three-way classification task.
  - The music recommendation engine turns out to be better handled not via deep learning, but via matrix factorization (collaborative filtering).
  - The credit card fraud detection project is a binary classification task.
  - The click-through-rate prediction project is a scalar regression task.
  - Anomalous cookie detection is a binary classification task, but it will also require an object detection model as a first stage in order to correctly crop out the cookies in raw images. Note that the set of machine learning techniques known as "anomaly detection" would not be a good fit in this setting!
  - The project for finding new archeological sites from satellite images is an image-similarity ranking task: you need to retrieve new images that look the most like known archeological sites.

- What do existing solutions look like? Perhaps your customer already has a handcrafted algorithm that handles spam filtering or credit card fraud detection, with lots of nested `if` statements. Perhaps a human is currently in charge of manually handling the process under consideration—monitoring the conveyor belt at the cookie plant and manually removing the bad cookies, or crafting playlists of song recommendations to be sent out to users who liked a specific artist. You should make sure you understand what systems are already in place and how they work.
- Are there particular constraints you will need to deal with? For example, you could find out that the app for which you're building a spam detection system is strictly end-to-end encrypted, so that the spam detection model will have to live on the end user's phone and must be trained on an external dataset. Perhaps the cookie-filtering model has such latency constraints that it will need to run on an embedded device at the factory rather than on a remote server. You should understand the full context in which your work will fit.

Once you've done your research, you should know what your inputs will be, what your targets will be, and what broad type of machine learning task the problem maps to. Be aware of the hypotheses you're making at this stage:

- You hypothesize that your targets can be predicted given your inputs.
- You hypothesize that the data that's available (or that you will soon collect) is sufficiently informative to learn the relationship between inputs and targets.

Until you have a working model, these are merely hypotheses, waiting to be validated or invalidated. Not all problems can be solved with machine learning; just because you've assembled examples of inputs X and targets Y doesn't mean X contains enough information to predict Y. For instance, if you're trying to predict the movements of a stock on the stock market given its recent price history, you're unlikely to succeed, because price history doesn't contain much predictive information.

6.1.2 Collect a dataset

Once you understand the nature of the task and you know what your inputs and targets are going to be, it's time for data collection—the most arduous, time-consuming, and costly part of most machine learning projects.

- The photo search engine project requires you to first select the set of labels you want to classify—you settle on 10,000 common image categories. Then you need to manually tag hundreds of thousands of your past user-uploaded images with labels from this set.
- For the chat app's spam detection project, because user chats are end-to-end encrypted, you cannot use their contents for training a model. You need to gain access to a separate dataset of tens of thousands of unfiltered social media posts, and manually tag them as spam, offensive, or acceptable.

- For the music recommendation engine, you can just use the "likes" of your users. No new data needs to be collected. Likewise for the click-through-rate prediction project: you have an extensive record of click-through rate for your past ads, going back years.
- For the cookie-flagging model, you will need to install cameras above the conveyor belts to collect tens of thousands of images, and then someone will need to manually label these images. The people who know how to do this currently work at the cookie factory, but it doesn't seem too difficult. You should be able to train people to do it.
- The satellite imagery project will require a team of archeologists to collect a database of existing sites of interest, and for each site you will need to find existing satellite images taken in different weather conditions. To get a good model, you're going to need thousands of different sites.

You learned in chapter 5 that a model's ability to generalize comes almost entirely from the properties of the data it is trained on—the number of data points you have, the reliability of your labels, the quality of your features. A good dataset is an asset worthy of care and investment. If you get an extra 50 hours to spend on a project, chances are that the most effective way to allocate them is to collect more data rather than search for incremental modeling improvements.

The point that data matters more than algorithms was most famously made in a 2009 paper by Google researchers titled "The Unreasonable Effectiveness of Data" (the title is a riff on the well-known 1960 article "The Unreasonable Effectiveness of Mathematics in the Natural Sciences" by Eugene Wigner). This was before deep learning was popular, but, remarkably, the rise of deep learning has only made the importance of data greater.

If you're doing supervised learning, then once you've collected inputs (such as images) you're going to need *annotations* for them (such as tags for those images)—the targets you will train your model to predict. Sometimes, annotations can be retrieved automatically, such as those for the music recommendation task or the click-through-rate prediction task. But often you have to annotate your data by hand. This is a labor-heavy process.

Investing in data annotation infrastructure

Your data annotation process will determine the quality of your targets, which in turn determine the quality of your model. Carefully consider the options you have available:

- Should you annotate the data yourself?
- Should you use a crowdsourcing platform like Mechanical Turk to collect labels?
- Should you use the services of a specialized data-labeling company?

Outsourcing can potentially save you time and money, but it takes away control. Using something like Mechanical Turk is likely to be inexpensive and to scale well, but your annotations may end up being quite noisy.

To pick the best option, consider the constraints you're working with:

- Do the data labelers need to be subject matter experts, or could anyone annotate the data? The labels for a cat-versus-dog image classification problem can be selected by anyone, but those for a dog breed classification task require specialized knowledge. Meanwhile, annotating CT scans of bone fractures pretty much requires a medical degree.
- If annotating the data requires specialized knowledge, can you train people to do it? If not, how can you get access to relevant experts?
- Do you, yourself, understand the way experts come up with the annotations? If you don't, you will have to treat your dataset as a black box, and you won't be able to perform manual feature engineering—this isn't critical, but it can be limiting.

If you decide to label your data in-house, ask yourself what software you will use to record annotations. You may well need to develop that software yourself. Productive data annotation software will save you a lot of time, so it's worth investing in it early in a project.

BEWARE OF NON-REPRESENTATIVE DATA

Machine learning models can only make sense of inputs that are similar to what they've seen before. As such, it's critical that the data used for training should be *representative* of the production data. This concern should be the foundation of all your data collection work.

Suppose you're developing an app where users can take pictures of a plate of food to find out the name of the dish. You train a model using pictures from an image-sharing social network that's popular with foodies. Come deployment time, feedback from angry users starts rolling in: your app gets the answer wrong 8 times out of 10. What's going on? Your accuracy on the test set was well over 90%! A quick look at user-uploaded data reveals that mobile picture uploads of random dishes from random restaurants taken with random smartphones look nothing like the professional-quality, well-lit, appetizing pictures you trained the model on: *your training data wasn't representative of the production data*. That's a cardinal sin—welcome to machine learning hell.

If possible, collect data directly from the environment where your model will be used. A movie review sentiment classification model should be used on new IMDB reviews, not on Yelp restaurant reviews, nor on Twitter status updates. If you want to rate the sentiment of a tweet, start by collecting and annotating actual tweets from a similar set of users as those you're expecting in production. If it's not possible to train on production data, then make sure you fully understand how your training and production data differ, and that you are actively correcting for these differences.

A related phenomenon you should be aware of is *concept drift*. You'll encounter concept drift in almost all real-world problems, especially those that deal with user-generated data. Concept drift occurs when the properties of the production data change over time, causing model accuracy to gradually decay. A music recommendation engine trained in the year 2013 may not be very effective today. Likewise, the IMDB dataset you worked with was collected in 2011, and a model trained on it would

likely not perform as well on reviews from 2020 compared to reviews from 2012, as vocabulary, expressions, and movie genres evolve over time. Concept drift is particularly acute in adversarial contexts like credit card fraud detection, where fraud patterns change practically every day. Dealing with fast concept drift requires constant data collection, annotation, and model retraining.

Keep in mind that machine learning can only be used to memorize patterns that are present in your training data. You can only recognize what you've seen before. Using machine learning trained on past data to predict the future is making the assumption that the future will behave like the past. That often isn't the case.

The problem of sampling bias

A particularly insidious and common case of non-representative data is *sampling bias*. Sampling bias occurs when your data collection process interacts with what you are trying to predict, resulting in biased measurements. A famous historical example occurred in the 1948 US presidential election. On election night, the Chicago Tribune printed the headline "DEWEY DEFEATS TRUMAN." The next morning, Truman emerged as the winner. The editor of the Tribune had trusted the results of a phone survey—but phone users in 1948 were not a random, representative sample of the voting population. They were more likely to be richer, conservative, and to vote for Dewey, the Republican candidate.

"DEWEY DEFEATS TRUMAN": A famous example of sampling bias

Nowadays, every phone survey takes sampling bias into account. That doesn't mean that sampling bias is a thing of the past in political polling—far from it. But unlike in 1948, pollsters are aware of it and take steps to correct it.

6.1.3 Understand your data

It's pretty bad practice to treat a dataset as a black box. Before you start training models, you should explore and visualize your data to gain insights about what makes it predictive, which will inform feature engineering and screen for potential issues.

- If your data includes images or natural language text, take a look at a few samples (and their labels) directly.
- If your data contains numerical features, it's a good idea to plot the histogram of feature values to get a feel for the range of values taken and the frequency of different values.
- If your data includes location information, plot it on a map. Do any clear patterns emerge?
- Are some samples missing values for some features? If so, you'll need to deal with this when you prepare the data (we'll cover how to do this in the next section).
- If your task is a classification problem, print the number of instances of each class in your data. Are the classes roughly equally represented? If not, you will need to account for this imbalance.
- Check for *target leaking*: the presence of features in your data that provide information about the targets and which may not be available in production. If you're training a model on medical records to predict whether someone will be treated for cancer in the future, and the records include the feature "this person has been diagnosed with cancer," then your targets are being artificially leaked into your data. Always ask yourself, is every feature in your data something that will be available in the same form in production?

6.1.4 Choose a measure of success

To control something, you need to be able to observe it. To achieve success on a project, you must first define what you mean by success. Accuracy? Precision and recall? Customer retention rate? Your metric for success will guide all of the technical choices you make throughout the project. It should directly align with your higher-level goals, such as the business success of your customer.

For balanced classification problems, where every class is equally likely, accuracy and the area under a *receiver operating characteristic* (ROC) curve, abbreviated as ROC AUC, are common metrics. For class-imbalanced problems, ranking problems, or multilabel classification, you can use precision and recall, as well as a weighted form of accuracy or ROC AUC. And it isn't uncommon to have to define your own custom metric by which to measure success. To get a sense of the diversity of machine learning success metrics and how they relate to different problem domains, it's helpful to browse the data science competitions on Kaggle (https://kaggle.com); they showcase a wide range of problems and evaluation metrics.

6.2 Develop a model

Once you know how you will measure your progress, you can get started with model development. Most tutorials and research projects assume that this is the only step—skipping problem definition and dataset collection, which are assumed already done, and skipping model deployment and maintenance, which are assumed to be handled by someone else. In fact, model development is only one step in the machine learning workflow, and if you ask me, it's not the most difficult one. The hardest things in machine learning are framing problems and collecting, annotating, and cleaning data. So cheer up—what comes next will be easy in comparison!

6.2.1 Prepare the data

As you've learned before, deep learning models typically don't ingest raw data. Data preprocessing aims at making the raw data at hand more amenable to neural networks. This includes vectorization, normalization, or handling missing values. Many preprocessing techniques are domain-specific (for example, specific to text data or image data); we'll cover those in the following chapters as we encounter them in practical examples. For now, we'll review the basics that are common to all data domains.

VECTORIZATION

All inputs and targets in a neural network must typically be tensors of floating-point data (or, in specific cases, tensors of integers or strings). Whatever data you need to process—sound, images, text—you must first turn into tensors, a step called *data vectorization*. For instance, in the two previous text-classification examples in chapter 4, we started with text represented as lists of integers (standing for sequences of words), and we used one-hot encoding to turn them into a tensor of `float32` data. In the examples of classifying digits and predicting house prices, the data came in vectorized form, so we were able to skip this step.

VALUE NORMALIZATION

In the MNIST digit-classification example from chapter 2, we started with image data encoded as integers in the 0–255 range, encoding grayscale values. Before we fed this data into our network, we had to cast it to `float32` and divide by 255 so we'd end up with floating-point values in the 0–1 range. Similarly, when predicting house prices, we started with features that took a variety of ranges—some features had small floating-point values, and others had fairly large integer values. Before we fed this data into our network, we had to normalize each feature independently so that it had a standard deviation of 1 and a mean of 0.

In general, it isn't safe to feed into a neural network data that takes relatively large values (for example, multi-digit integers, which are much larger than the initial values taken by the weights of a network) or data that is heterogeneous (for example, data where one feature is in the range 0–1 and another is in the range 100–200). Doing so can trigger large gradient updates that will prevent the network

from converging. To make learning easier for your network, your data should have the following characteristics:

- *Take small values*—Typically, most values should be in the 0–1 range.
- *Be homogenous*—All features should take values in roughly the same range.

Additionally, the following stricter normalization practice is common and can help, although it isn't always necessary (for example, we didn't do this in the digit-classification example):

- Normalize each feature independently to have a mean of 0.
- Normalize each feature independently to have a standard deviation of 1.

This is easy to do with NumPy arrays:

```
x -= x.mean(axis=0)
x /= x.std(axis=0)
```

Assuming x is a 2D data matrix of shape (samples, features)

HANDLING MISSING VALUES

You may sometimes have missing values in your data. For instance, in the house-price example, the first feature (the column of index 0 in the data) was the per capita crime rate. What if this feature wasn't available for all samples? You'd then have missing values in the training or test data.

You could just discard the feature entirely, but you don't necessarily have to.

- If the feature is categorical, it's safe to create a new category that means "the value is missing." The model will automatically learn what this implies with respect to the targets.
- If the feature is numerical, avoid inputting an arbitrary value like `"0"`, because it may create a discontinuity in the latent space formed by your features, making it harder for a model trained on it to generalize. Instead, consider replacing the missing value with the average or median value for the feature in the dataset. You could also train a model to predict the feature value given the values of other features.

Note that if you're expecting missing categorial features in the test data, but the network was trained on data without any missing values, the network won't have learned to ignore missing values! In this situation, you should artificially generate training samples with missing entries: copy some training samples several times, and drop some of the categorical features that you expect are likely to be missing in the test data.

6.2.2 Choose an evaluation protocol

As you learned in the previous chapter, the purpose of a model is to achieve generalization, and every modeling decision you will make throughout the model development process will be guided by *validation metrics* that seek to measure generalization performance. The goal of your validation protocol is to accurately estimate what your

success metric of choice (such as accuracy) will be on actual production data. The reliability of that process is critical to building a useful model.

In chapter 5, we reviewed three common evaluation protocols:

- *Maintaining a holdout validation set*—This is the way to go when you have plenty of data.
- *Doing K-fold cross-validation*—This is the right choice when you have too few samples for holdout validation to be reliable.
- *Doing iterated K-fold validation*—This is for performing highly accurate model evaluation when little data is available.

Pick one of these. In most cases, the first will work well enough. As you learned, though, always be mindful of the *representativity* of your validation set, and be careful not to have redundant samples between your training set and your validation set.

6.2.3 Beat a baseline

As you start working on the model itself, your initial goal is to achieve *statistical power*, as you saw in chapter 5: that is, to develop a small model that is capable of beating a simple baseline.

At this stage, these are the three most important things you should focus on:

- *Feature engineering*—Filter out uninformative features (feature selection) and use your knowledge of the problem to develop new features that are likely to be useful.
- *Selecting the correct architecture priors*—What type of model architecture will you use? A densely connected network, a convnet, a recurrent neural network, a Transformer? Is deep learning even a good approach for the task, or should you use something else?
- *Selecting a good-enough training configuration*—What loss function should you use? What batch size and learning rate?

Picking the right loss function

It's often not possible to directly optimize for the metric that measures success on a problem. Sometimes there is no easy way to turn a metric into a loss function; loss functions, after all, need to be computable given only a mini-batch of data (ideally, a loss function should be computable for as little as a single data point) and must be differentiable (otherwise, you can't use backpropagation to train your network). For instance, the widely used classification metric ROC AUC can't be directly optimized. Hence, in classification tasks, it's common to optimize for a proxy metric of ROC AUC, such as crossentropy. In general, you can hope that the lower the crossentropy gets, the higher the ROC AUC will be.

The following table can help you choose a last-layer activation and a loss function for a few common problem types.

(continued)

Choosing the right last-layer activation and loss function for your model

Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`

For most problems, there are existing templates you can start from. You're not the first person to try to build a spam detector, a music recommendation engine, or an image classifier. Make sure you research prior art to identify the feature engineering techniques and model architectures that are most likely to perform well on your task.

Note that it's not always possible to achieve statistical power. If you can't beat a simple baseline after trying multiple reasonable architectures, it may be that the answer to the question you're asking isn't present in the input data. Remember that you're making two hypotheses:

- You hypothesize that your outputs can be predicted given your inputs.
- You hypothesize that the available data is sufficiently informative to learn the relationship between inputs and outputs.

It may well be that these hypotheses are false, in which case you must go back to the drawing board.

6.2.4 Scale up: Develop a model that overfits

Once you've obtained a model that has statistical power, the question becomes, is your model sufficiently powerful? Does it have enough layers and parameters to properly model the problem at hand? For instance, a logistic regression model has statistical power on MNIST but wouldn't be sufficient to solve the problem well. Remember that the universal tension in machine learning is between optimization and generalization. The ideal model is one that stands right at the border between underfitting and overfitting, between undercapacity and overcapacity. To figure out where this border lies, first you must cross it.

To figure out how big a model you'll need, you must develop a model that overfits. This is fairly easy, as you learned in chapter 5:

1. Add layers.
2. Make the layers bigger.
3. Train for more epochs.

Always monitor the training loss and validation loss, as well as the training and validation values for any metrics you care about. When you see that the model's performance on the validation data begins to degrade, you've achieved overfitting.

6.2.5 Regularize and tune your model

Once you've achieved statistical power and you're able to overfit, you know you're on the right path. At this point, your goal becomes to maximize generalization performance.

This phase will take the most time: you'll repeatedly modify your model, train it, evaluate on your validation data (not the test data at this point), modify it again, and repeat, until the model is as good as it can get. Here are some things you should try:

- Try different architectures; add or remove layers.
- Add dropout.
- If your model is small, add L1 or L2 regularization.
- Try different hyperparameters (such as the number of units per layer or the learning rate of the optimizer) to find the optimal configuration.
- Optionally, iterate on data curation or feature engineering: collect and annotate more data, develop better features, or remove features that don't seem to be informative.

It's possible to automate a large chunk of this work by using *automated hyperparameter tuning software*, such as KerasTuner. We'll cover this in chapter 13.

Be mindful of the following: Every time you use feedback from your validation process to tune your model, you leak information about the validation process into the model. Repeated just a few times, this is innocuous; done systematically over many iterations, it will eventually cause your model to overfit to the validation process (even though no model is directly trained on any of the validation data). This makes the evaluation process less reliable.

Once you've developed a satisfactory model configuration, you can train your final production model on all the available data (training and validation) and evaluate it one last time on the test set. If it turns out that performance on the test set is significantly worse than the performance measured on the validation data, this may mean either that your validation procedure wasn't reliable after all, or that you began overfitting to the validation data while tuning the parameters of the model. In this case, you may want to switch to a more reliable evaluation protocol (such as iterated K-fold validation).

6.3 Deploy the model

Your model has successfully cleared its final evaluation on the test set—it's ready to be deployed and to begin its productive life.

6.3.1 Explain your work to stakeholders and set expectations

Success and customer trust are about consistently meeting or exceeding people's expectations. The actual system you deliver is only half of that picture; the other half is setting appropriate expectations before launch.

The expectations of non-specialists towards AI systems are often unrealistic. For example, they might expect that the system "understands" its task and is capable of

exercising human-like common sense in the context of the task. To address this, you should consider showing some examples of the *failure modes* of your model (for instance, show what incorrectly classified samples look like, especially those for which the misclassification seems surprising).

They might also expect human-level performance, especially for processes that were previously handled by people. Most machine learning models, because they are (imperfectly) trained to approximate human-generated labels, do not nearly get there. You should clearly convey model performance expectations. Avoid using abstract statements like "The model has 98% accuracy" (which most people mentally round up to 100%), and prefer talking, for instance, about false negative rates and false positive rates. You could say, "With these settings, the fraud detection model would have a 5% false negative rate and a 2.5% false positive rate. Every day, an average of 200 valid transactions would be flagged as fraudulent and sent for manual review, and an average of 14 fraudulent transactions would be missed. An average of 266 fraudulent transactions would be correctly caught." Clearly relate the model's performance metrics to business goals.

You should also make sure to discuss with stakeholders the choice of key launch parameters—for instance, the probability threshold at which a transaction should be flagged (different thresholds will produce different false negative and false positive rates). Such decisions involve trade-offs that can only be handled with a deep understanding of the business context.

6.3.2 Ship an inference model

A machine learning project doesn't end when you arrive at a Colab notebook that can save a trained model. You rarely put in production the exact same Python model object that you manipulated during training.

First, you may want to export your model to something other than Python:

- Your production environment may not support Python at all—for instance, if it's a mobile app or an embedded system.
- If the rest of the app isn't in Python (it could be in JavaScript, C++, etc.), the use of Python to serve a model may induce significant overhead.

Second, since your production model will only be used to output predictions (a phase called *inference*), rather than for training, you have room to perform various optimizations that can make the model faster and reduce its memory footprint.

Let's take a quick look at the different model deployment options you have available.

DEPLOYING A MODEL AS A REST API

This is perhaps the common way to turn a model into a product: install TensorFlow on a server or cloud instance, and query the model's predictions via a REST API. You could build your own serving app using something like Flask (or any other Python web development library), or use TensorFlow's own library for shipping models as APIs, called *TensorFlow Serving* (www.tensorflow.org/tfx/guide/serving). With TensorFlow Serving, you can deploy a Keras model in minutes.

You should use this deployment setup when

- The application that will consume the model's prediction will have reliable access to the internet (obviously). For instance, if your application is a mobile app, serving predictions from a remote API means that the application won't be usable in airplane mode or in a low-connectivity environment.
- The application does not have strict latency requirements: the request, inference, and answer round trip will typically take around 500 ms.
- The input data sent for inference is not highly sensitive: the data will need to be available on the server in a decrypted form, since it will need to be seen by the model (but note that you should use SSL encryption for the HTTP request and answer).

For instance, the image search engine project, the music recommender system, the credit card fraud detection project, and the satellite imagery project are all good fits for serving via a REST API.

An important question when deploying a model as a REST API is whether you want to host the code on your own, or whether you want to use a fully managed third-party cloud service. For instance, Cloud AI Platform, a Google product, lets you simply upload your TensorFlow model to Google Cloud Storage (GCS), and it gives you an API endpoint to query it. It takes care of many practical details such as batching predictions, load balancing, and scaling.

Deploying a model on a device

Sometimes, you may need your model to live on the same device that runs the application that uses it—maybe a smartphone, an embedded ARM CPU on a robot, or a microcontroller on a tiny device. You may have seen a camera capable of automatically detecting people and faces in the scenes you pointed it at: that was probably a small deep learning model running directly on the camera.

You should use this setup when

- Your model has strict latency constraints or needs to run in a low-connectivity environment. If you're building an immersive augmented reality application, querying a remote server is not a viable option.
- Your model can be made sufficiently small that it can run under the memory and power constraints of the target device. You can use the TensorFlow Model Optimization Toolkit to help with this (www.tensorflow.org/model_optimization).
- Getting the highest possible accuracy isn't mission critical for your task. There is always a trade-off between runtime efficiency and accuracy, so memory and power constraints often require you to ship a model that isn't quite as good as the best model you could run on a large GPU.
- The input data is strictly sensitive and thus shouldn't be decryptable on a remote server.

Our spam detection model will need to run on the end user's smartphone as part of the chat app, because messages are end-to-end encrypted and thus cannot be read by a remotely hosted model. Likewise, the bad-cookie detection model has strict latency constraints and will need to run at the factory. Thankfully, in this case, we don't have any power or space constraints, so we can actually run the model on a GPU.

To deploy a Keras model on a smartphone or embedded device, your go-to solution is TensorFlow Lite (www.tensorflow.org/lite). It's a framework for efficient on-device deep learning inference that runs on Android and iOS smartphones, as well as ARM64-based computers, Raspberry Pi, or certain microcontrollers. It includes a converter that can straightforwardly turn your Keras model into the TensorFlow Lite format.

DEPLOYING A MODEL IN THE BROWSER

Deep learning is often used in browser-based or desktop-based JavaScript applications. While it is usually possible to have the application query a remote model via a REST API, there can be key advantages in having the model run directly in the browser, on the user's computer (utilizing GPU resources if they're available).

Use this setup when

- You want to offload compute to the end user, which can dramatically reduce server costs.
- The input data needs to stay on the end user's computer or phone. For instance, in our spam detection project, the web version and the desktop version of the chat app (implemented as a cross-platform app written in JavaScript) should use a locally run model.
- Your application has strict latency constraints. While a model running on the end user's laptop or smartphone is likely to be slower than one running on a large GPU on your own server, you don't have the extra 100 ms of network round trip.
- You need your app to keep working without connectivity, after the model has been downloaded and cached.

You should only go with this option if your model is small enough that it won't hog the CPU, GPU, or RAM of your user's laptop or smartphone. In addition, since the entire model will be downloaded to the user's device, you should make sure that nothing about the model needs to stay confidential. Be mindful of the fact that, given a trained deep learning model, it is usually possible to recover some information about the training data: better not to make your trained model public if it was trained on sensitive data.

To deploy a model in JavaScript, the TensorFlow ecosystem includes TensorFlow.js (www.tensorflow.org/js), a JavaScript library for deep learning that implements almost all of the Keras API (originally developed under the working name WebKeras) as well as many lower-level TensorFlow APIs. You can easily import a saved Keras model into TensorFlow.js to query it as part of your browser-based JavaScript app or your desktop Electron app.

Inference model optimization

Optimizing your model for inference is especially important when deploying in an environment with strict constraints on available power and memory (smartphones and embedded devices) or for applications with low latency requirements. You should always seek to optimize your model before importing into TensorFlow.js or exporting it to TensorFlow Lite.

There are two popular optimization techniques you can apply:

- *Weight pruning*—Not every coefficient in a weight tensor contributes equally to the predictions. It's possible to considerably lower the number of parameters in the layers of your model by only keeping the most significant ones. This reduces the memory and compute footprint of your model, at a small cost in performance metrics. By deciding how much pruning you want to apply, you are in control of the trade-off between size and accuracy.
- *Weight quantization*—Deep learning models are trained with single-precision floating-point (`float32`) weights. However, it's possible to *quantize* weights to 8-bit signed integers (`int8`) to get an inference-only model that's a quarter the size but remains near the accuracy of the original model.

The TensorFlow ecosystem includes a weight pruning and quantization toolkit (www.tensorflow.org/model_optimization) that is deeply integrated with the Keras API.

6.3.3 Monitor your model in the wild

You've exported an inference model, you've integrated it into your application, and you've done a dry run on production data—the model behaved exactly as you expected. You've written unit tests as well as logging and status-monitoring code—perfect. Now it's time to press the big red button and deploy to production.

Even this is not the end. Once you've deployed a model, you need to keep monitoring its behavior, its performance on new data, its interaction with the rest of the application, and its eventual impact on business metrics.

- Is user engagement in your online radio up or down after deploying the new music recommender system? Has the average ad click-through rate increased after switching to the new click-through-rate prediction model? Consider using *randomized A/B testing* to isolate the impact of the model itself from other changes: a subset of cases should go through the new model, while another control subset should stick to the old process. Once sufficiently many cases have been processed, the difference in outcomes between the two is likely attributable to the model.
- If possible, do a regular manual audit of the model's predictions on production data. It's generally possible to reuse the same infrastructure as for data annotation: send some fraction of the production data to be manually annotated, and compare the model's predictions to the new annotations. For instance, you should definitely do this for the image search engine and the bad-cookie flagging system.

- When manual audits are impossible, consider alternative evaluation avenues such as user surveys (for example, in the case of the spam and offensive-content flagging system).

6.3.4 Maintain your model

Lastly, no model lasts forever. You've already learned about *concept drift*: over time, the characteristics of your production data will change, gradually degrading the performance and relevance of your model. The lifespan of your music recommender system will be counted in weeks. For the credit card fraud detection systems, it will be days. A couple of years in the best case for the image search engine.

As soon as your model has launched, you should be getting ready to train the next generation that will replace it. As such,

- Watch out for changes in the production data. Are new features becoming available? Should you expand or otherwise edit the label set?
- Keep collecting and annotating data, and keep improving your annotation pipeline over time. In particular, you should pay special attention to collecting samples that seem to be difficult for your current model to classify—such samples are the most likely to help improve performance.

This concludes the universal workflow of machine learning—that's a lot of things to keep in mind. It takes time and experience to become an expert, but don't worry, you're already a lot wiser than you were a few chapters ago. You are now familiar with the big picture—the entire spectrum of what machine learning projects entail. While most of this book will focus on model development, you're now aware that it's only one part of the entire workflow. Always keep in mind the big picture!

Summary

- When you take on a new machine learning project, first define the problem at hand:
  - Understand the broader context of what you're setting out to do—what's the end goal and what are the constraints?
  - Collect and annotate a dataset; make sure you understand your data in depth.
  - Choose how you'll measure success for your problem—what metrics will you monitor on your validation data?
- Once you understand the problem and you have an appropriate dataset, develop a model:
  - Prepare your data.
  - Pick your evaluation protocol: holdout validation? K-fold validation? Which portion of the data should you use for validation?
  - Achieve statistical power: beat a simple baseline.
  - Scale up: develop a model that can overfit.

  - Regularize your model and tune its hyperparameters, based on performance on the validation data. A lot of machine learning research tends to focus only on this step, but keep the big picture in mind.
- When your model is ready and yields good performance on the test data, it's time for deployment:
  - First, make sure you set appropriate expectations with stakeholders.
  - Optimize a final model for inference, and ship a model to the deployment environment of choice—web server, mobile, browser, embedded device, etc.
  - Monitor your model's performance in production, and keep collecting data so you can develop the next generation of the model.

7 Working with Keras: A deep dive

This chapter covers

- Creating Keras models with the `Sequential` class, the Functional API, and model subclassing
- Using built-in Keras training and evaluation loops
- Using Keras callbacks to customize training
- Using TensorBoard to monitor training and evaluation metrics
- Writing training and evaluation loops from scratch

You've now got some experience with Keras—you're familiar with the Sequential model, `Dense` layers, and built-in APIs for training, evaluation, and inference—`compile()`, `fit()`, `evaluate()`, and `predict()`. You even learned in chapter 3 how to inherit from the `Layer` class to create custom layers, and how to use the TensorFlow `GradientTape` to implement a step-by-step training loop.

In the coming chapters, we'll dig into computer vision, timeseries forecasting, natural language processing, and generative deep learning. These complex applications will require much more than a `Sequential` architecture and the default `fit()` loop. So let's first turn you into a Keras expert! In this chapter, you'll get a complete overview of the key ways to work with Keras APIs: everything

you're going to need to handle the advanced deep learning use cases you'll encounter next.

7.1 A spectrum of workflows

The design of the Keras API is guided by the principle of *progressive disclosure of complexity*: make it easy to get started, yet make it possible to handle high-complexity use cases, only requiring incremental learning at each step. Simple use cases should be easy and approachable, and arbitrarily advanced workflows should be *possible*: no matter how niche and complex the thing you want to do, there should be a clear path to it. A path that builds upon the various things you've learned from simpler workflows. This means that you can grow from beginner to expert and still use the same tools—only in different ways.

As such, there's not a single "true" way of using Keras. Rather, Keras offers a *spectrum of workflows*, from the very simple to the very flexible. There are different ways to build Keras models, and different ways to train them, answering different needs. Because all these workflows are based on shared APIs, such as `Layer` and `Model`, components from any workflow can be used in any other workflow—they can all talk to each other.

7.2 Different ways to build Keras models

There are three APIs for building models in Keras (see figure 7.1):

- The *Sequential model*, the most approachable API—it's basically a Python list. As such, it's limited to simple stacks of layers.
- The *Functional API*, which focuses on graph-like model architectures. It represents a nice mid-point between usability and flexibility, and as such, it's the most commonly used model-building API.
- *Model subclassing*, a low-level option where you write everything yourself from scratch. This is ideal if you want full control over every little thing. However, you won't get access to many built-in Keras features, and you will be more at risk of making mistakes.

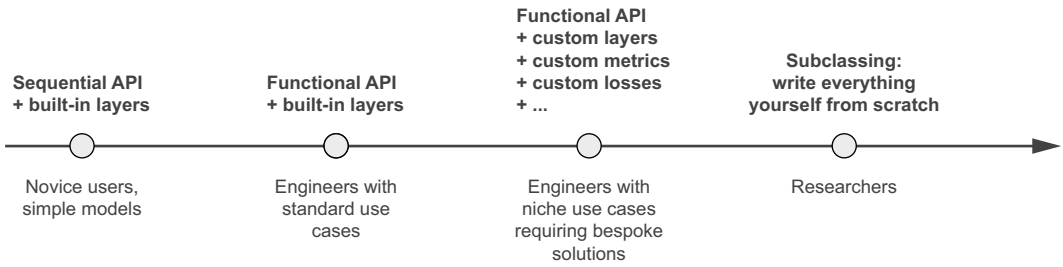

Figure 7.1 Progressive disclosure of complexity for model building

7.2.1 The Sequential model

The simplest way to build a Keras model is to use the Sequential model, which you already know about.

Listing 7.1 The `Sequential` class

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

Note that it's possible to build the same model incrementally via the `add()` method, which is similar to the `append()` method of a Python list.

Listing 7.2 Incrementally building a Sequential model

```
model = keras.Sequential()
model.add(layers.Dense(64, activation="relu"))
model.add(layers.Dense(10, activation="softmax"))
```

You saw in chapter 4 that layers only get built (which is to say, create their weights) when they are called for the first time. That's because the shape of the layers' weights depends on the shape of their input: until the input shape is known, they can't be created.

As such, the preceding Sequential model does not have any weights (listing 7.3) until you actually call it on some data, or call its `build()` method with an input shape (listing 7.4).

Listing 7.3 Models that aren't yet built have no weights

```
>>> model.weights
ValueError: Weights for model sequential_1 have not yet been created.
```

At that point, the model isn't built yet.

Listing 7.4 Calling a model for the first time to build it

```
>>> model.build(input_shape=(None, 3))
>>> model.weights
[<tf.Variable "dense_2/kernel:0" shape=(3, 64) dtype=float32, ... >,
 <tf.Variable "dense_2/bias:0" shape=(64,) dtype=float32, ... >
 <tf.Variable "dense_3/kernel:0" shape=(64, 10) dtype=float32, ... >,
 <tf.Variable "dense_3/bias:0" shape=(10,) dtype=float32, ... >]
```

Builds the model—now the model will expect samples of shape (3,). The None in the input shape signals that the batch size could be anything.

Now you can retrieve the model's weights.

After the model is built, you can display its contents via the `summary()` method, which comes in handy for debugging.

Listing 7.5 The `summary()` method

```
>>> model.summary()
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 64)                256
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650
=================================================================
Total params: 906
Trainable params: 906
Non-trainable params: 0
_________________________________________________________________
```

As you can see, this model happens to be named "sequential_1." You can give names to everything in Keras—every model, every layer.

Listing 7.6 Naming models and layers with the `name` argument

```
>>> model = keras.Sequential(name="my_example_model")
>>> model.add(layers.Dense(64, activation="relu", name="my_first_layer"))
>>> model.add(layers.Dense(10, activation="softmax", name="my_last_layer"))
>>> model.build((None, 3))
>>> model.summary()
Model: "my_example_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
my_first_layer (Dense)       (None, 64)                256
_________________________________________________________________
my_last_layer (Dense)        (None, 10)                650
=================================================================
Total params: 906
Trainable params: 906
Non-trainable params: 0
_________________________________________________________________
```

When building a Sequential model incrementally, it's useful to be able to print a summary of what the current model looks like after you add each layer. But you can't print a summary until the model is built! There's actually a way to have your `Sequential` built on the fly: just declare the shape of the model's inputs in advance. You can do this via the `Input` class.

Listing 7.7 Specifying the input shape of your model in advance

```
model = keras.Sequential()
model.add(keras.Input(shape=(3,)))
model.add(layers.Dense(64, activation="relu"))
```

Use Input to declare the shape of the inputs. Note that the shape argument must be the shape of each sample, not the shape of one batch.