

Chapter 15. Processing Sequences Using RNNs and CNNs

Predicting the future is something you do all the time, whether you are finishing a friend’s sentence or anticipating the smell of coffee at breakfast. In this chapter we will discuss recurrent neural networks (RNNs)—a class of nets that can predict the future (well, up to a point). RNNs can analyze time series data, such as the number of daily active users on your website, the hourly temperature in your city, your home’s daily power consumption, the trajectories of nearby cars, and more. Once an RNN learns past patterns in the data, it is able to use its knowledge to forecast the future, assuming of course that past patterns still hold in the future.

More generally, RNNs can work on sequences of arbitrary lengths, rather than on fixed-sized inputs. For example, they can take sentences, documents, or audio samples as input, making them extremely useful for natural language processing applications such as automatic translation or speech-to-text.

In this chapter, we will first go through the fundamental concepts underlying RNNs and how to train them using backpropagation through time. Then, we will use them to forecast a time series. Along the way, we will look at the popular ARMA family of models, often used to forecast time series, and use them as baselines to compare with our RNNs. After that, we’ll explore the two main difficulties that RNNs face:

- Unstable gradients (discussed in [Chapter 11](#)), which can be alleviated using various techniques, including *recurrent dropout* and *recurrent layer normalization*.
- A (very) limited short-term memory, which can be extended using LSTM and GRU cells.

RNNs are not the only types of neural networks capable of handling

sequential data. For small sequences, a regular dense network can do the trick, and for very long sequences, such as audio samples or text, convolutional neural networks can actually work quite well too. We will discuss both of these possibilities, and we will finish this chapter by implementing a WaveNet—a CNN architecture capable of handling sequences of tens of thousands of time steps. Let's get started!

Recurrent Neurons and Layers

Up to now we have focused on feedforward neural networks, where the activations flow only in one direction, from the input layer to the output layer. A recurrent neural network looks very much like a feedforward neural network, except it also has connections pointing backward.

Let's look at the simplest possible RNN, composed of one neuron receiving inputs, producing an output, and sending that output back to itself, as shown in [Figure 15-1](#) (left). At each *time step* t (also called a *frame*), this *recurrent neuron* receives the inputs $\mathbf{x}_{(t)}$ as well as its own output from the previous time step, $\hat{\mathbf{y}}_{(t-1)}$. Since there is no previous output at the first time step, it is generally set to 0. We can represent this tiny network against the time axis, as shown in [Figure 15-1](#) (right). This is called *unrolling the network through time* (it's the same recurrent neuron represented once per time step).

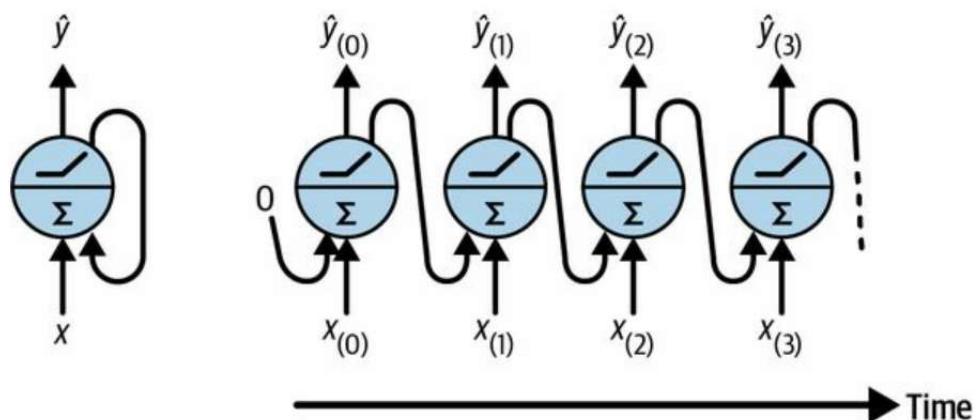


Figure 15-1. A recurrent neuron (left) unrolled through time (right)

You can easily create a layer of recurrent neurons. At each time step t , every neuron receives both the input vector $\mathbf{x}_{(t)}$ and the output vector from the previous time step $\hat{\mathbf{y}}_{(t-1)}$, as shown in [Figure 15-2](#). Note that both the inputs and outputs are now vectors (when there was just a single neuron, the output was a scalar).

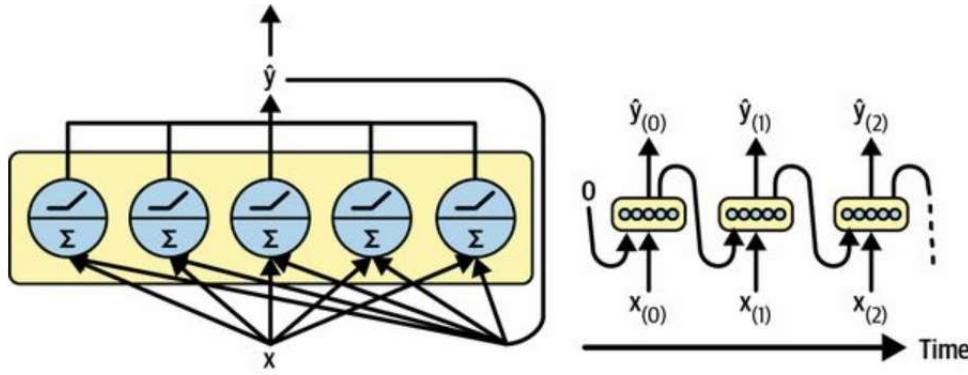


Figure 15-2. A layer of recurrent neurons (left) unrolled through time (right)

Each recurrent neuron has two sets of weights: one for the inputs $x_{(t)}$ and the other for the outputs of the previous time step, $\hat{y}_{(t-1)}$. Let's call these weight vectors w_x and $w_{\hat{y}}$. If we consider the whole recurrent layer instead of just one recurrent neuron, we can place all the weight vectors in two weight matrices: W_x and $W_{\hat{y}}$.

The output vector of the whole recurrent layer can then be computed pretty much as you might expect, as shown in [Equation 15-1](#), where b is the bias vector and $\phi(\cdot)$ is the activation function (e.g., ReLU⁻¹).

Equation 15-1. Output of a recurrent layer for a single instance

$$\hat{y}(t) = \phi(Wx^\top x(t) + W\hat{y}^\top \hat{y}(t-1) + b)$$

Just as with feedforward neural networks, we can compute a recurrent layer's output in one shot for an entire mini-batch by placing all the inputs at time step t into an input matrix $X_{(t)}$ (see [Equation 15-2](#)).

Equation 15-2. Outputs of a layer of recurrent neurons for all instances in a pass:[mini-batch]

$$\hat{Y}(t) = \phi(X(t)Wx + \hat{Y}(t-1)W\hat{y} + b) = \phi(X(t)\hat{Y}(t-1)W + b) \text{ with } W = Wx \\ W\hat{y}$$

In this equation:

- $\hat{Y}_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch (m is the number of instances in the

mini-batch and n_{neurons} is the number of neurons).

- $\mathbf{X}_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances (n_{inputs} is the number of input features).
- \mathbf{W}_x is an $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix containing the connection weights for the inputs of the current time step.
- $\mathbf{W}_{\hat{y}}$ is an $n_{\text{neurons}} \times n_{\text{neurons}}$ matrix containing the connection weights for the outputs of the previous time step.
- \mathbf{b} is a vector of size n_{neurons} containing each neuron's bias term.
- The weight matrices \mathbf{W}_x and $\mathbf{W}_{\hat{y}}$ are often concatenated vertically into a single weight matrix \mathbf{W} of shape $(n_{\text{inputs}} + n_{\text{neurons}}) \times n_{\text{neurons}}$ (see the second line of [Equation 15-2](#)).
- The notation $[\mathbf{X}_{(t)} \ \hat{\mathbf{Y}}_{(t-1)}]$ represents the horizontal concatenation of the matrices $\mathbf{X}_{(t)}$ and $\hat{\mathbf{Y}}_{(t-1)}$.

Notice that $\hat{\mathbf{Y}}_{(t)}$ is a function of $\mathbf{X}_{(t)}$ and $\hat{\mathbf{Y}}_{(t-1)}$, which is a function of $\mathbf{X}_{(t-1)}$ and $\hat{\mathbf{Y}}_{(t-2)}$, which is a function of $\mathbf{X}_{(t-2)}$ and $\hat{\mathbf{Y}}_{(t-3)}$, and so on. This makes $\hat{\mathbf{Y}}_{(t)}$ a function of all the inputs since time $t = 0$ (that is, $\mathbf{X}_{(0)}, \mathbf{X}_{(1)}, \dots, \mathbf{X}_{(t)}$). At the first time step, $t = 0$, there are no previous outputs, so they are typically assumed to be all zeros.

Memory Cells

Since the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, you could say it has a form of *memory*. A part of a neural network that preserves some state across time steps is called a *memory cell* (or simply a *cell*). A single recurrent neuron, or a layer of recurrent neurons, is a very basic cell, capable of learning only short patterns (typically about 10 steps long, but this varies depending on the task). Later in this chapter, we will look at some more complex and powerful types of cells capable of learning longer patterns (roughly 10 times longer, but again, this depends on the task).

A cell's state at time step t , denoted $\mathbf{h}_{(t)}$ (the "h" stands for "hidden"), is a function of some inputs at that time step and its state at the previous time step: $\mathbf{h}_{(t)} = f(\mathbf{x}_{(t)}, \mathbf{h}_{(t-1)})$. Its output at time step t , denoted $\hat{\mathbf{y}}_{(t)}$, is also a function of the previous state and the current inputs. In the case of the basic cells we have discussed so far, the output is just equal to the state, but in more complex cells this is not always the case, as shown in [Figure 15-3](#).

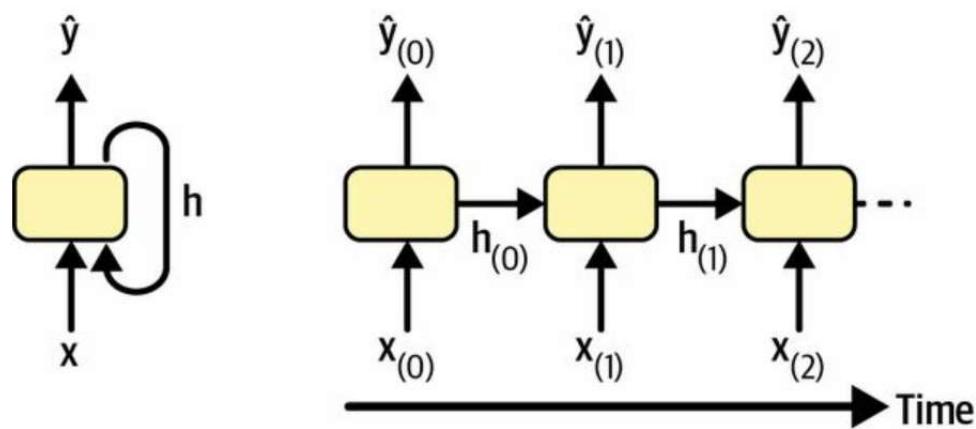


Figure 15-3. A cell's hidden state and its output may be different

Input and Output Sequences

An RNN can simultaneously take a sequence of inputs and produce a sequence of outputs (see the top-left network in [Figure 15-4](#)). This type of *sequence-to-sequence network* is useful to forecast time series, such as your home's daily power consumption: you feed it the data over the last N days, and you train it to output the power consumption shifted by one day into the future (i.e., from $N - 1$ days ago to tomorrow).

Alternatively, you could feed the network a sequence of inputs and ignore all outputs except for the last one (see the top-right network in [Figure 15-4](#)).

This is a *sequence-to-vector network*. For example, you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score (e.g., from 0 [hate] to 1 [love]).

Conversely, you could feed the network the same input vector over and over again at each time step and let it output a sequence (see the bottom-left network of [Figure 15-4](#)). This is a *vector-to-sequence network*. For example, the input could be an image (or the output of a CNN), and the output could be a caption for that image.

Lastly, you could have a sequence-to-vector network, called an *encoder*, followed by a vector-to-sequence network, called a *decoder* (see the bottom-right network of [Figure 15-4](#)). For example, this could be used for translating a sentence from one language to another. You would feed the network a sentence in one language, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in another language. This two-step model, called an *encoder-decoder*, ² works much better than trying to translate on the fly with a single sequence-to-sequence RNN (like the one represented at the top left): the last words of a sentence can affect the first words of the translation, so you need to wait until you have seen the whole sentence before translating it. We will go through the implementation of an encoder–decoder in [Chapter 16](#) (as you will see, it is a bit more complex than what [Figure 15-4](#) suggests).

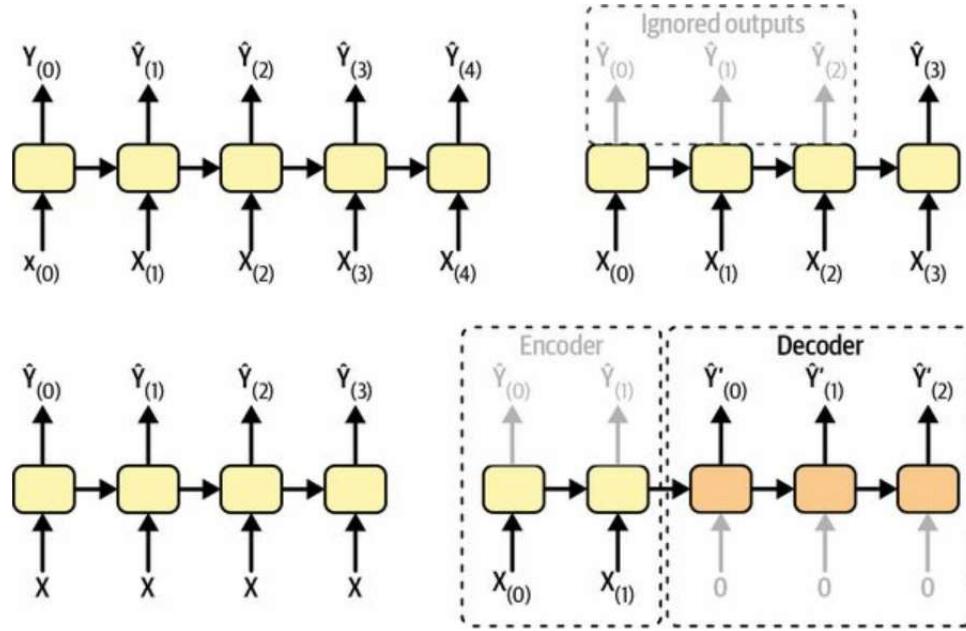


Figure 15-4. Sequence-to-sequence (top left), sequence-to-vector (top right), vector-to-sequence (bottom left), and encoder-decoder (bottom right) networks

This versatility sounds promising, but how do you train a recurrent neural network?

Training RNNs

To train an RNN, the trick is to unroll it through time (like we just did) and then use regular backpropagation (see [Figure 15-5](#)). This strategy is called *backpropagation through time* (BPTT).

Just like in regular backpropagation, there is a first forward pass through the unrolled network (represented by the dashed arrows). Then the output sequence is evaluated using a loss function $\mathcal{L}(\mathbf{Y}_{(0)}, \mathbf{Y}_{(1)}, \dots, \mathbf{Y}_{(T)}; \hat{\mathbf{Y}}_{(0)}, \hat{\mathbf{Y}}_{(1)}, \dots, \hat{\mathbf{Y}}_{(T)})$ (where $\mathbf{Y}_{(i)}$ is the i^{th} target, $\hat{\mathbf{Y}}_{(i)}$ is the i^{th} prediction, and T is the max time step). Note that this loss function may ignore some outputs. For example, in a sequence-to-vector RNN, all outputs are ignored except for the very last one. In [Figure 15-5](#), the loss function is computed based on the last three outputs only. The gradients of that loss function are then propagated backward through the unrolled network (represented by the solid arrows). In this example, since the outputs $\hat{\mathbf{Y}}_{(0)}$ and $\hat{\mathbf{Y}}_{(1)}$ are not used to compute the loss, the gradients do not flow backward through them; they only flow through $\hat{\mathbf{Y}}_{(2)}$, $\hat{\mathbf{Y}}_{(3)}$, and $\hat{\mathbf{Y}}_{(4)}$. Moreover, since the same parameters \mathbf{W} and \mathbf{b} are used at each time step, their gradients will be tweaked multiple times during backprop. Once the backward phase is complete and all the gradients have been computed, BPTT can perform a gradient descent step to update the parameters (this is no different from regular backprop).

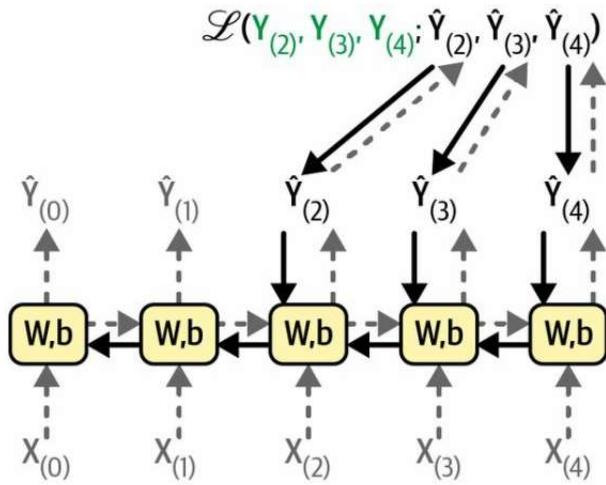


Figure 15-5. Backpropagation through time

Fortunately, Keras takes care of all of this complexity for you, as you will see. But before we get there, let's load a time series and start analyzing it using classical tools to better understand what we're dealing with, and to get some baseline metrics.

Forecasting a Time Series

All right! Let's pretend you've just been hired as a data scientist by Chicago's Transit Authority. Your first task is to build a model capable of forecasting the number of passengers that will ride on bus and rail the next day. You have access to daily ridership data since 2001. Let's walk through together how you would handle this. We'll start by loading and cleaning up the data:³

```
import pandas as pd
from pathlib import Path

path = Path("datasets/ridership/CTA_-_Ridership_-_Daily_Boarding_Totals.csv")
df = pd.read_csv(path, parse_dates=["service_date"])
df.columns = ["date", "day_type", "bus", "rail", "total"] # shorter names
df = df.sort_values("date").set_index("date")
df = df.drop("total", axis=1) # no need for total, it's just bus + rail
df = df.drop_duplicates() # remove duplicated months (2011-10 and 2014-07)
```

We load the CSV file, set short column names, sort the rows by date, remove the redundant total column, and drop duplicate rows. Now let's check what the first few rows look like:

```
>>> df.head()
      day_type  bus  rail
date
2001-01-01    U 297192 126455
2001-01-02    W  780827 501952
2001-01-03    W  824923 536432
2001-01-04    W  870021 550011
2001-01-05    W  890426 557917
```

On January 1st, 2001, 297,192 people boarded a bus in Chicago, and 126,455 boarded a train. The day_type column contains W for Weekdays, A for Saturdays, and U for Sundays or holidays.

Now let's plot the bus and rail ridership figures over a few months in 2019, to see what it looks like (see [Figure 15-6](#)):

```

import matplotlib.pyplot as plt

df["2019-03":"2019-05"].plot(grid=True, marker=".", figsize=(8, 3.5))
plt.show()

```

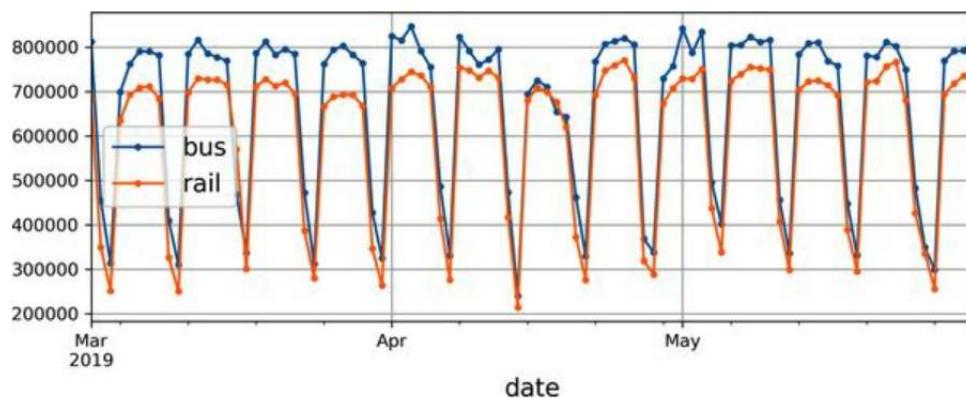


Figure 15-6. Daily ridership in Chicago

Note that Pandas includes both the start and end month in the range, so this plots the data from the 1st of March all the way up to the 31st of May. This is a *time series*: data with values at different time steps, usually at regular intervals. More specifically, since there are multiple values per time step, this is called a *multivariate time series*. If we only looked at the bus column, it would be a *univariate time series*, with a single value per time step.

Predicting future values (i.e., forecasting) is the most typical task when dealing with time series, and this is what we will focus on in this chapter.

Other tasks include imputation (filling in missing past values), classification, anomaly detection, and more.

Looking at Figure 15-6, we can see that a similar pattern is clearly repeated every week. This is called a weekly *seasonality*. In fact, it's so strong in this case that forecasting tomorrow's ridership by just copying the values from a week earlier will yield reasonably good results. This is called *naive forecasting*: simply copying a past value to make our forecast. Naive forecasting is often a great baseline, and it can even be tricky to beat in some cases.

NOTE

In general, naive forecasting means copying the latest known value (e.g., forecasting that tomorrow will be the same as today). However, in our case, copying the value from the previous week works better, due to the strong weekly seasonality.

To visualize these naive forecasts, let's overlay the two time series (for bus and rail) as well as the same time series lagged by one week (i.e., shifted toward the right) using dotted lines. We'll also plot the difference between the two (i.e., the value at time t minus the value at time $t - 7$); this is called *differencing* (see [Figure 15-7](#)):

```
diff_7 = df[["bus", "rail"]].diff(7)[“2019-03”：“2019-05”]

fig, axs = plt.subplots(2, 1, sharex=True, figsize=(8, 5))
df.plot(ax=axs[0], legend=False, marker=".") # original time series
df.shift(7).plot(ax=axs[0], grid=True, legend=False, linestyle=":") # lagged
diff_7.plot(ax=axs[1], grid=True, marker=".") # 7-day difference time series
plt.show()
```

Not too bad! Notice how closely the lagged time series track the actual time series. When a time series is correlated with a lagged version of itself, we say that the time series is *autocorrelated*. As you can see, most of the differences are fairly small, except at the end of May. Maybe there was a holiday at that time? Let's check the day_type column:

```
>>> list(df.loc["2019-05-25":“2019-05-27”][“day_type”])
['A', 'U', 'U']
```

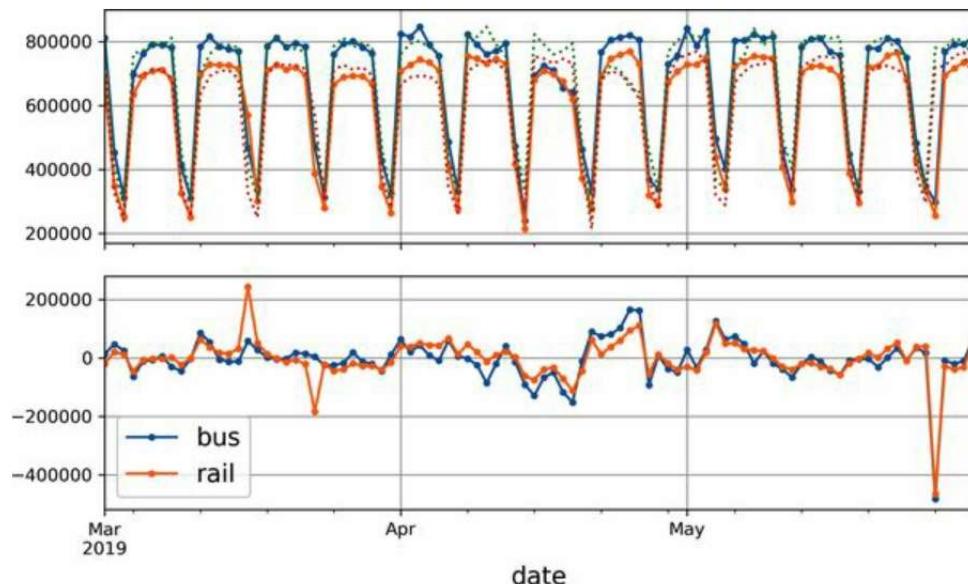


Figure 15-7. Time series overlaid with 7-day lagged time series (top), and difference between t and $t - 7$ (bottom)

Indeed, there was a long weekend back then: the Monday was the Memorial Day holiday. We could use this column to improve our forecasts, but for now let's just measure the mean absolute error over the three-month period we're arbitrarily focusing on—March, April, and May 2019—to get a rough idea:

```
>>> diff_7.abs().mean()
bus    43915.608696
rail   42143.271739
dtype: float64
```

Our naive forecasts get an MAE of about 43,916 bus riders, and about 42,143 rail riders. It's hard to tell at a glance how good or bad this is, so let's put the forecast errors into perspective by dividing them by the target values:

```
>>> targets = df[["bus", "rail"]]["2019-03":"2019-05"]
>>> (diff_7 / targets).abs().mean()
bus    0.082938
rail   0.089948
dtype: float64
```

What we just computed is called the *mean absolute percentage error* (MAPE): it looks like our naive forecasts give us a MAPE of roughly 8.3% for bus and 9.0% for rail. It's interesting to note that the MAE for the rail forecasts looks slightly better than the MAE for the bus forecasts, while the opposite is true for the MAPE. That's because the bus ridership is larger than the rail ridership, so naturally the forecast errors are also larger, but when we put the errors into perspective, it turns out that the bus forecasts are actually slightly better than the rail forecasts.

TIP

The MAE, MAPE, and MSE are among the most common metrics you can use to evaluate your forecasts. As always, choosing the right metric depends on the task. For example, if your project suffers quadratically more from large errors than from small ones, then the MSE may be preferable, as it strongly penalizes large errors.

Looking at the time series, there doesn't appear to be any significant monthly seasonality, but let's check whether there's any yearly seasonality. We'll look at the data from 2001 to 2019. To reduce the risk of data snooping, we'll ignore more recent data for now. Let's also plot a 12-month rolling average for each series to visualize long-term trends (see [Figure 15-8](#)):

```
period = slice("2001", "2019")
df_monthly = df.resample('M').mean() # compute the mean for each month
rolling_average_12_months = df_monthly[period].rolling(window=12).mean()

fig, ax = plt.subplots(figsize=(8, 4))
df_monthly[period].plot(ax=ax, marker=".")
rolling_average_12_months.plot(ax=ax, grid=True, legend=False)
plt.show()
```

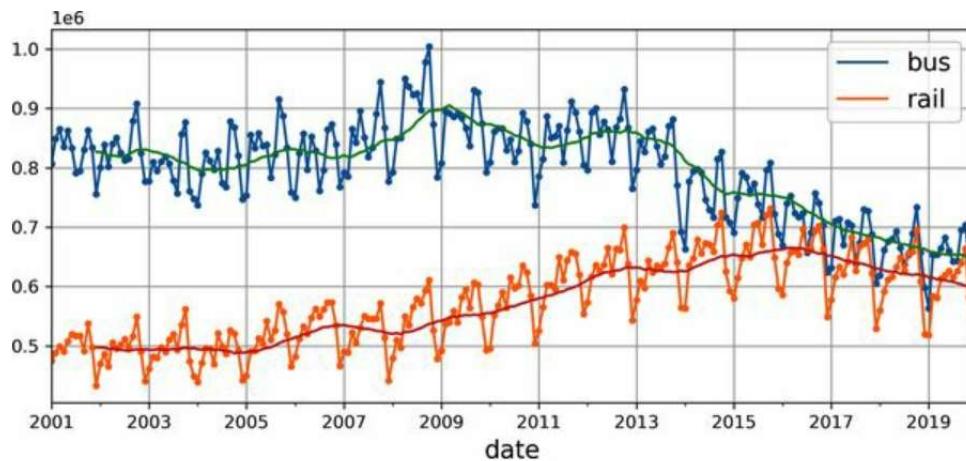


Figure 15-8. Yearly seasonality and long-term trends

Yep! There's definitely some yearly seasonality as well, although it is noisier than the weekly seasonality, and more visible for the rail series than the bus series: we see peaks and troughs at roughly the same dates each year. Let's check what we get if we plot the 12-month difference (see [Figure 15-9](#)):

```
df_monthly.diff(12)[period].plot(grid=True, marker=".", figsize=(8, 3))
plt.show()
```

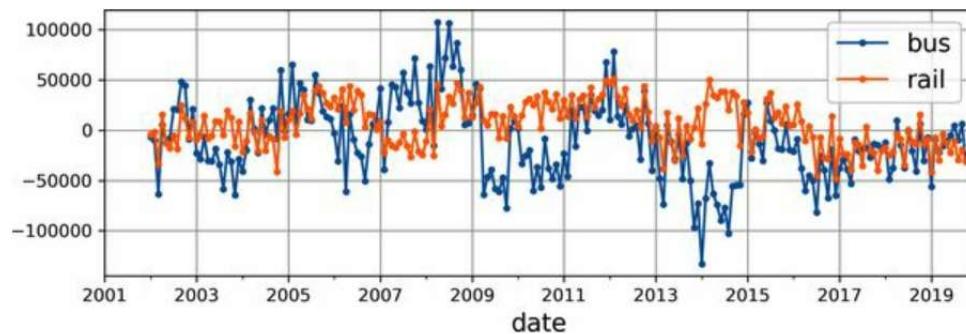


Figure 15-9. The 12-month difference

Notice how differencing not only removed the yearly seasonality, but it also removed the long-term trends. For example, the linear downward trend present in the time series from 2016 to 2019 became a roughly constant negative value in the differenced time series. In fact, differencing is a

common technique used to remove trend and seasonality from a time series: it's easier to study a *stationary* time series, meaning one whose statistical properties remain constant over time, without any seasonality or trends. Once you're able to make accurate forecasts on the differenced time series, it's easy to turn them into forecasts for the actual time series by just adding back the past values that were previously subtracted.

You may be thinking that we're only trying to predict tomorrow's ridership, so the long-term patterns matter much less than the short-term ones. You're right, but still, we may be able to improve performance slightly by taking long-term patterns into account. For example, daily bus ridership dropped by about 2,500 in October 2017, which represents about 570 fewer passengers each week, so if we were at the end of October 2017, it would make sense to forecast tomorrow's ridership by copying the value from last week, minus 570. Accounting for the trend will make your forecasts a bit more accurate on average.

Now that you're familiar with the ridership time series, as well as some of the most important concepts in time series analysis, including seasonality, trend, differencing, and moving averages, let's take a quick look at a very popular family of statistical models that are commonly used to analyze time series.

The ARMA Model Family

We'll start with the *autoregressive moving average* (ARMA) model, developed by Herman Wold in the 1930s: it computes its forecasts using a simple weighted sum of lagged values and corrects these forecasts by adding a moving average, very much like we just discussed. Specifically, the moving average component is computed using a weighted sum of the last few forecast errors. [Equation 15-3](#) shows how the model makes its forecasts.

Equation 15-3. Forecasting using an ARMA model

$$\hat{y}^{(t)} = \sum_{i=1}^p \alpha_i y^{(t-i)} + \sum_{i=1}^q \theta_i \epsilon^{(t-i)} \text{ with } \epsilon^{(t)} = y^{(t)} - \hat{y}^{(t)}$$

In this equation:

- $\hat{y}^{(t)}$ is the model's forecast for time step t .
- $y^{(t)}$ is the time series' value at time step t .
- The first sum is the weighted sum of the past p values of the time series, using the learned weights α_i . The number p is a hyperparameter, and it determines how far back into the past the model should look. This sum is the *autoregressive* component of the model: it performs regression based on past values.
- The second sum is the weighted sum over the past q forecast errors $\epsilon^{(t)}$, using the learned weights θ_i . The number q is a hyperparameter. This sum is the moving average component of the model.

Importantly, this model assumes that the time series is stationary. If it is not, then differencing may help. Using differencing over a single time step will produce an approximation of the derivative of the time series: indeed, it will give the slope of the series at each time step. This means that it will eliminate any linear trend, transforming it into a constant value. For example, if you apply one-step differencing to the series [3, 5, 7, 9, 11], you get the differenced series [2, 2, 2, 2].

If the original time series has a quadratic trend instead of a linear trend, then a

single round of differencing will not be enough. For example, the series [1, 4, 9, 16, 25, 36] becomes [3, 5, 7, 9, 11] after one round of differencing, but if you run differencing for a second round, then you get [2, 2, 2, 2]. So, running two rounds of differencing will eliminate quadratic trends. More generally, running d consecutive rounds of differencing computes an approximation of the d^{th} order derivative of the time series, so it will eliminate polynomial trends up to degree d . This hyperparameter d is called the *order of integration*.

Differencing is the central contribution of the *autoregressive integrated moving average* (ARIMA) model, introduced in 1970 by George Box and Gwilym Jenkins in their book *Time Series Analysis* (Wiley): this model runs d rounds of differencing to make the time series more stationary, then it applies a regular ARMA model. When making forecasts, it uses this ARMA model, then it adds back the terms that were subtracted by differencing.

One last member of the ARMA family is the *seasonal ARIMA* (SARIMA) model: it models the time series in the same way as ARIMA, but it additionally models a seasonal component for a given frequency (e.g., weekly), using the exact same ARIMA approach. It has a total of seven hyperparameters: the same p , d , and q hyperparameters as ARIMA, plus additional P , D , and Q hyperparameters to model the seasonal pattern, and lastly the period of the seasonal pattern, noted s . The hyperparameters P , D , and Q are just like p , d , and q , but they are used to model the time series at $t - s$, $t - 2s$, $t - 3s$, etc.

Let's see how to fit a SARIMA model to the rail time series, and use it to make a forecast for tomorrow's ridership. We'll pretend today is the last day of May 2019, and we want to forecast the rail ridership for "tomorrow", the 1st of June, 2019. For this, we can use the statsmodels library, which contains many different statistical models, including the ARMA model and its variants, implemented by the ARIMA class:

```
from statsmodels.tsa.arima.model import ARIMA  
  
origin, today = "2019-01-01", "2019-05-31"  
rail_series = df.loc[origin:today][["rail"]].asfreq("D")
```

```

model = ARIMA(rail_series,
              order=(1, 0, 0),
              seasonal_order=(0, 1, 1, 7))
model = model.fit()
y_pred = model.forecast() # returns 427,758.6

```

In this code example:

- We start by importing the ARIMA class, then we take the rail ridership data from the start of 2019 up to “today”, and we use `asfreq("D")` to set the time series’ frequency to daily: this doesn’t change the data at all in this case, since it’s already daily, but without this the ARIMA class would have to guess the frequency, and it would display a warning.
- Next, we create an ARIMA instance, passing it all the data until “today”, and we set the model hyperparameters: `order=(1, 0, 0)` means that $p = 1$, $d = 0$, $q = 0$, and `seasonal_order=(0, 1, 1, 7)` means that $P = 0$, $D = 1$, $Q = 1$, and $s = 7$. Notice that the `statsmodels` API differs a bit from Scikit-Learn’s API, since we pass the data to the model at construction time, instead of passing it to the `fit()` method.
- Next, we fit the model, and we use it to make a forecast for “tomorrow”, the 1st of June, 2019.

The forecast is 427,759 passengers, when in fact there were 379,044. Yikes, we’re 12.9% off—that’s pretty bad. It’s actually slightly worse than naive forecasting, which forecasts 426,932, off by 12.6%. But perhaps we were just unlucky that day? To check this, we can run the same code in a loop to make forecasts for every day in March, April, and May, and compute the MAE over that period:

```

origin, start_date, end_date = "2019-01-01", "2019-03-01", "2019-05-31"
time_period = pd.date_range(start_date, end_date)
rail_series = df.loc[origin:end_date][["rail"]].asfreq("D")
y_preds = []
for today in time_period.shift(-1):
    model = ARIMA(rail_series[origin:today], # train on data up to "today"
                  order=(1, 0, 0),
                  seasonal_order=(0, 1, 1, 7))
    model = model.fit() # note that we retrain the model every day!
    y_preds.append(model.forecast())

```

```
y_pred = model.forecast()[0]
y_preds.append(y_pred)

y_preds = pd.Series(y_preds, index=time_period)
mae = (y_preds - rail_series[time_period]).abs().mean() # returns 32,040.7
```

Ah, that's much better! The MAE is about 32,041, which is significantly lower than the MAE we got with naive forecasting (42,143). So although the model is not perfect, it still beats naive forecasting by a large margin, on average.

At this point, you may be wondering how to pick good hyperparameters for the SARIMA model. There are several methods, but the simplest to understand and to get started with is the brute-force approach: just run a grid search. For each model you want to evaluate (i.e., each hyperparameter combination), you can run the preceding code example, changing only the hyperparameter values. Good p , q , P , and Q values are usually fairly small (typically 0 to 2, sometimes up to 5 or 6), and d and D are typically 0 or 1, sometimes 2. As for s , it's just the main seasonal pattern's period: in our case it's 7 since there's a strong weekly seasonality. The model with the lowest MAE wins. Of course, you can replace the MAE with another metric if it better matches your business objective. And that's it!⁴

Preparing the Data for Machine Learning Models

Now that we have two baselines, naive forecasting and SARIMA, let's try to use the machine learning models we've covered so far to forecast this time series, starting with a basic linear model. Our goal will be to forecast tomorrow's ridership based on the ridership of the past 8 weeks of data (56 days). The inputs to our model will therefore be sequences (usually a single sequence per day once the model is in production), each containing 56 values from time steps $t - 55$ to t . For each input sequence, the model will output a single value: the forecast for time step $t + 1$.

But what will we use as training data? Well, that's the trick: we will use every 56-day window from the past as training data, and the target for each window will be the value immediately following it.

Keras actually has a nice utility function called `tf.keras.utils.timeseries_data_set_from_array()` to help us prepare the training set. It takes a time series as input, and it builds a `tf.data.Dataset` (introduced in [Chapter 13](#)) containing all the windows of the desired length, as well as their corresponding targets. Here's an example that takes a time series containing the numbers 0 to 5 and creates a dataset containing all the windows of length 3, with their corresponding targets, grouped into batches of size 2:

```
import tensorflow as tf

my_series = [0, 1, 2, 3, 4, 5]
my_dataset = tf.keras.utils.timeseries_dataset_from_array(
    my_series,
    targets=my_series[3:], # the targets are 3 steps into the future
    sequence_length=3,
    batch_size=2
)
```

Let's inspect the contents of this dataset:

```
>>> list(my_dataset)
[(<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[0, 1, 2],
```

```
[1, 2, 3]], dtype=int32),
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([3, 4], dtype=int32)>,
<tf.Tensor: shape=(1, 3), dtype=int32, numpy=array([[2, 3, 4]], dtype=int32)>,
<tf.Tensor: shape=(1,), dtype=int32, numpy=array([5], dtype=int32)>]
```

Each sample in the dataset is a window of length 3, along with its corresponding target (i.e., the value immediately after the window). The windows are [0, 1, 2], [1, 2, 3], and [2, 3, 4], and their respective targets are 3, 4, and 5. Since there are three windows in total, which is not a multiple of the batch size, the last batch only contains one window instead of two.

Another way to get the same result is to use the `window()` method of `tf.data`'s `Dataset` class. It's more complex, but it gives you full control, which will come in handy later in this chapter, so let's see how it works. The `window()` method returns a dataset of window datasets:

```
>>> for window_dataset in tf.data.Dataset.range(6).window(4, shift=1):
...     for element in window_dataset:
...         print(f"{element}", end=" ")
...     print()
...
0 1 2 3
1 2 3 4
2 3 4 5
3 4 5
4 5
5
```

In this example, the dataset contains six windows, each shifted by one step compared to the previous one, and the last three windows are smaller because they've reached the end of the series. In general you'll want to get rid of these smaller windows by passing `drop_remainder=True` to the `window()` method.

The `window()` method returns a *nested dataset*, analogous to a list of lists. This is useful when you want to transform each window by calling its dataset methods (e.g., to shuffle them or batch them). However, we cannot use a nested dataset directly for training, as our model will expect tensors as input, not datasets.

Therefore, we must call the `flat_map()` method: it converts a nested dataset

into a *flat dataset* (one that contains tensors, not datasets). For example, suppose $\{1, 2, 3\}$ represents a dataset containing the sequence of tensors 1, 2, and 3. If you flatten the nested dataset $\{\{1, 2\}, \{3, 4, 5, 6\}\}$, you get back the flat dataset $\{1, 2, 3, 4, 5, 6\}$.

Moreover, the `flat_map()` method takes a function as an argument, which allows you to transform each dataset in the nested dataset before flattening. For example, if you pass the function `lambda ds: ds.batch(2)` to `flat_map()`, then it will transform the nested dataset $\{\{1, 2\}, \{3, 4, 5, 6\}\}$ into the flat dataset $\{[1, 2], [3, 4], [5, 6]\}$: it's a dataset containing 3 tensors, each of size 2.

With that in mind, we are ready to flatten our dataset:

```
>>> dataset = tf.data.Dataset.range(6).window(4, shift=1, drop_remainder=True)
>>> dataset = dataset.flat_map(lambda window_dataset: window_dataset.batch(4))
>>> for window_tensor in dataset:
...     print(f"{window_tensor}")
...
[0 1 2 3]
[1 2 3 4]
[2 3 4 5]
```

Since each window dataset contains exactly four items, calling `batch(4)` on a window produces a single tensor of size 4. Great! We now have a dataset containing consecutive windows represented as tensors. Let's create a little helper function to make it easier to extract windows from a dataset:

```
def to_windows(dataset, length):
    dataset = dataset.window(length, shift=1, drop_remainder=True)
    return dataset.flat_map(lambda window_ds: window_ds.batch(length))
```

The last step is to split each window into inputs and targets, using the `map()` method. We can also group the resulting windows into batches of size 2:

```
>>> dataset = to_windows(tf.data.Dataset.range(6), 4) # 3 inputs + 1 target = 4
>>> dataset = dataset.map(lambda window: (window[:-1], window[-1]))
>>> list(dataset.batch(2))
[(<tf.Tensor: shape=(2, 3), dtype=int64, numpy=
array([[0, 1, 2],
```

```
[1, 2, 3]]>,
<tf.Tensor: shape=(2,), dtype=int64, numpy=array([3, 4])>,
<tf.Tensor: shape=(1, 3), dtype=int64, numpy=array([[2, 3, 4]])>,
<tf.Tensor: shape=(1,), dtype=int64, numpy=array([5])>]
```

As you can see, we now have the same output as we got earlier with the `timeseries_dataset_from_array()` function (with a bit more effort, but it will be worthwhile soon).

Now, before we start training, we need to split our data into a training period, a validation period, and a test period. We will focus on the rail ridership for now. We will also scale it down by a factor of one million, to ensure the values are near the 0–1 range; this plays nicely with the default weight initialization and learning rate:

```
rail_train = df["rail"]["2016-01":"2018-12"] / 1e6
rail_valid = df["rail"]["2019-01":"2019-05"] / 1e6
rail_test = df["rail"]["2019-06":] / 1e6
```

NOTE

When dealing with time series, you generally want to split across time. However, in some cases you may be able to split along other dimensions, which will give you a longer time period to train on. For example, if you have data about the financial health of 10,000 companies from 2001 to 2019, you might be able to split this data across the different companies. It's very likely that many of these companies will be strongly correlated, though (e.g., whole economic sectors may go up or down jointly), and if you have correlated companies across the training set and the test set, your test set will not be as useful, as its measure of the generalization error will be optimistically biased.

Next, let's use `timeseries_dataset_from_array()` to create datasets for training and validation. Since gradient descent expects the instances in the training set to be independent and identically distributed (IID), as we saw in [Chapter 4](#), we must set the argument `shuffle=True` to shuffle the training windows (but not their contents):

```
seq_length = 56
train_ds = tf.keras.utils.timeseries_dataset_from_array(
    rail_train.to_numpy(),
```

```
targets=rail_train[seq_length:],  
sequence_length=seq_length,  
batch_size=32,  
shuffle=True,  
seed=42  
)  
valid_ds = tf.keras.utils.timeseries_dataset_from_array(  
    rail_valid.to_numpy(),  
    targets=rail_valid[seq_length:],  
    sequence_length=seq_length,  
    batch_size=32  
)
```

And now we're ready to build and train any regression model we want!

Forecasting Using a Linear Model

Let's try a basic linear model first. We will use the Huber loss, which usually works better than minimizing the MAE directly, as discussed in [Chapter 10](#). We'll also use early stopping:

```
tf.random.set_seed(42)
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_shape=[seq_length])
])
early_stopping_cb = tf.keras.callbacks.EarlyStopping(
    monitor="val_mae", patience=50, restore_best_weights=True)
opt = tf.keras.optimizers.SGD(learning_rate=0.02, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(), optimizer=opt, metrics=["mae"])
history = model.fit(train_ds, validation_data=valid_ds, epochs=500,
                     callbacks=[early_stopping_cb])
```

This model reaches a validation MAE of about 37,866 (your mileage may vary). That's better than naive forecasting, but worse than the SARIMA model.⁵

Can we do better with an RNN? Let's see!

Forecasting Using a Simple RNN

Let's try the most basic RNN, containing a single recurrent layer with just one recurrent neuron, as we saw in [Figure 15-1](#):

```
model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

All recurrent layers in Keras expect 3D inputs of shape [*batch size, time steps, dimensionality*], where *dimensionality* is 1 for univariate time series and more for multivariate time series. Recall that the `input_shape` argument ignores the first dimension (i.e., the batch size), and since recurrent layers can accept input sequences of any length, we can set the second dimension to `None`, which means “any size”. Lastly, since we’re dealing with a univariate time series, we need the last dimension’s size to be 1. This is why we specified the input shape `[None, 1]`: it means “univariate sequences of any length”. Note that the datasets actually contain inputs of shape [*batch size, time steps*], so we’re missing the last dimension, of size 1, but Keras is kind enough to add it for us in this case.

This model works exactly as we saw earlier: the initial state $h_{(\text{init})}$ is set to 0, and it is passed to a single recurrent neuron, along with the value of the first time step, $x_{(0)}$. The neuron computes a weighted sum of these values plus the bias term, and it applies the activation function to the result, using the hyperbolic tangent function by default. The result is the first output, y_0 . In a simple RNN, this output is also the new state h_0 . This new state is passed to the same recurrent neuron along with the next input value, $x_{(1)}$, and the process is repeated until the last time step. At the end, the layer just outputs the last value: in our case the sequences are 56 steps long, so the last value is y_{55} . All of this is performed simultaneously for every sequence in the batch, of which there are 32 in this case.

NOTE

By default, recurrent layers in Keras only return the final output. To make them return one output per time step, you must set `return_sequences=True`, as you will see.

So that's our first recurrent model! It's a sequence-to-vector model. Since there's a single output neuron, the output vector has a size of 1.

Now if you compile, train, and evaluate this model just like the previous model, you will find that it's no good at all: its validation MAE is greater than 100,000! Ouch. That was to be expected, for two reasons:

1. The model only has a single recurrent neuron, so the only data it can use to make a prediction at each time step is the input value at the current time step and the output value from the previous time step. That's not much to go on! In other words, the RNN's memory is extremely limited: it's just a single number, its previous output. And let's count how many parameters this model has: since there's just one recurrent neuron with only two input values, the whole model only has three parameters (two weights plus a bias term). That's far from enough for this time series. In contrast, our previous model could look at all 56 previous values at once, and it had a total of 57 parameters.
2. The time series contains values from 0 to about 1.4, but since the default activation function is `tanh`, the recurrent layer can only output values between -1 and +1. There's no way it can predict values between 1.0 and 1.4.

Let's fix both of these issues: we will create a model with a larger recurrent layer, containing 32 recurrent neurons, and we will add a dense output layer on top of it with a single output neuron and no activation function. The recurrent layer will be able to carry much more information from one time step to the next, and the dense output layer will project the final output from 32 dimensions down to 1, without any value range constraints:

```
univar_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 1]),
    tf.keras.layers.Dense(1) # no activation function by default
])
```

Now if you compile, fit, and evaluate this model just like the previous one, you will find that its validation MAE reaches 27,703. That's the best model we've trained so far, and it even beats the SARIMA model: we're doing pretty well!

TIP

We've only normalized the time series, without removing trend and seasonality, and yet the model still performs well. This is convenient, as it makes it possible to quickly search for promising models without worrying too much about preprocessing. However, to get the best performance, you may want to try making the time series more stationary; for example, using differencing.

Forecasting Using a Deep RNN

It is quite common to stack multiple layers of cells, as shown in [Figure 15-10](#). This gives you a *deep RNN*.

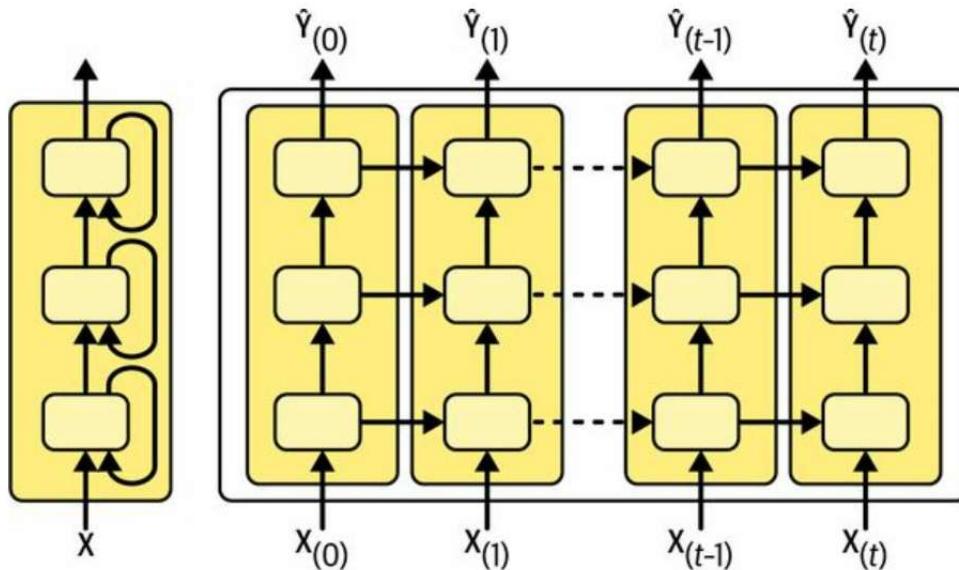


Figure 15-10. A deep RNN (left) unrolled through time (right)

Implementing a deep RNN with Keras is straightforward: just stack recurrent layers. In the following example, we use three SimpleRNN layers (but we could use any other type of recurrent layer instead, such as an LSTM layer or a GRU layer, which we will discuss shortly). The first two are sequence-to-sequence layers, and the last one is a sequence-to-vector layer. Finally, the Dense layer produces the model’s forecast (you can think of it as a vector-to-vector layer). So this model is just like the model represented in [Figure 15-10](#), except the outputs $\hat{Y}_{(0)}$ to $\hat{Y}_{(t-1)}$ are ignored, and there’s a dense layer on top of $\hat{Y}_{(t)}$, which outputs the actual forecast:

```
deep_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 1]),
    tf.keras.layers.SimpleRNN(32, return_sequences=True),
    tf.keras.layers.SimpleRNN(32),
    tf.keras.layers.Dense(1)
```

)

WARNING

Make sure to set `return_sequences=True` for all recurrent layers (except the last one, if you only care about the last output). If you forget to set this parameter for one recurrent layer, it will output a 2D array containing only the output of the last time step, instead of a 3D array containing outputs for all time steps. The next recurrent layer will complain that you are not feeding it sequences in the expected 3D format.

If you train and evaluate this model, you will find that it reaches an MAE of about 31,211. That's better than both baselines, but it doesn't beat our "shallow" RNN. It looks like this RNN is a bit too large for our task.

Forecasting Multivariate Time Series

A great quality of neural networks is their flexibility: in particular, they can deal with multivariate time series with almost no change to their architecture. For example, let's try to forecast the rail time series using both the bus and rail data as input. In fact, let's also throw in the day type! Since we can always know in advance whether tomorrow is going to be a weekday, a weekend, or a holiday, we can shift the day type series one day into the future, so that the model is given tomorrow's day type as input. For simplicity, we'll do this processing using Pandas:

```
df_mulvar = df[["bus", "rail"]] / 1e6 # use both bus & rail series as input
df_mulvar["next_day_type"] = df["day_type"].shift(-1) # we know tomorrow's type
df_mulvar = pd.get_dummies(df_mulvar) # one-hot encode the day type
```

Now `df_mulvar` is a DataFrame with five columns: the bus and rail data, plus three columns containing the one-hot encoding of the next day's type (recall that there are three possible day types, W, A, and U). Next we can proceed much like we did earlier. First we split the data into three periods, for training, validation, and testing:

```
mulvar_train = df_mulvar["2016-01":"2018-12"]
mulvar_valid = df_mulvar["2019-01":"2019-05"]
mulvar_test = df_mulvar["2019-06":]
```

Then we create the datasets:

```
train_mulvar_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_train.to_numpy(), # use all 5 columns as input
    targets=mulvar_train["rail"][seq_length:], # forecast only the rail series
    [...] # the other 4 arguments are the same as earlier
)
valid_mulvar_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_valid.to_numpy(),
    targets=mulvar_valid["rail"][seq_length:],
    [...] # the other 2 arguments are the same as earlier
)
```

And finally we create the RNN:

```
mulvar_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
    tf.keras.layers.Dense(1)
])
```

Notice that the only difference from the univar_model RNN we built earlier is the input shape: at each time step, the model now receives five inputs instead of one. This model actually reaches a validation MAE of 22,062. Now we're making big progress!

In fact, it's not too hard to make the RNN forecast both the bus and rail ridership. You just need to change the targets when creating the datasets, setting them to `mulvar_train[["bus", "rail"]][seq_length:]` for the training set, and `mulvar_valid[["bus", "rail"]][seq_length:]` for the validation set. You must also add an extra neuron in the output Dense layer, since it must now make two forecasts: one for tomorrow's bus ridership, and the other for rail. That's all there is to it!

As we discussed in [Chapter 10](#), using a single model for multiple related tasks often results in better performance than using a separate model for each task, since features learned for one task may be useful for the other tasks, and also because having to perform well across multiple tasks prevents the model from overfitting (it's a form of regularization). However, it depends on the task, and in this particular case the multitask RNN that forecasts both the bus and the rail ridership doesn't perform quite as well as dedicated models that forecast one or the other (using all five columns as input). Still, it reaches a validation MAE of 25,330 for rail and 26,369 for bus, which is pretty good.

Forecasting Several Time Steps Ahead

So far we have only predicted the value at the next time step, but we could just as easily have predicted the value several steps ahead by changing the targets appropriately (e.g., to predict the ridership 2 weeks from now, we could just change the targets to be the value 14 days ahead instead of 1 day ahead). But what if we want to predict the next 14 values?

The first option is to take the univar_model RNN we trained earlier for the rail time series, make it predict the next value, and add that value to the inputs, acting as if the predicted value had actually occurred; we would then use the model again to predict the following value, and so on, as in the following code:

```
import numpy as np

X = rail_valid.to_numpy()[:,np.newaxis, :seq_length, np.newaxis]
for step_ahead in range(14):
    y_pred_one = univar_model.predict(X)
    X = np.concatenate([X, y_pred_one.reshape(1, 1, 1)], axis=1)
```

In this code, we take the rail ridership of the first 56 days of the validation period, and we convert the data to a NumPy array of shape [1, 56, 1] (recall that recurrent layers expect 3D inputs). Then we repeatedly use the model to forecast the next value, and we append each forecast to the input series, along the time axis (axis=1). The resulting forecasts are plotted in [Figure 15-11](#).

WARNING

If the model makes an error at one time step, then the forecasts for the following time steps are impacted as well: the errors tend to accumulate. So, it's preferable to use this technique only for a small number of steps.

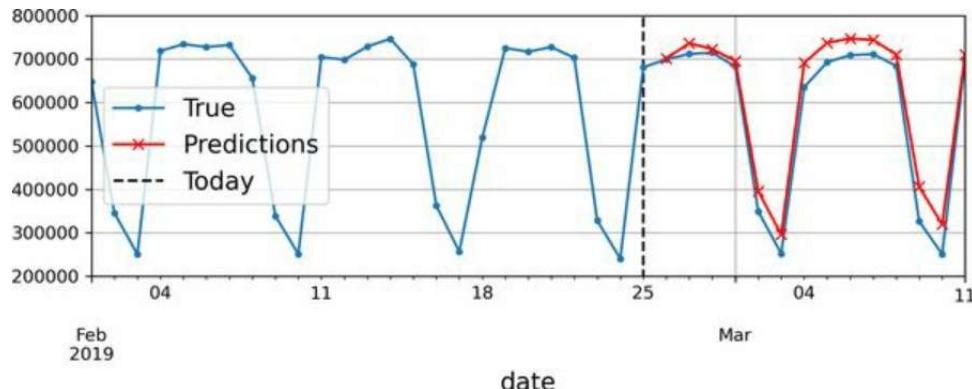


Figure 15-11. Forecasting 14 steps ahead, 1 step at a time

The second option is to train an RNN to predict the next 14 values in one shot. We can still use a sequence-to-vector model, but it will output 14 values instead of 1. However, we first need to change the targets to be vectors containing the next 14 values. To do this, we can use `timeseries_dataset_from_array()` again, but this time asking it to create datasets without targets (`targets=None`) and with longer sequences, of length `seq_length + 14`. Then we can use the datasets' `map()` method to apply a custom function to each batch of sequences, splitting them into inputs and targets. In this example, we use the multivariate time series as input (using all five columns), and we forecast the rail ridership for the next 14 days:⁶

```
def split_inputs_and_targets(mulvar_series, ahead=14, target_col=1):
    return mulvar_series[:, :-ahead], mulvar_series[:, -ahead:, target_col]

ahead_train_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_train.to_numpy(),
    targets=None,
    sequence_length=seq_length + 14,
    [...] # the other 3 arguments are the same as earlier
).map(split_inputs_and_targets)
ahead_valid_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_valid.to_numpy(),
    targets=None,
    sequence_length=seq_length + 14,
    batch_size=32
).map(split_inputs_and_targets)
```

Now we just need the output layer to have 14 units instead of 1:

```
ahead_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
```

After training this model, you can predict the next 14 values at once like this:

```
X = mulvar_valid.to_numpy()[:, np.newaxis, :seq_length] # shape [1, 56, 5]
Y_pred = ahead_model.predict(X) # shape [1, 14]
```

This approach works quite well. Its forecasts for the next day are obviously better than its forecasts for 14 days into the future, but it doesn't accumulate errors like the previous approach did. However, we can still do better, using a sequence-to-sequence (or *seq2seq*) model.

Forecasting Using a Sequence-to-Sequence Model

Instead of training the model to forecast the next 14 values only at the very last time step, we can train it to forecast the next 14 values at each and every time step. In other words, we can turn this sequence-to-vector RNN into a sequence-to-sequence RNN. The advantage of this technique is that the loss will contain a term for the output of the RNN at each and every time step, not just for the output at the last time step.

This means there will be many more error gradients flowing through the model, and they won't have to flow through time as much since they will come from the output of each time step, not just the last one. This will both stabilize and speed up training.

To be clear, at time step 0 the model will output a vector containing the forecasts for time steps 1 to 14, then at time step 1 the model will forecast time steps 2 to 15, and so on. In other words, the targets are sequences of consecutive windows, shifted by one time step at each time step. The target is not a vector anymore, but a sequence of the same length as the inputs, containing a 14-dimensional vector at each step.

Preparing the datasets is not trivial, since each instance has a window as input and a sequence of windows as output. One way to do this is to use the `to_windows()` utility function we created earlier, twice in a row, to get windows of consecutive windows. For example, let's turn the series of numbers 0 to 6 into a dataset containing sequences of 4 consecutive windows, each of length 3:

```
>>> my_series = tf.data.Dataset.range(7)
>>> dataset = to_windows(to_windows(my_series, 3), 4)
>>> list(dataset)
[<tf.Tensor: shape=(4, 3), dtype=int64, numpy=
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])>,
 <tf.Tensor: shape=(4, 3), dtype=int64, numpy=
array([[1, 2, 3],
```

```
[2, 3, 4],  
[3, 4, 5],  
[4, 5, 6]])>]
```

Now we can use the `map()` method to split these windows of windows into inputs and targets:

```
>>> dataset = dataset.map(lambda S: (S[:, 0], S[:, 1:]))

>>> list(dataset)
[(<tf.Tensor: shape=(4,), dtype=int64, numpy=array([0, 1, 2, 3])>,
 <tf.Tensor: shape=(4, 2), dtype=int64, numpy=
 array([[1, 2],
        [2, 3],
        [3, 4],
        [4, 5]]>),
 (<tf.Tensor: shape=(4,), dtype=int64, numpy=array([1, 2, 3, 4])>,
 <tf.Tensor: shape=(4, 2), dtype=int64, numpy=
 array([[2, 3],
        [3, 4],
        [4, 5],
        [5, 6]]>))]
```

Now the dataset contains sequences of length 4 as inputs, and the targets are sequences containing the next two steps, for each time step. For example, the first input sequence is `[0, 1, 2, 3]`, and its corresponding targets are `[[1, 2], [2, 3], [3, 4], [4, 5]]`, which are the next two values for each time step. If you're like me, you will probably need a few minutes to wrap your head around this. Take your time!

NOTE

It may be surprising that the targets contain values that appear in the inputs. Isn't that cheating? Fortunately, not at all: at each time step, an RNN only knows about past time steps; it cannot look ahead. It is said to be a *causal* model.

Let's create another little utility function to prepare the datasets for our sequence-to-sequence model. It will also take care of shuffling (optional) and batching:

```

def to_seq2seq_dataset(series, seq_length=56, ahead=14, target_col=1,
                      batch_size=32, shuffle=False, seed=None):
    ds = to_windows(tf.data.Dataset.from_tensor_slices(series), ahead + 1)
    ds = to_windows(ds, seq_length).map(lambda S: (S[:, 0], S[:, 1:, 1]))
    if shuffle:
        ds = ds.shuffle(8 * batch_size, seed=seed)
    return ds.batch(batch_size)

```

Now we can use this function to create the datasets:

```

seq2seq_train = to_seq2seq_dataset(mulvar_train, shuffle=True, seed=42)
seq2seq_valid = to_seq2seq_dataset(mulvar_valid)

```

And lastly, we can build the sequence-to-sequence model:

```

seq2seq_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])

```

It is almost identical to our previous model: the only difference is that we set `return_sequences=True` in the `SimpleRNN` layer. This way, it will output a sequence of vectors (each of size 32), instead of outputting a single vector at the last time step. The `Dense` layer is smart enough to handle sequences as input: it will be applied at each time step, taking a 32-dimensional vector as input and outputting a 14-dimensional vector. In fact, another way to get the exact same result is to use a `Conv1D` layer with a kernel size of 1: `Conv1D(14, kernel_size=1)`.

TIP

Keras offers a `TimeDistributed` layer that lets you apply any vector-to-vector layer to every vector in the input sequences, at every time step. It does this efficiently, by reshaping the inputs so that each time step is treated as a separate instance, then it reshapes the layer's outputs to recover the time dimension. In our case, we don't need it since the `Dense` layer already supports sequences as inputs.

The training code is the same as usual. During training, all the model's

outputs are used, but after training only the output of the very last time step matters, and the rest can be ignored. For example, we can forecast the rail ridership for the next 14 days like this:

```
X = mulvar_valid.to_numpy()[:, seq_length]
y_pred_14 = seq2seq_model.predict(X)[:, -1] # only the last time step's output
```

If you evaluate this model's forecasts for $t + 1$, you will find a validation MAE of 25,519. For $t + 2$ it's 26,274, and the performance continues to drop gradually as the model tries to forecast further into the future. At $t + 14$, the MAE is 34,322.

TIP

You can combine both approaches to forecasting multiple steps ahead: for example, you can train a model that forecasts 14 days ahead, then take its output and append it to the inputs, then run the model again to get forecasts for the following 14 days, and possibly repeat the process.

Simple RNNs can be quite good at forecasting time series or handling other kinds of sequences, but they do not perform as well on long time series or sequences. Let's discuss why and see what we can do about it.

Handling Long Sequences

To train an RNN on long sequences, we must run it over many time steps, making the unrolled RNN a very deep network. Just like any deep neural network it may suffer from the unstable gradients problem, discussed in [Chapter 11](#): it may take forever to train, or training may be unstable. Moreover, when an RNN processes a long sequence, it will gradually forget the first inputs in the sequence. Let's look at both these problems, starting with the unstable gradients problem.

Fighting the Unstable Gradients Problem

Many of the tricks we used in deep nets to alleviate the unstable gradients problem can also be used for RNNs: good parameter initialization, faster optimizers, dropout, and so on. However, nonsaturating activation functions (e.g., ReLU) may not help as much here. In fact, they may actually lead the RNN to be even more unstable during training. Why? Well, suppose gradient descent updates the weights in a way that increases the outputs slightly at the first time step. Because the same weights are used at every time step, the outputs at the second time step may also be slightly increased, and those at the third, and so on until the outputs explode—and a nonsaturating activation function does not prevent that.

You can reduce this risk by using a smaller learning rate, or you can use a saturating activation function like the hyperbolic tangent (this explains why it's the default).

In much the same way, the gradients themselves can explode. If you notice that training is unstable, you may want to monitor the size of the gradients (e.g., using TensorBoard) and perhaps use gradient clipping.

Moreover, batch normalization cannot be used as efficiently with RNNs as with deep feedforward nets. In fact, you cannot use it between time steps, only between recurrent layers.

To be more precise, it is technically possible to add a BN layer to a memory cell (as you will see shortly) so that it will be applied at each time step (both on the inputs for that time step and on the hidden state from the previous step). However, the same BN layer will be used at each time step, with the same parameters, regardless of the actual scale and offset of the inputs and hidden state. In practice, this does not yield good results, as was demonstrated by César Laurent et al. in a [2015 paper](#):⁷ the authors found that BN was slightly beneficial only when it was applied to the layer's inputs, not to the hidden states. In other words, it was slightly better than nothing when applied between recurrent layers (i.e., vertically in [Figure 15-10](#)), but not within recurrent layers (i.e., horizontally). In Keras, you can apply BN

between layers simply by adding a BatchNormalization layer before each recurrent layer, but it will slow down training, and it may not help much.

Another form of normalization often works better with RNNs: *layer normalization*. This idea was introduced by Jimmy Lei Ba et al. in a [2016 paper](#):⁸ it is very similar to batch normalization, but instead of normalizing across the batch dimension, layer normalization normalizes across the features dimension. One advantage is that it can compute the required statistics on the fly, at each time step, independently for each instance. This also means that it behaves the same way during training and testing (as opposed to BN), and it does not need to use exponential moving averages to estimate the feature statistics across all instances in the training set, like BN does. Like BN, layer normalization learns a scale and an offset parameter for each input. In an RNN, it is typically used right after the linear combination of the inputs and the hidden states.

Let's use Keras to implement layer normalization within a simple memory cell. To do this, we need to define a custom memory cell, which is just like a regular layer, except its `call()` method takes two arguments: the inputs at the current time step and the hidden states from the previous time step.

Note that the `states` argument is a list containing one or more tensors. In the case of a simple RNN cell it contains a single tensor equal to the outputs of the previous time step, but other cells may have multiple state tensors (e.g., an `LSTMCell` has a long-term state and a short-term state, as you will see shortly). A cell must also have a `state_size` attribute and an `output_size` attribute. In a simple RNN, both are simply equal to the number of units. The following code implements a custom memory cell that will behave like a `SimpleRNNCell`, except it will also apply layer normalization at each time step:

```

self.layer_norm = tf.keras.layers.LayerNormalization()
self.activation = tf.keras.activations.get(activation)

def call(self, inputs, states):
    outputs, new_states = self.simple_rnn_cell(inputs, states)
    norm_outputs = self.activation(self.layer_norm(outputs))
    return norm_outputs, [norm_outputs]

```

Let's walk through this code:

- Our LNSimpleRNNCell class inherits from the `tf.keras.layers.Layer` class, just like any custom layer.
- The constructor takes the number of units and the desired activation function and sets the `state_size` and `output_size` attributes, then creates a `SimpleRNNCell` with no activation function (because we want to perform layer normalization after the linear operation but before the activation function). ⁹ Then the constructor creates the `LayerNormalization` layer, and finally it fetches the desired activation function.
- The `call()` method starts by applying the `simpleRNNCell`, which computes a linear combination of the current inputs and the previous hidden states, and it returns the result twice (indeed, in a `SimpleRNNCell`, the outputs are just equal to the hidden states: in other words, `new_states[0]` is equal to `outputs`, so we can safely ignore `new_states` in the rest of the `call()` method). Next, the `call()` method applies layer normalization, followed by the activation function. Finally, it returns the outputs twice: once as the outputs, and once as the new hidden states. To use this custom cell, all we need to do is create a `tf.keras.layers.RNN` layer, passing it a cell instance:

```

custom_ln_model = tf.keras.Sequential([
    tf.keras.layers.RNN(LNSimpleRNNCell(32), return_sequences=True,
                        input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])

```

Similarly, you could create a custom cell to apply dropout between each time

step. But there's a simpler way: most recurrent layers and cells provided by Keras have dropout and recurrent_dropout hyperparameters: the former defines the dropout rate to apply to the inputs, and the latter defines the dropout rate for the hidden states, between time steps. So, there's no need to create a custom cell to apply dropout at each time step in an RNN.

With these techniques, you can alleviate the unstable gradients problem and train an RNN much more efficiently. Now let's look at how to deal with the short-term memory problem.

TIP

When forecasting time series, it is often useful to have some error bars along with your predictions. For this, one approach is to use MC dropout, introduced in [Chapter 11](#): use recurrent_dropout during training, then keep dropout active at inference time by calling the model using `model(X, training=True)`. Repeat this several times to get multiple slightly different forecasts, then compute the mean and standard deviation of these predictions for each time step.

Tackling the Short-Term Memory Problem

Due to the transformations that the data goes through when traversing an RNN, some information is lost at each time step. After a while, the RNN's state contains virtually no trace of the first inputs. This can be a showstopper. Imagine Dory the fish ¹⁰ trying to translate a long sentence; by the time she's finished reading it, she has no clue how it started. To tackle this problem, various types of cells with long-term memory have been introduced. They have proven so successful that the basic cells are not used much anymore. Let's first look at the most popular of these long-term memory cells: the LSTM cell.

LSTM cells

The *long short-term memory* (LSTM) cell was proposed in 1997¹¹ by Sepp Hochreiter and Jürgen Schmidhuber and gradually improved over the years by several researchers, such as Alex Graves, Haşim Sak, ¹² and Wojciech Zaremba. ¹³ If you consider the LSTM cell as a black box, it can be used very much like a basic cell, except it will perform much better; training will converge faster, and it will detect longer-term patterns in the data. In Keras, you can simply use the LSTM layer instead of the SimpleRNN layer:

```
model = tf.keras.Sequential([
    tf.keras.layers.LSTM(32, return_sequences=True, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
```

Alternatively, you could use the general-purpose `tf.keras.layers.RNN` layer, giving it an `LSTMCell` as an argument. However, the LSTM layer uses an optimized implementation when running on a GPU (see [Chapter 19](#)), so in general it is preferable to use it (the RNN layer is mostly useful when you define custom cells, as we did earlier).

So how does an LSTM cell work? Its architecture is shown in [Figure 15-12](#). If you don't look at what's inside the box, the LSTM cell looks exactly like a regular cell, except that its state is split into two vectors: $\mathbf{h}_{(t)}$ and $\mathbf{c}_{(t)}$ ("c"

stands for “cell”). You can think of $\mathbf{h}_{(t)}$ as the short-term state and $\mathbf{c}_{(t)}$ as the long-term state.

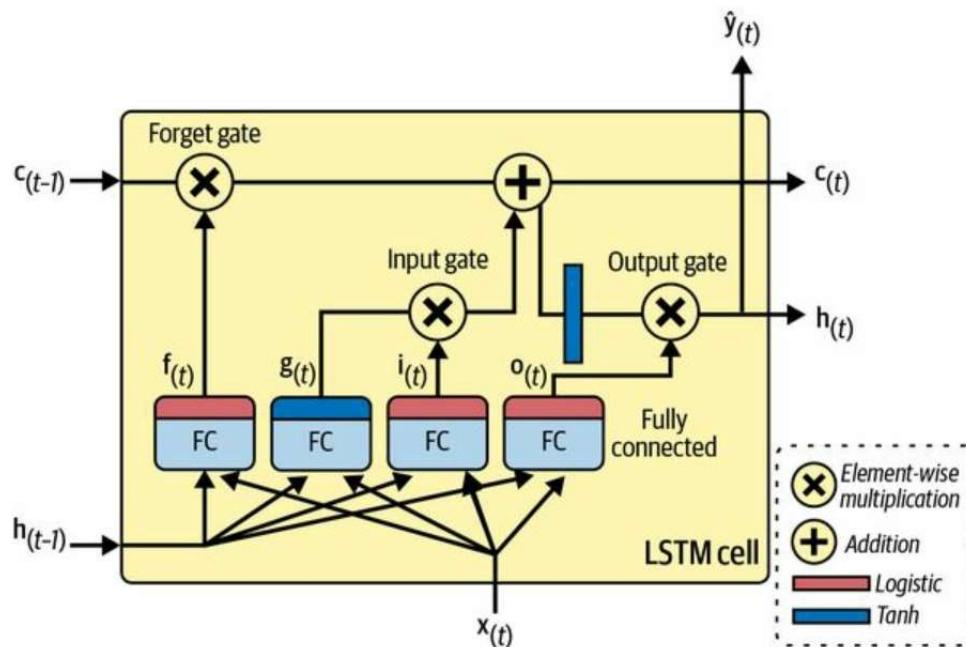


Figure 15-12. An LSTM cell

Now let's open the box! The key idea is that the network can learn what to store in the long-term state, what to throw away, and what to read from it. As the long-term state $\mathbf{c}_{(t-1)}$ traverses the network from left to right, you can see that it first goes through a *forget gate*, dropping some memories, and then it adds some new memories via the addition operation (which adds the memories that were selected by an *input gate*). The result $\mathbf{c}_{(t)}$ is sent straight out, without any further transformation. So, at each time step, some memories are dropped and some memories are added. Moreover, after the addition operation, the long-term state is copied and passed through the tanh function, and then the result is filtered by the *output gate*. This produces the short-term state $\mathbf{h}_{(t)}$ (which is equal to the cell's output for this time step, $\mathbf{y}_{(t)}$). Now let's look at where new memories come from and how the gates work.

First, the current input vector $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t-1)}$ are fed to four different fully connected layers. They all serve a different

purpose:

- The main layer is the one that outputs $\mathbf{g}_{(t)}$. It has the usual role of analyzing the current inputs $\mathbf{x}_{(t)}$ and the previous (short-term) state $\mathbf{h}_{(t-1)}$. In a basic cell, there is nothing other than this layer, and its output goes straight out to $\mathbf{y}_{(t)}$ and $\mathbf{h}_{(t)}$. But in an LSTM cell, this layer's output does not go straight out; instead its most important parts are stored in the long-term state (and the rest is dropped).
- The three other layers are *gate controllers*. Since they use the logistic activation function, the outputs range from 0 to 1. As you can see, the gate controllers' outputs are fed to element-wise multiplication operations: if they output 0s they close the gate, and if they output 1s they open it. Specifically:
 - The *forget gate* (controlled by $\mathbf{f}_{(t)}$) controls which parts of the long-term state should be erased.
 - The *input gate* (controlled by $\mathbf{i}_{(t)}$) controls which parts of $\mathbf{g}_{(t)}$ should be added to the long-term state.
 - Finally, the *output gate* (controlled by $\mathbf{o}_{(t)}$) controls which parts of the long-term state should be read and output at this time step, both to $\mathbf{h}_{(t)}$ and to $\mathbf{y}_{(t)}$.

In short, an LSTM cell can learn to recognize an important input (that's the role of the input gate), store it in the long-term state, preserve it for as long as it is needed (that's the role of the forget gate), and extract it whenever it is needed. This explains why these cells have been amazingly successful at capturing long-term patterns in time series, long texts, audio recordings, and more.

Equation 15-4 summarizes how to compute the cell's long-term state, its short-term state, and its output at each time step for a single instance (the equations for a whole mini-batch are very similar).

Equation 15-4. LSTM computations

$$\begin{aligned}
i(t) &= \sigma(W_{xi}^T x(t) + W_{hi}^T h(t-1) + b_i) \\
f(t) &= \sigma(W_{xf}^T x(t) + W_{hf}^T h(t-1) + b_f) \\
o(t) &= \sigma(W_{xo}^T x(t) + W_{ho}^T h(t-1) + b_o) \\
g(t) &= \tanh(W_{xg}^T x(t) + W_{hg}^T h(t-1) + b_g) \\
c(t) &= f(t) \otimes c(t-1) + i(t) \otimes g(t) \\
y(t) &= h(t) = o(t) \otimes \tanh(c(t))
\end{aligned}$$

In this equation:

- \mathbf{W}_{xi} , \mathbf{W}_{xf} , \mathbf{W}_{xo} , and \mathbf{W}_{xg} are the weight matrices of each of the four layers for their connection to the input vector $\mathbf{x}_{(t)}$.
- \mathbf{W}_{hi} , \mathbf{W}_{hf} , \mathbf{W}_{ho} , and \mathbf{W}_{hg} are the weight matrices of each of the four layers for their connection to the previous short-term state $\mathbf{h}_{(t-1)}$.
- \mathbf{b}_i , \mathbf{b}_f , \mathbf{b}_o , and \mathbf{b}_g are the bias terms for each of the four layers. Note that TensorFlow initializes \mathbf{b}_f to a vector full of 1s instead of 0s. This prevents forgetting everything at the beginning of training.

There are several variants of the LSTM cell. One particularly popular variant is the GRU cell, which we will look at now.

GRU cells

The *gated recurrent unit* (GRU) cell (see [Figure 15-13](#)) was proposed by Kyunghyun Cho et al. in a [2014 paper](#)¹⁴ that also introduced the encoder–decoder network we discussed earlier.

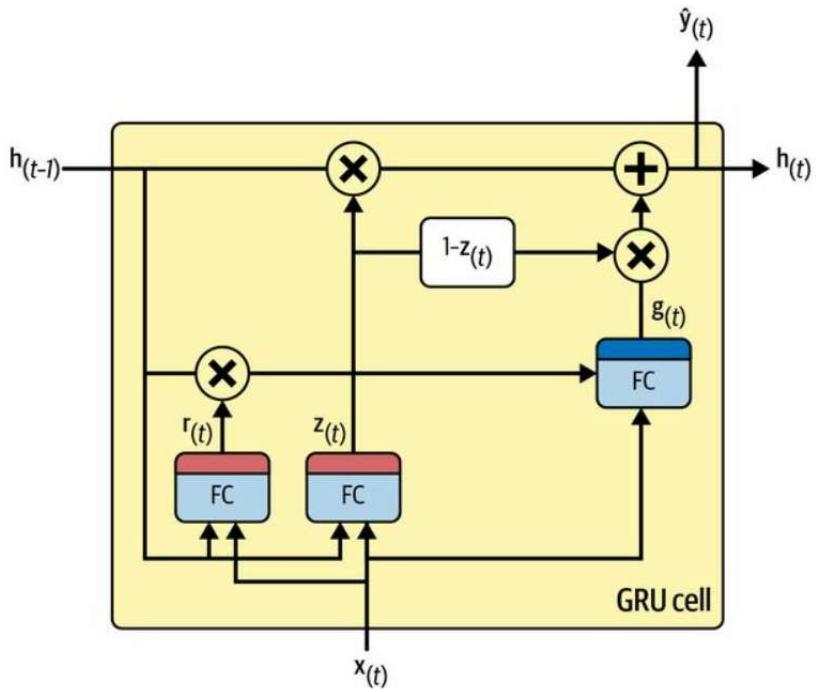


Figure 15-13. GRU cell

The GRU cell is a simplified version of the LSTM cell, and it seems to perform just as well ¹⁵ (which explains its growing popularity). These are the main simplifications:

- Both state vectors are merged into a single vector $\mathbf{h}_{(t)}$.
- A single gate controller $\mathbf{z}_{(t)}$ controls both the forget gate and the input gate. If the gate controller outputs a 1, the forget gate is open ($= 1$) and the input gate is closed ($1 - 1 = 0$). If it outputs a 0, the opposite happens. In other words, whenever a memory must be stored, the location where it will be stored is erased first. This is actually a frequent variant to the LSTM cell in and of itself.
- There is no output gate; the full state vector is output at every time step. However, there is a new gate controller $\mathbf{r}_{(t)}$ that controls which part of the previous state will be shown to the main layer ($\mathbf{g}_{(t)}$).

Equation 15-5 summarizes how to compute the cell's state at each time step for a single instance.

Equation 15-5. GRU computations

$$\begin{aligned} z(t) &= \sigma(W_{xz}^T x(t) + W_{hz}^T h(t-1) + b_z) \\ r(t) &= \sigma(W_{xr}^T x(t) + W_{hr}^T h(t-1) + b_r) \\ g(t) &= \tanh(W_{xg}^T x(t) + W_{hg}^T (r(t) \otimes h(t-1)) + b_g) \\ h(t) &= z(t) \otimes h(t-1) + (1 - z(t)) \otimes g(t) \end{aligned}$$

Keras provides a `tf.keras.layers.GRU` layer: using it is just a matter of replacing `SimpleRNN` or `LSTM` with `GRU`. It also provides a `tf.keras.layers.GRUCell`, in case you want to create a custom cell based on a `GRU` cell.

LSTM and GRU cells are one of the main reasons behind the success of RNNs. Yet while they can tackle much longer sequences than simple RNNs, they still have a fairly limited short-term memory, and they have a hard time learning long-term patterns in sequences of 100 time steps or more, such as audio samples, long time series, or long sentences. One way to solve this is to shorten the input sequences; for example, using 1D convolutional layers.

Using 1D convolutional layers to process sequences

In [Chapter 14](#), we saw that a 2D convolutional layer works by sliding several fairly small kernels (or filters) across an image, producing multiple 2D feature maps (one per kernel). Similarly, a 1D convolutional layer slides several kernels across a sequence, producing a 1D feature map per kernel. Each kernel will learn to detect a single very short sequential pattern (no longer than the kernel size). If you use 10 kernels, then the layer's output will be composed of 10 1D sequences (all of the same length), or equivalently you can view this output as a single 10D sequence. This means that you can build a neural network composed of a mix of recurrent layers and 1D convolutional layers (or even 1D pooling layers). If you use a 1D convolutional layer with a stride of 1 and "same" padding, then the output sequence will have the same length as the input sequence. But if you use "valid" padding or a stride greater than 1, then the output sequence will be shorter than the input sequence, so make sure you adjust the targets accordingly.

For example, the following model is the same as earlier, except it starts with a 1D convolutional layer that downsamples the input sequence by a factor of 2, using a stride of 2. The kernel size is larger than the stride, so all inputs will be used to compute the layer's output, and therefore the model can learn to preserve the useful information, dropping only the unimportant details. By shortening the sequences the convolutional layer may help the GRU layers detect longer patterns, so we can afford to double the input sequence length to 112 days. Note that we must also crop off the first three time steps in the targets: indeed, the kernel's size is 4, so the first output of the convolutional layer will be based on the input time steps 0 to 3, and the first forecasts will be for time steps 4 to 17 (instead of time steps 1 to 14). Moreover, we must downsample the targets by a factor of 2, because of the stride:

```
conv_rnn_model = tf.keras.Sequential([
    tf.keras.layers.Conv1D(filters=32, kernel_size=4, strides=2,
        activation="relu", input_shape=[None, 5]),
    tf.keras.layers.GRU(32, return_sequences=True),
    tf.keras.layers.Dense(14)
])

longer_train = to_seq2seq_dataset(mulvar_train, seq_length=112,
    shuffle=True, seed=42)
longer_valid = to_seq2seq_dataset(mulvar_valid, seq_length=112)
downsampled_train = longer_train.map(lambda X, Y: (X, Y[:, 3::2]))
downsampled_valid = longer_valid.map(lambda X, Y: (X, Y[:, 3::2]))
[...] # compile and fit the model using the downsampled datasets
```

If you train and evaluate this model, you will find that it outperforms the previous model (by a small margin). In fact, it is actually possible to use only 1D convolutional layers and drop the recurrent layers entirely!

WaveNet

In a [2016 paper](#), ¹⁶ Aaron van den Oord and other DeepMind researchers introduced a novel architecture called *WaveNet*. They stacked 1D convolutional layers, doubling the dilation rate (how spread apart each neuron's inputs are) at every layer: the first convolutional layer gets a glimpse of just two time steps at a time, while the next one sees four time steps (its receptive field is four time steps long), the next one sees eight time

steps, and so on (see [Figure 15-14](#)). This way, the lower layers learn short-term patterns, while the higher layers learn long-term patterns. Thanks to the doubling dilation rate, the network can process extremely large sequences very efficiently.

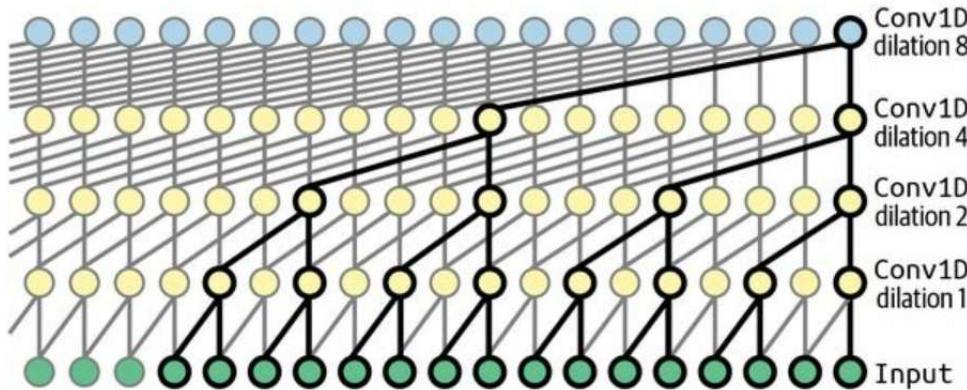


Figure 15-14. WaveNet architecture

The authors of the paper actually stacked 10 convolutional layers with dilation rates of 1, 2, 4, 8, ..., 256, 512, then they stacked another group of 10 identical layers (also with dilation rates 1, 2, 4, 8, ..., 256, 512), then again another identical group of 10 layers. They justified this architecture by pointing out that a single stack of 10 convolutional layers with these dilation rates will act like a super-efficient convolutional layer with a kernel of size 1,024 (except way faster, more powerful, and using significantly fewer parameters). They also left-padded the input sequences with a number of zeros equal to the dilation rate before every layer, to preserve the same sequence length throughout the network.

Here is how to implement a simplified WaveNet to tackle the same sequences as earlier: [17](#)

```
wavenet_model = tf.keras.Sequential()
wavenet_model.add(tf.keras.layers.Input(shape=[None, 5]))
for rate in (1, 2, 4, 8) * 2:
    wavenet_model.add(tf.keras.layers.Conv1D(
        filters=32, kernel_size=2, padding="causal", activation="relu",
        dilation_rate=rate))
wavenet_model.add(tf.keras.layers.Conv1D(filters=14, kernel_size=1))
```

This Sequential model starts with an explicit input layer—this is simpler than trying to set `input_shape` only on the first layer. Then it continues with a 1D convolutional layer using "causal" padding, which is like "same" padding except that the zeros are appended only at the start of the input sequence, instead of on both sides. This ensures that the convolutional layer does not peek into the future when making predictions. Then we add similar pairs of layers using growing dilation rates: 1, 2, 4, 8, and again 1, 2, 4, 8. Finally, we add the output layer: a convolutional layer with 14 filters of size 1 and without any activation function. As we saw earlier, such a convolutional layer is equivalent to a Dense layer with 14 units. Thanks to the causal padding, every convolutional layer outputs a sequence of the same length as its input sequence, so the targets we use during training can be the full 112-day sequences: no need to crop them or downsample them.

The models we've discussed in this section offer similar performance for the ridership forecasting task, but they may vary significantly depending on the task and the amount of available data. In the WaveNet paper, the authors achieved state-of-the-art performance on various audio tasks (hence the name of the architecture), including text-to-speech tasks, producing incredibly realistic voices across several languages. They also used the model to generate music, one audio sample at a time. This feat is all the more impressive when you realize that a single second of audio can contain tens of thousands of time steps—even LSTMs and GRUs cannot handle such long sequences.

WARNING

If you evaluate our best Chicago ridership models on the test period, starting in 2020, you will find that they perform much worse than expected! Why is that? Well, that's when the Covid-19 pandemic started, which greatly affected public transportation. As mentioned earlier, these models will only work well if the patterns they learned from the past continue in the future. In any case, before deploying a model to production, verify that it works well on recent data. And once it's in production, make sure to monitor its performance regularly.

With that, you can now tackle all sorts of time series! In [Chapter 16](#), we will continue to explore RNNs, and we will see how they can tackle various NLP tasks as well.

Exercises

1. Can you think of a few applications for a sequence-to-sequence RNN? What about a sequence-to-vector RNN, and a vector-to-sequence RNN?
2. How many dimensions must the inputs of an RNN layer have? What does each dimension represent? What about its outputs?
3. If you want to build a deep sequence-to-sequence RNN, which RNN layers should have `return_sequences=True`? What about a sequence-to-vector RNN?
4. Suppose you have a daily univariate time series, and you want to forecast the next seven days. Which RNN architecture should you use?
5. What are the main difficulties when training RNNs? How can you handle them?
6. Can you sketch the LSTM cell's architecture?
7. Why would you want to use 1D convolutional layers in an RNN?
8. Which neural network architecture could you use to classify videos?
9. Train a classification model for the SketchRNN dataset, available in TensorFlow Datasets.
10. Download the [Bach chorales](#) dataset and unzip it. It is composed of 382 chorales composed by Johann Sebastian Bach. Each chorale is 100 to 640 time steps long, and each time step contains 4 integers, where each integer corresponds to a note's index on a piano (except for the value 0, which means that no note is played). Train a model—recurrent, convolutional, or both—that can predict the next time step (four notes), given a sequence of time steps from a chorale. Then use this model to generate Bach-like music, one note at a time: you can do this by giving the model the start of a chorale and asking it to predict the next time step, then appending these time steps to the input sequence and asking

the model for the next note, and so on. Also make sure to check out [Google's Coconet model](#), which was used for a nice Google doodle about Bach.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

-
- 1 Note that many researchers prefer to use the hyperbolic tangent (\tanh) activation function in RNNs rather than the ReLU activation function. For example, see Vu Pham et al.'s [2013 paper](#) "Dropout Improves Recurrent Neural Networks for Handwriting Recognition". ReLU-based RNNs are also possible, as shown in Quoc V. Le et al.'s [2015 paper](#) "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units".
 - 2 Nal Kalchbrenner and Phil Blunsom, "Recurrent Continuous Translation Models", *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (2013): 1700–1709.
 - 3 The latest data from the Chicago Transit Authority is available at the [Chicago Data Portal](#).
 - 4 There are other more principled approaches to selecting good hyperparameters, based on analyzing the *autocorrelation function* (ACF) and *partial autocorrelation function* (PACF), or minimizing the AIC or BIC metrics (introduced in [Chapter 9](#)) to penalize models that use too many parameters and reduce the risk of overfitting the data, but grid search is a good place to start. For more details on the ACF-PACF approach, check out this very nice [post by Jason Brownlee](#).
 - 5 Note that the validation period starts on the 1st of January 2019, so the first prediction is for the 26th of February 2019, eight weeks later. When we evaluated the baseline models we used predictions starting on the 1st of March instead, but this should be close enough.
 - 6 Feel free to play around with this model. For example, you can try forecasting both the bus and rail ridership for the next 14 days. You'll need to tweak the targets to include both, and make your model output 28 forecasts instead of 14.
 - 7 César Laurent et al., "Batch Normalized Recurrent Neural Networks", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2016): 2657–2661.
 - 8 Jimmy Lei Ba et al., "Layer Normalization", arXiv preprint arXiv:1607.06450 (2016).
 - 9 It would have been simpler to inherit from `SimpleRNNCell` instead so that we wouldn't have to create an internal `SimpleRNNCell` or handle the `state_size` and `output_size` attributes, but the goal here was to show how to create a custom cell from scratch.
 - 10 A character from the animated movies *Finding Nemo* and *Finding Dory* who has short-term memory loss.
 - 11 Sepp Hochreiter and Jürgen Schmidhuber, "Long Short-Term Memory", *Neural Computation* 9, no. 8 (1997): 1735–1780.

- ¹² Haşim Sak et al., “Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition”, arXiv preprint arXiv:1402.1128 (2014).
- ¹³ Wojciech Zaremba et al., “Recurrent Neural Network Regularization”, arXiv preprint arXiv:1409.2329 (2014).
- ¹⁴ Kyunghyun Cho et al., “Learning Phrase Representations Using RNN Encoder–Decoder for Statistical Machine Translation”, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing* (2014): 1724–1734.
- ¹⁵ See Klaus Greff et al., “LSTM: A Search Space Odyssey”, *IEEE Transactions on Neural Networks and Learning Systems* 28, no. 10 (2017): 2222–2232. This paper seems to show that all LSTM variants perform roughly the same.
- ¹⁶ Aaron van den Oord et al., “WaveNet: A Generative Model for Raw Audio”, arXiv preprint arXiv:1609.03499 (2016).
- ¹⁷ The complete WaveNet uses a few more tricks, such as skip connections like in a ResNet, and *gated activation units* similar to those found in a GRU cell. See this chapter’s notebook for more details.

Chapter 16. Natural Language Processing with RNNs and Attention

When Alan Turing imagined his famous [Turing test](#)¹ in 1950, he proposed a way to evaluate a machine's ability to match human intelligence. He could have tested for many things, such as the ability to recognize cats in pictures, play chess, compose music, or escape a maze, but, interestingly, he chose a linguistic task. More specifically, he devised a *chatbot* capable of fooling its interlocutor into thinking it was human.² This test does have its weaknesses: a set of hardcoded rules can fool unsuspecting or naive humans (e.g., the machine could give vague predefined answers in response to some keywords, it could pretend that it is joking or drunk to get a pass on its weirdest answers, or it could escape difficult questions by answering them with its own questions), and many aspects of human intelligence are utterly ignored (e.g., the ability to interpret nonverbal communication such as facial expressions, or to learn a manual task). But the test does highlight the fact that mastering language is arguably *Homo sapiens*'s greatest cognitive ability.

Can we build a machine that can master written and spoken language? This is the ultimate goal of NLP research, but it's a bit too broad, so in practice researchers focus on more specific tasks, such as text classification, translation, summarization, question answering, and many more.

A common approach for natural language tasks is to use recurrent neural networks. We will therefore continue to explore RNNs (introduced in [Chapter 15](#)), starting with a *character RNN*, or *char-RNN*, trained to predict the next character in a sentence. This will allow us to generate some original text. We will first use a *stateless RNN* (which learns on random portions of text at each iteration, without any information on the rest of the text), then we

will build a *stateful RNN* (which preserves the hidden state between training iterations and continues reading where it left off, allowing it to learn longer patterns). Next, we will build an RNN to perform sentiment analysis (e.g., reading movie reviews and extracting the rater's feeling about the movie), this time treating sentences as sequences of words, rather than characters. Then we will show how RNNs can be used to build an encoder–decoder architecture capable of performing neural machine translation (NMT), translating English to Spanish.

In the second part of this chapter, we will explore *attention mechanisms*. As their name suggests, these are neural network components that learn to select the part of the inputs that the rest of the model should focus on at each time step. First, we will boost the performance of an RNN-based encoder–decoder architecture using attention. Next, we will drop RNNs altogether and use a very successful attention-only architecture, called the *transformer*, to build a translation model. We will then discuss some of the most important advances in NLP in the last few years, including incredibly powerful language models such as GPT and BERT, both based on transformers. Lastly, I will show you how to get started with the excellent Transformers library by Hugging Face.

Let's start with a simple and fun model that can write like Shakespeare (sort of).

Generating Shakespearean Text Using a Character RNN

In a famous [2015 blog post](#) titled “The Unreasonable Effectiveness of Recurrent Neural Networks”, Andrej Karpathy showed how to train an RNN to predict the next character in a sentence. This *char-RNN* can then be used to generate novel text, one character at a time. Here is a small sample of the text generated by a char-RNN model after it was trained on all of Shakespeare’s works:

PANDARUS:

*Alas, I think he shall be come approached and the day
When little strain would be attain’d into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.*

Not exactly a masterpiece, but it is still impressive that the model was able to learn words, grammar, proper punctuation, and more, just by learning to predict the next character in a sentence. This is our first example of a *language model*; similar (but much more powerful) language models, discussed later in this chapter, are at the core of modern NLP. In the remainder of this section we’ll build a char-RNN step by step, starting with the creation of the dataset.

Creating the Training Dataset

First, using Keras's handy `tf.keras.utils.get_file()` function, let's download all of Shakespeare's works. The data is loaded from Andrej Karpathy's [char-rnn project](#):

```
import tensorflow as tf

shakespeare_url = "https://homl.info/shakespeare" # shortcut URL
filepath = tf.keras.utils.get_file("shakespeare.txt", shakespeare_url)
with open(filepath) as f:
    shakespeare_text = f.read()
```

Let's print the first few lines:

```
>>> print(shakespeare_text[:80])
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.
```

Looks like Shakespeare all right!

Next, we'll use a `tf.keras.layers.TextVectorization` layer (introduced in [Chapter 13](#)) to encode this text. We set `split="character"` to get character-level encoding rather than the default word-level encoding, and we use `standardize="lower"` to convert the text to lowercase (which will simplify the task):

```
text_vec_layer = tf.keras.layers.TextVectorization(split="character",
                                                 standardize="lower")
text_vec_layer.adapt([shakespeare_text])
encoded = text_vec_layer([shakespeare_text])[0]
```

Each character is now mapped to an integer, starting at 2. The `TextVectorization` layer reserved the value 0 for padding tokens, and it reserved 1 for unknown characters. We won't need either of these tokens for

now, so let's subtract 2 from the character IDs and compute the number of distinct characters and the total number of characters:

```
encoded -= 2 # drop tokens 0 (pad) and 1 (unknown), which we will not use
n_tokens = text_vec_layer.vocabulary_size() - 2 # number of distinct chars = 39
dataset_size = len(encoded) # total number of chars = 1,115,394
```

Next, just like we did in [Chapter 15](#), we can turn this very long sequence into a dataset of windows that we can then use to train a sequence-to-sequence RNN. The targets will be similar to the inputs, but shifted by one time step into the “future”. For example, one sample in the dataset may be a sequence of character IDs representing the text “to be or not to b” (without the final “e”), and the corresponding target—a sequence of character IDs representing the text “o be or not to be” (with the final “e”, but without the leading “t”). Let's write a small utility function to convert a long sequence of character IDs into a dataset of input/target window pairs:

```
def to_dataset(sequence, length, shuffle=False, seed=None, batch_size=32):
    ds = tf.data.Dataset.from_tensor_slices(sequence)
    ds = ds.window(length + 1, shift=1, drop_remainder=True)
    ds = ds.flat_map(lambda window_ds: window_ds.batch(length + 1))
    if shuffle:
        ds = ds.shuffle(buffer_size=100_000, seed=seed)
    ds = ds.batch(batch_size)
    return ds.map(lambda window: (window[:, :-1], window[:, 1:])).prefetch(1)
```

This function starts much like the `to_windows()` custom utility function we created in [Chapter 15](#):

- It takes a sequence as input (i.e., the encoded text), and creates a dataset containing all the windows of the desired length.
- It increases the length by one, since we need the next character for the target.
- Then, it shuffles the windows (optionally), batches them, splits them into input/output pairs, and activates prefetching.

[Figure 16-1](#) summarizes the dataset preparation steps: it shows windows of

length 11, and a batch size of 3. The start index of each window is indicated next to it.

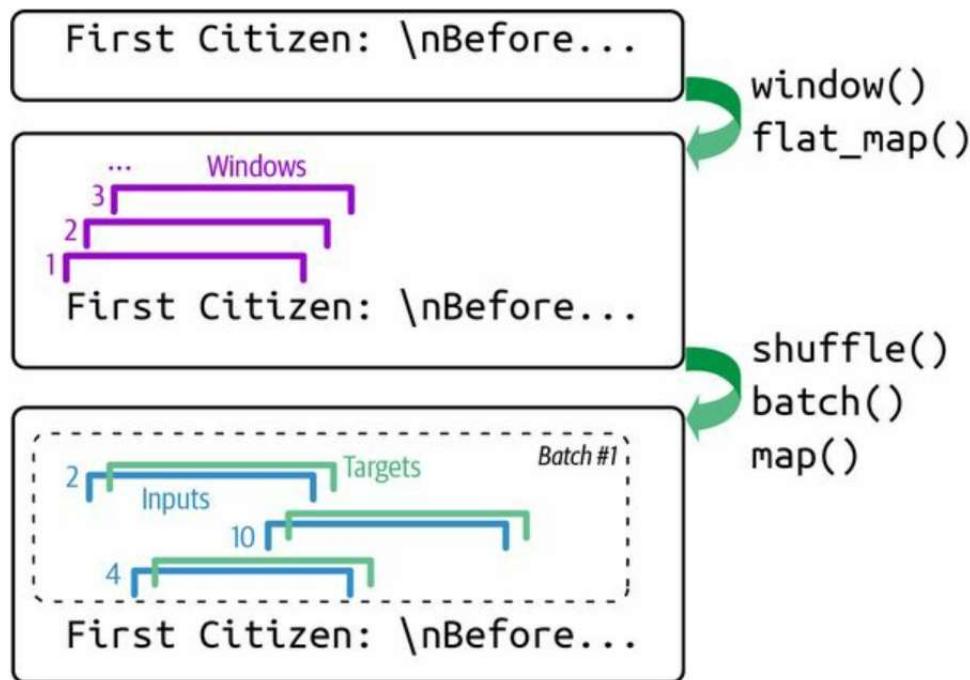


Figure 16-1. Preparing a dataset of shuffled windows

Now we're ready to create the training set, the validation set, and the test set. We will use roughly 90% of the text for training, 5% for validation, and 5% for testing:

```
length = 100
tf.random.set_seed(42)
train_set = to_dataset(encoded[:1_000_000], length=length, shuffle=True,
                      seed=42)
valid_set = to_dataset(encoded[1_000_000:1_060_000], length=length)
test_set = to_dataset(encoded[1_060_000:], length=length)
```

TIP

We set the window length to 100, but you can try tuning it: it's easier and faster to train RNNs on shorter input sequences, but the RNN will not be able to learn any pattern longer than length, so don't make it too small.

That's it! Preparing the dataset was the hardest part. Now let's create the model.

Building and Training the Char-RNN Model

Since our dataset is reasonably large, and modeling language is quite a difficult task, we need more than a simple RNN with a few recurrent neurons. Let's build and train a model with one GRU layer composed of 128 units (you can try tweaking the number of layers and units later, if needed):

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_tokens, output_dim=16),
    tf.keras.layers.GRU(128, return_sequences=True),
    tf.keras.layers.Dense(n_tokens, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam",
               metrics=["accuracy"])
model_ckpt = tf.keras.callbacks.ModelCheckpoint(
    "my_shakespeare_model", monitor="val_accuracy", save_best_only=True)
history = model.fit(train_set, validation_data=valid_set, epochs=10,
                     callbacks=[model_ckpt])
```

Let's go over this code:

- We use an Embedding layer as the first layer, to encode the character IDs (embeddings were introduced in [Chapter 13](#)). The Embedding layer's number of input dimensions is the number of distinct character IDs, and the number of output dimensions is a hyperparameter you can tune—we'll set it to 16 for now. Whereas the inputs of the Embedding layer will be 2D tensors of shape [*batch size*, *window length*], the output of the Embedding layer will be a 3D tensor of shape [*batch size*, *window length*, *embedding size*].
- We use a Dense layer for the output layer: it must have 39 units (*n_tokens*) because there are 39 distinct characters in the text, and we want to output a probability for each possible character (at each time step). The 39 output probabilities should sum up to 1 at each time step, so we apply the softmax activation function to the outputs of the Dense layer.
- Lastly, we compile this model, using the

"sparse_categorical_crossentropy" loss and a Nadam optimizer, and we train the model for several epochs,³ using a ModelCheckpoint callback to save the best model (in terms of validation accuracy) as training progresses.

TIP

If you are running this code on Colab with a GPU activated, then training should take roughly one to two hours. You can reduce the number of epochs if you don't want to wait that long, but of course the model's accuracy will probably be lower. If the Colab session times out, make sure to reconnect quickly, or else the Colab runtime will be destroyed.

This model does not handle text preprocessing, so let's wrap it in a final model containing the `tf.keras.layers.TextVectorization` layer as the first layer, plus a `tf.keras.layers.Lambda` layer to subtract 2 from the character IDs since we're not using the padding and unknown tokens for now:

```
shakespeare_model = tf.keras.Sequential([
    text_vec_layer,
    tf.keras.layers.Lambda(lambda X: X - 2), # no <PAD> or <UNK> tokens
    model
])
```

And now let's use it to predict the next character in a sentence:

```
>>> y_proba = shakespeare_model.predict(["To be or not to b"])[0, -1]
>>> y_pred = tf.argmax(y_proba) # choose the most probable character ID
>>> text_vec_layer.get_vocabulary()[y_pred + 2]
'e'
```

Great, the model correctly predicted the next character. Now let's use this model to pretend we're Shakespeare!

Generating Fake Shakespearean Text

To generate new text using the char-RNN model, we could feed it some text, make the model predict the most likely next letter, add it to the end of the text, then give the extended text to the model to guess the next letter, and so on. This is called *greedy decoding*. But in practice this often leads to the same words being repeated over and over again. Instead, we can sample the next character randomly, with a probability equal to the estimated probability, using TensorFlow's `tf.random.categorical()` function. This will generate more diverse and interesting text. The `categorical()` function samples random class indices, given the class log probabilities (logits). For example:

```
>>> log_probas = tf.math.log([[0.5, 0.4, 0.1]]) # probas = 50%, 40%, and 10%
>>> tf.random.set_seed(42)
>>> tf.random.categorical(log_probas, num_samples=8) # draw 8 samples
<tf.Tensor: shape=(1, 8), dtype=int64, numpy=array([[0, 1, 0, 2, 1, 0, 0, 1]])>
```

To have more control over the diversity of the generated text, we can divide the logits by a number called the *temperature*, which we can tweak as we wish. A temperature close to zero favors high-probability characters, while a high temperature gives all characters an equal probability. Lower temperatures are typically preferred when generating fairly rigid and precise text, such as mathematical equations, while higher temperatures are preferred when generating more diverse and creative text. The following `next_char()` custom helper function uses this approach to pick the next character to add to the input text:

```
def next_char(text, temperature=1):
    y_proba = shakespeare_model.predict([text])[0, -1:]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits, num_samples=1)[0, 0]
    return text_vec_layer.get_vocabulary()[char_id + 2]
```

Next, we can write another small helper function that will repeatedly call `next_char()` to get the next character and append it to the given text:

```
def extend_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(text, temperature)
    return text
```

We are now ready to generate some text! Let's try with different temperature values:

```
>>> tf.random.set_seed(42)
>>> print(extend_text("To be or not to be", temperature=0.01))
To be or not to be the duke
as it is a proper strange death,
and the
>>> print(extend_text("To be or not to be", temperature=1))
To be or not to behold?

second push:
gremio, lord all, a sistermen,
>>> print(extend_text("To be or not to be", temperature=100))
To be or not to bef ,mt'&o3fpadm!$  
wh!nse?bws3est--vgerdjw?c-y-ewzqnq
```

Shakespeare seems to be suffering from a heatwave. To generate more convincing text, a common technique is to sample only from the top k characters, or only from the smallest set of top characters whose total probability exceeds some threshold (this is called *nucleus sampling*).

Alternatively, you could try using *beam search*, which we will discuss later in this chapter, or using more GRU layers and more neurons per layer, training for longer, and adding some regularization if needed. Also note that the model is currently incapable of learning patterns longer than length, which is just 100 characters. You could try making this window larger, but it will also make training harder, and even LSTM and GRU cells cannot handle very long sequences. An alternative approach is to use a stateful RNN.

Stateful RNN

Until now, we have only used *stateless RNNs*: at each training iteration the model starts with a hidden state full of zeros, then it updates this state at each time step, and after the last time step, it throws it away as it is not needed anymore. What if we instructed the RNN to preserve this final state after processing a training batch and use it as the initial state for the next training batch? This way the model could learn long-term patterns despite only backpropagating through short sequences. This is called a *stateful RNN*. Let's go over how to build one.

First, note that a stateful RNN only makes sense if each input sequence in a batch starts exactly where the corresponding sequence in the previous batch left off. So the first thing we need to do to build a stateful RNN is to use sequential and nonoverlapping input sequences (rather than the shuffled and overlapping sequences we used to train stateless RNNs). When creating the `tf.data.Dataset`, we must therefore use `shift=length` (instead of `shift=1`) when calling the `window()` method. Moreover, we must *not* call the `shuffle()` method.

Unfortunately, batching is much harder when preparing a dataset for a stateful RNN than it is for a stateless RNN. Indeed, if we were to call `batch(32)`, then 32 consecutive windows would be put in the same batch, and the following batch would not continue each of these windows where it left off. The first batch would contain windows 1 to 32 and the second batch would contain windows 33 to 64, so if you consider, say, the first window of each batch (i.e., windows 1 and 33), you can see that they are not consecutive. The simplest solution to this problem is to just use a batch size of 1. The following `to_dataset_for_stateful_rnn()` custom utility function uses this strategy to prepare a dataset for a stateful RNN:

```
def to_dataset_for_stateful_rnn(sequence, length):
    ds = tf.data.Dataset.from_tensor_slices(sequence)
    ds = ds.window(length + 1, shift=length, drop_remainder=True)
    ds = ds.flat_map(lambda window: window.batch(length + 1)).batch(1)
    return ds.map(lambda window: (window[:, :-1], window[:, 1:])).prefetch(1)
```

```

stateful_train_set = to_dataset_for_stateful_rnn(encoded[:1_000_000], length)
stateful_valid_set = to_dataset_for_stateful_rnn(encoded[1_000_000:1_060_000],
                                              length)
stateful_test_set = to_dataset_for_stateful_rnn(encoded[1_060_000:], length)

```

Figure 16-2 summarizes the main steps of this function.

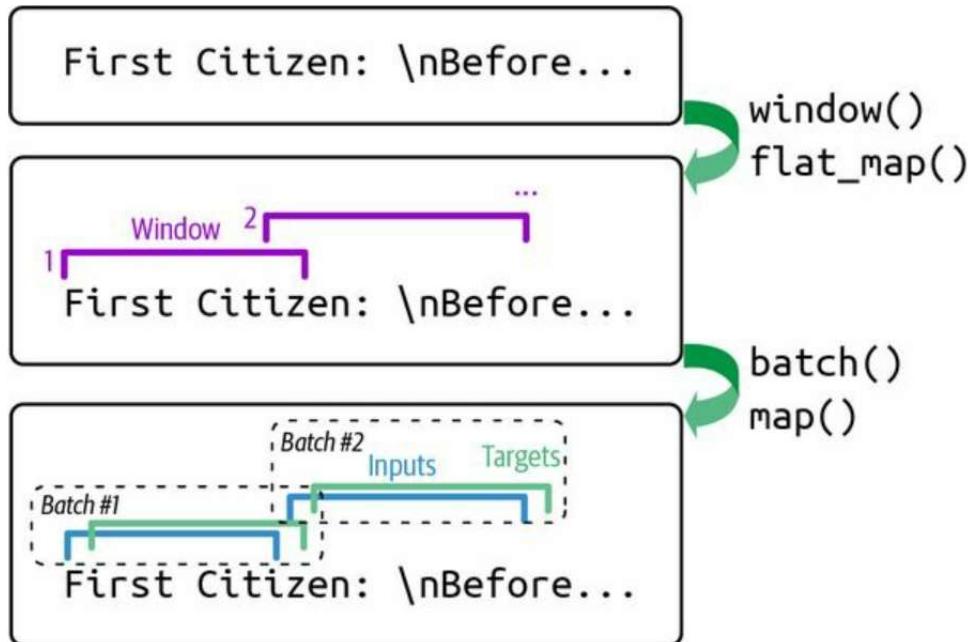


Figure 16-2. Preparing a dataset of consecutive sequence fragments for a stateful RNN

Batching is harder, but it is not impossible. For example, we could chop Shakespeare's text into 32 texts of equal length, create one dataset of consecutive input sequences for each of them, and finally use `tf.data.Dataset.zip(datasets).map(lambda *windows: tf.stack(windows))` to create proper consecutive batches, where the n^{th} input sequence in a batch starts off exactly where the n^{th} input sequence ended in the previous batch (see the notebook for the full code).

Now, let's create the stateful RNN. We need to set the `stateful` argument to `True` when creating each recurrent layer, and because the stateful RNN needs to know the batch size (since it will preserve a state for each input sequence

in the batch). Therefore we must set the batch_input_shape argument in the first layer. Note that we can leave the second dimension unspecified, since the input sequences could have any length:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_tokens, output_dim=16,
        batch_input_shape=[1, None]),
    tf.keras.layers.GRU(128, return_sequences=True, stateful=True),
    tf.keras.layers.Dense(n_tokens, activation="softmax")
])
```

At the end of each epoch, we need to reset the states before we go back to the beginning of the text. For this, we can use a small custom Keras callback:

```
class ResetStatesCallback(tf.keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

And now we can compile the model and train it using our callback:

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam",
    metrics=["accuracy"])
history = model.fit(stateful_train_set, validation_data=stateful_valid_set,
    epochs=10, callbacks=[ResetStatesCallback(), model_ckpt])
```

TIP

After this model is trained, it will only be possible to use it to make predictions for batches of the same size as were used during training. To avoid this restriction, create an identical *stateless* model, and copy the stateful model's weights to this model.

Interestingly, although a char-RNN model is just trained to predict the next character, this seemingly simple task actually requires it to learn some higher-level tasks as well. For example, to find the next character after “Great movie, I really”, it’s helpful to understand that the sentence is positive, so what follows is more likely to be the letter “l” (for “loved”) rather than “h” (for “hated”). In fact, a [2017 paper⁴](#) by Alec Radford and other OpenAI

researchers describes how the authors trained a big char-RNN-like model on a large dataset, and found that one of the neurons acted as an excellent sentiment analysis classifier: although the model was trained without any labels, the *sentiment neuron*—as they called it—reached state-of-the-art performance on sentiment analysis benchmarks. This foreshadowed and motivated unsupervised pretraining in NLP.

But before we explore unsupervised pretraining, let's turn our attention to word-level models and how to use them in a supervised fashion for sentiment analysis. In the process, you will learn how to handle sequences of variable lengths using masking.

Sentiment Analysis

Generating text can be fun and instructive, but in real-life projects, one of the most common applications of NLP is text classification—especially sentiment analysis. If image classification on the MNIST dataset is the “Hello world!” of computer vision, then sentiment analysis on the IMDb reviews dataset is the “Hello world!” of natural language processing. The IMDb dataset consists of 50,000 movie reviews in English (25,000 for training, 25,000 for testing) extracted from the famous [Internet Movie Database](#), along with a simple binary target for each review indicating whether it is negative (0) or positive (1). Just like MNIST, the IMDb reviews dataset is popular for good reasons: it is simple enough to be tackled on a laptop in a reasonable amount of time, but challenging enough to be fun and rewarding.

Let’s load the IMDb dataset using the TensorFlow Datasets library (introduced in [Chapter 13](#)). We’ll use the first 90% of the training set for training, and the remaining 10% for validation:

```
import tensorflow_datasets as tfds

raw_train_set, raw_valid_set, raw_test_set = tfds.load(
    name="imdb_reviews",
    split=["train[:90%]", "train[90%:]", "test"],
    as_supervised=True
)
tf.random.set_seed(42)
train_set = raw_train_set.shuffle(5000, seed=42).batch(32).prefetch(1)
valid_set = raw_valid_set.batch(32).prefetch(1)
test_set = raw_test_set.batch(32).prefetch(1)
```

TIP

Keras also includes a function for loading the IMDb dataset, if you prefer: `tf.keras.datasets.imdb.load_data()`. The reviews are already preprocessed as sequences of word IDs.

Let's inspect a few reviews:

```
>>> for review, label in raw_train_set.take(4):
...     print(review.numpy().decode("utf-8"))
...     print("Label:", label.numpy())
...
This was an absolutely terrible movie. Don't be lured in by Christopher [...]
Label: 0
I have been known to fall asleep during films, but this is usually due to [...]
Label: 0
Mann photographs the Alberta Rocky Mountains in a superb fashion, and [...]
Label: 0
This is the kind of film for a snowy Sunday afternoon when the rest of the [...]
Label: 1
```

Some reviews are easy to classify. For example, the first review includes the words “terrible movie” in the very first sentence. But in many cases things are not that simple. For example, the third review starts off positively, even though it's ultimately a negative review (label 0).

To build a model for this task, we need to preprocess the text, but this time we will chop it into words instead of characters. For this, we can use the `tf.keras.layers.TextVectorization` layer again. Note that it uses spaces to identify word boundaries, which will not work well in some languages. For example, Chinese writing does not use spaces between words, Vietnamese uses spaces even within words, and German often attaches multiple words together, without spaces. Even in English, spaces are not always the best way to tokenize text: think of “San Francisco” or “#ILoveDeepLearning”.

Fortunately, there are solutions to address these issues. In a [2016 paper](#), ⁵ Rico Sennrich et al. from the University of Edinburgh explored several methods to tokenize and detokenize text at the subword level. This way, even if your model encounters a rare word it has never seen before, it can still reasonably guess what it means. For example, even if the model never saw the word “smartest” during training, if it learned the word “smart” and it also learned that the suffix “est” means “the most”, it can infer the meaning of “smartest”. One of the techniques the authors evaluated is *byte pair encoding* (BPE). BPE works by splitting the whole training set into individual characters (including spaces), then repeatedly merging the most frequent

adjacent pairs until the vocabulary reaches the desired size.

A [2018 paper⁶](#) by Taku Kudo at Google further improved subword tokenization, often removing the need for language-specific preprocessing prior to tokenization. Moreover, the paper proposed a novel regularization technique called *subword regularization*, which improves accuracy and robustness by introducing some randomness in tokenization during training: for example, “New England” may be tokenized as “New” + “England”, or “New” + “Eng” + “land”, or simply “New England” (just one token). Google’s [SentencePiece](#) project provides an open source implementation, which is described in a [paper⁷](#) by Taku Kudo and John Richardson.

The [TensorFlow Text](#) library also implements various tokenization strategies, including [WordPiece⁸](#) (a variant of BPE), and last but not least, the [Tokenizers library by Hugging Face](#) implements a wide range of extremely fast tokenizers.

However, for the IMDb task in English, using spaces for token boundaries should be good enough. So let’s go ahead with creating a TextVectorization layer and adapting it to the training set. We will limit the vocabulary to 1,000 tokens, including the most frequent 998 words plus a padding token and a token for unknown words, since it’s unlikely that very rare words will be important for this task, and limiting the vocabulary size will reduce the number of parameters the model needs to learn:

```
vocab_size = 1000
text_vec_layer = tf.keras.layers.TextVectorization(max_tokens=vocab_size)
text_vec_layer.adapt(train_set.map(lambda reviews, labels: reviews))
```

Finally, we can create the model and train it:

```
embed_size = 128
tf.random.set_seed(42)
model = tf.keras.Sequential([
    text_vec_layer,
    tf.keras.layers.Embedding(vocab_size, embed_size),
    tf.keras.layers.GRU(128),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
```

```
model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=2)
```

The first layer is the TextVectorization layer we just prepared, followed by an Embedding layer that will convert word IDs into embeddings. The embedding matrix needs to have one row per token in the vocabulary (vocab_size) and one column per embedding dimension (this example uses 128 dimensions, but this is a hyperparameter you could tune). Next we use a GRU layer and a Dense layer with a single neuron and the sigmoid activation function, since this is a binary classification task: the model's output will be the estimated probability that the review expresses a positive sentiment regarding the movie. We then compile the model, and we fit it on the dataset we prepared earlier for a couple of epochs (or you can train for longer to get better results).

Sadly, if you run this code, you will generally find that the model fails to learn anything at all: the accuracy remains close to 50%, no better than random chance. Why is that? The reviews have different lengths, so when the TextVectorization layer converts them to sequences of token IDs, it pads the shorter sequences using the padding token (with ID 0) to make them as long as the longest sequence in the batch. As a result, most sequences end with many padding tokens—often dozens or even hundreds of them. Even though we're using a GRU layer, which is much better than a SimpleRNN layer, its short-term memory is still not great, so when it goes through many padding tokens, it ends up forgetting what the review was about! One solution is to feed the model with batches of equal-length sentences (which also speeds up training). Another solution is to make the RNN ignore the padding tokens. This can be done using masking.

Masking

Making the model ignore padding tokens is trivial using Keras: simply add `mask_zero=True` when creating the Embedding layer. This means that padding tokens (whose ID is 0) will be ignored by all downstream layers. That's all! If you retrain the previous model for a few epochs, you will find that the validation accuracy quickly reaches over 80%.

The way this works is that the Embedding layer creates a *mask tensor* equal to `tf.math.not_equal(inputs, 0)`: it is a Boolean tensor with the same shape as the inputs, and it is equal to False anywhere the token IDs are 0, or True otherwise. This mask tensor is then automatically propagated by the model to the next layer. If that layer's `call()` method has a `mask` argument, then it automatically receives the mask. This allows the layer to ignore the appropriate time steps. Each layer may handle the mask differently, but in general they simply ignore masked time steps (i.e., time steps for which the mask is False). For example, when a recurrent layer encounters a masked time step, it simply copies the output from the previous time step.

Next, if the layer's `supports_masking` attribute is True, then the mask is automatically propagated to the next layer. It keeps propagating this way for as long as the layers have `supports_masking=True`. As an example, a recurrent layer's `supports_masking` attribute is True when `return_sequences=True`, but it's False when `return_sequences=False` since there's no need for a mask anymore in this case. So if you have a model with several recurrent layers with `return_sequences=True`, followed by a recurrent layer with `return_sequences=False`, then the mask will automatically propagate up to the last recurrent layer: that layer will use the mask to ignore masked steps, but it will not propagate the mask any further. Similarly, if you set `mask_zero=True` when creating the Embedding layer in the sentiment analysis model we just built, then the GRU layer will receive and use the mask automatically, but it will not propagate it any further, since `return_sequences` is not set to True.

TIP

Some layers need to update the mask before propagating it to the next layer: they do so by implementing the `compute_mask()` method, which takes two arguments: the inputs and the previous mask. It then computes the updated mask and returns it. The default implementation of `compute_mask()` just returns the previous mask unchanged.

Many Keras layers support masking: SimpleRNN, GRU, LSTM, Bidirectional, Dense, TimeDistributed, Add, and a few others (all in the `tf.keras.layers` package). However, convolutional layers (including Conv1D) do not support masking—it's not obvious how they would do so anyway.

If the mask propagates all the way to the output, then it gets applied to the losses as well, so the masked time steps will not contribute to the loss (their loss will be 0). This assumes that the model outputs sequences, which is not the case in our sentiment analysis model.

WARNING

The LSTM and GRU layers have an optimized implementation for GPUs, based on Nvidia's cuDNN library. However, this implementation only supports masking if all the padding tokens are at the end of the sequences. It also requires you to use the default values for several hyperparameters: `activation`, `recurrent_activation`, `recurrent_dropout`, `unroll`, `use_bias`, and `reset_after`. If that's not the case, then these layers will fall back to the (much slower) default GPU implementation.

If you want to implement your own custom layer with masking support, you should add a `mask` argument to the `call()` method, and obviously make the method use the mask. Additionally, if the mask must be propagated to the next layers, then you should set `self.supports_masking=True` in the constructor. If the mask must be updated before it is propagated, then you must implement the `compute_mask()` method.

If your model does not start with an Embedding layer, you may use the `tf.keras.layers.Masking` layer instead: by default, it sets the mask to `tf.math.reduce_any(tf.math.not_equal(X, 0), axis=-1)`, meaning that time steps where

the last dimension is full of zeros will be masked out in subsequent layers.

Using masking layers and automatic mask propagation works best for simple models. It will not always work for more complex models, such as when you need to mix Conv1D layers with recurrent layers. In such cases, you will need to explicitly compute the mask and pass it to the appropriate layers, using either the functional API or the subclassing API. For example, the following model is equivalent to the previous model, except it is built using the functional API and handles masking manually. It also adds a bit of dropout since the previous model was overfitting slightly:

```
inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)
token_ids = text_vec_layer(inputs)
mask = tf.math.not_equal(token_ids, 0)
Z = tf.keras.layers.Embedding(vocab_size, embed_size)(token_ids)
Z = tf.keras.layers.GRU(128, dropout=0.2)(Z, mask=mask)
outputs = tf.keras.layers.Dense(1, activation="sigmoid")(Z)
model = tf.keras.Model(inputs=[inputs], outputs=[outputs])
```

One last approach to masking is to feed the model with ragged tensors.⁹ In practice, all you need to do is to set `ragged=True` when creating the `TextVectorization` layer, so that the input sequences are represented as ragged tensors:

```
>>> text_vec_layer_ragged = tf.keras.layers.TextVectorization(
...     max_tokens=vocab_size, ragged=True)
...
>>> text_vec_layer_ragged.adapt(train_set.map(lambda reviews, labels: reviews))
>>> text_vec_layer_ragged(["Great movie!", "This is DiCaprio's best role."])
<tf.RaggedTensor [[86, 18], [11, 7, 1, 116, 217]]>
```

Compare this ragged tensor representation with the regular tensor representation, which uses padding tokens:

```
>>> text_vec_layer(["Great movie!", "This is DiCaprio's best role."])
<tf.Tensor: shape=(2, 5), dtype=int64, numpy=
array([[ 86,  18,   0,   0,   0],
       [ 11,    7,   1, 116, 217]])>
```

Keras's recurrent layers have built-in support for ragged tensors, so there's nothing else you need to do: just use this TextVectorization layer in your model. There's no need to pass mask_zero=True or handle masks explicitly—it's all implemented for you. That's convenient! However, as of early 2022, the support for ragged tensors in Keras is still fairly recent, so there are a few rough edges. For example, it is currently not possible to use ragged tensors as targets when running on the GPU (but this may be resolved by the time you read these lines).

Whichever masking approach you prefer, after training this model for a few epochs, it will become quite good at judging whether a review is positive or not. If you use the tf.keras.callbacks.TensorBoard() callback, you can visualize the embeddings in TensorBoard as they are being learned: it is fascinating to see words like "awesome" and "amazing" gradually cluster on one side of the embedding space, while words like "awful" and "terrible" cluster on the other side. Some words are not as positive as you might expect (at least with this model), such as the word "good", presumably because many negative reviews contain the phrase "not good".

Reusing Pretrained Embeddings and Language Models

It's impressive that the model is able to learn useful word embeddings based on just 25,000 movie reviews. Imagine how good the embeddings would be if we had billions of reviews to train on! Unfortunately, we don't, but perhaps we can reuse word embeddings trained on some other (very) large text corpus (e.g., Amazon reviews, available on TensorFlow Datasets), even if it is not composed of movie reviews? After all, the word "amazing" generally has the same meaning whether you use it to talk about movies or anything else.

Moreover, perhaps embeddings would be useful for sentiment analysis even if they were trained on another task: since words like "awesome" and "amazing" have a similar meaning, they will likely cluster in the embedding space even for tasks such as predicting the next word in a sentence. If all positive words and all negative words form clusters, then this will be helpful for sentiment analysis. So, instead of training word embeddings, we could just download and use pretrained embeddings, such as Google's [Word2vec embeddings](#), Stanford's [GloVe embeddings](#), or Facebook's [FastText embeddings](#).

Using pretrained word embeddings was popular for several years, but this approach has its limits. In particular, a word has a single representation, no matter the context. For example, the word "right" is encoded the same way in "left and right" and "right and wrong", even though it means two very different things. To address this limitation, a [2018 paper¹⁰](#) by Matthew Peters introduced *Embeddings from Language Models* (ELMo): these are contextualized word embeddings learned from the internal states of a deep bidirectional language model. Instead of just using pretrained embeddings in your model, you reuse part of a pretrained language model.

At roughly the same time, the [Universal Language Model Fine-Tuning \(ULMFiT\) paper¹¹](#) by Jeremy Howard and Sebastian Ruder demonstrated the effectiveness of unsupervised pretraining for NLP tasks: the authors trained an LSTM language model on a huge text corpus using self-supervised learning (i.e., generating the labels automatically from the data), then they fine-tuned it on various tasks. Their model outperformed the state of the art

on six text classification tasks by a large margin (reducing the error rate by 18–24% in most cases). Moreover, the authors showed a pretrained model fine-tuned on just 100 labeled examples could achieve the same performance as one trained from scratch on 10,000 examples. Before the ULMFiT paper, using pretrained models was only the norm in computer vision; in the context of NLP, pretraining was limited to word embeddings. This paper marked the beginning of a new era in NLP: today, reusing pretrained language models is the norm.

For example, let's build a classifier based on the Universal Sentence Encoder, a model architecture introduced in a [2018 paper¹²](#) by a team of Google researchers. This model is based on the transformer architecture, which we will look at later in this chapter. Conveniently, the model is available on TensorFlow Hub:

```
import os
import tensorflow_hub as hub

os.environ["TFHUB_CACHE_DIR"] = "my_tfhub_cache"
model = tf.keras.Sequential([
    hub.KerasLayer("https://tfhub.dev/google/universal-sentence-encoder/4",
                  trainable=True, dtype=tf.string, input_shape=[]),
    tf.keras.layers.Dense(64, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
model.fit(train_set, validation_data=valid_set, epochs=10)
```

TIP

This model is quite large—close to 1 GB in size—so it may take a while to download. By default, TensorFlow Hub modules are saved to a temporary directory, and they get downloaded again and again every time you run your program. To avoid that, you must set the TFHUB_CACHE_DIR environment variable to a directory of your choice: the modules will then be saved there, and only downloaded once.

Note that the last part of the TensorFlow Hub module URL specifies that we want version 4 of the model. This versioning ensures that if a new module

version is released on TF Hub, it will not break our model. Conveniently, if you just enter this URL in a web browser, you will get the documentation for this module.

Also note that we set trainable=True when creating the hub.KerasLayer. This way, the pretrained Universal Sentence Encoder is fine-tuned during training: some of its weights are tweaked via backprop. Not all TensorFlow Hub modules are fine-tunable, so make sure to check the documentation for each pretrained module you're interested in.

After training, this model should reach a validation accuracy of over 90%. That's actually really good: if you try to perform the task yourself, you will probably do only marginally better since many reviews contain both positive and negative comments. Classifying these ambiguous reviews is like flipping a coin.

So far we have looked at text generation using a char-RNN, and sentiment analysis with word-level RNN models (based on trainable embeddings) and using a powerful pretrained language model from TensorFlow Hub. In the next section, we will explore another important NLP task: *neural machine translation* (NMT).

An Encoder–Decoder Network for Neural Machine Translation

Let's begin with a simple [NMT model¹³](#) that will translate English sentences to Spanish (see [Figure 16-3](#)).

In short, the architecture is as follows: English sentences are fed as inputs to the encoder, and the decoder outputs the Spanish translations. Note that the Spanish translations are also used as inputs to the decoder during training, but shifted back by one step. In other words, during training the decoder is given as input the word that it *should* have output at the previous step, regardless of what it actually output. This is called *teacher forcing*—a technique that significantly speeds up training and improves the model's performance. For the very first word, the decoder is given the start-of-sequence (SOS) token, and the decoder is expected to end the sentence with an end-of-sequence (EOS) token.

Each word is initially represented by its ID (e.g., 854 for the word “soccer”). Next, an Embedding layer returns the word embedding. These word embeddings are then fed to the encoder and the decoder.

At each step, the decoder outputs a score for each word in the output vocabulary (i.e., Spanish), then the softmax activation function turns these scores into probabilities. For example, at the first step the word “Me” may have a probability of 7%, “Yo” may have a probability of 1%, and so on. The word with the highest probability is output. This is very much like a regular classification task, and indeed you can train the model using the “sparse_categorical_crossentropy” loss, much like we did in the char-RNN model.

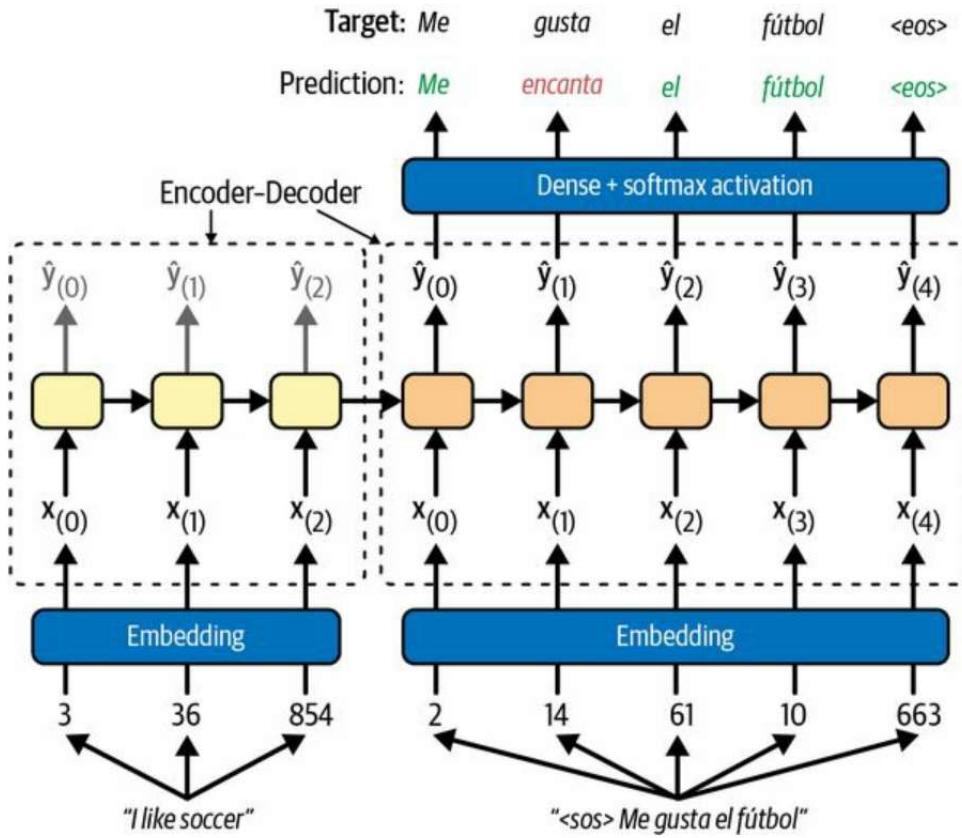


Figure 16-3. A simple machine translation model

Note that at inference time (after training), you will not have the target sentence to feed to the decoder. Instead, you need to feed it the word that it has just output at the previous step, as shown in [Figure 16-4](#) (this will require an embedding lookup that is not shown in the diagram).

TIP

In a [2015 paper](#), ¹⁴ Samy Bengio et al. proposed gradually switching from feeding the decoder the previous *target* token to feeding it the previous *output* token during training.

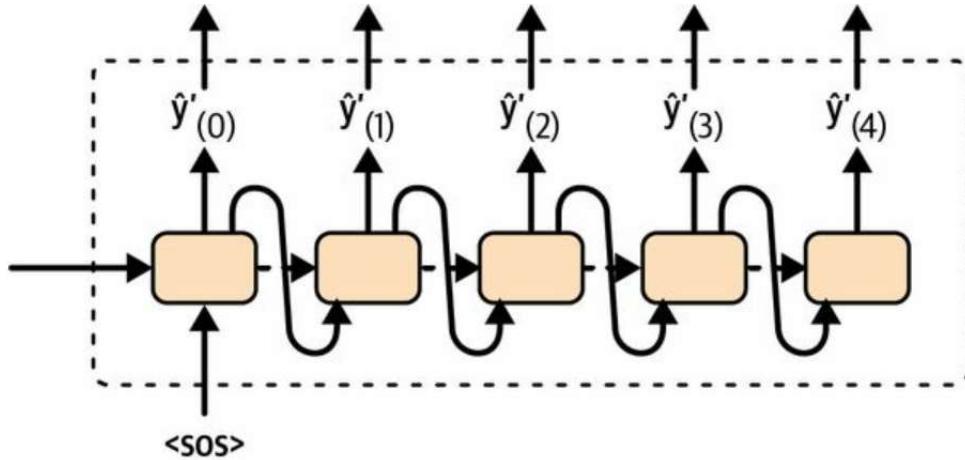


Figure 16-4. At inference time, the decoder is fed as input the word it just output at the previous time step

Let's build and train this model! First, we need to download a dataset of English/Spanish sentence pairs: [15](#)

```
url = "https://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip"
path = tf.keras.utils.get_file("spa-eng.zip", origin=url, cache_dir="datasets",
                               extract=True)
text = (Path(path).with_name("spa-eng") / "spa.txt").read_text()
```

Each line contains an English sentence and the corresponding Spanish translation, separated by a tab. We'll start by removing the Spanish characters “í” and “¿”, which the TextVectorization layer doesn't handle, then we will parse the sentence pairs and shuffle them. Finally, we will split them into two separate lists, one per language:

```
import numpy as np

text = text.replace("í", "").replace("¿", "")
pairs = [line.split("\t") for line in text.splitlines()]
np.random.shuffle(pairs)
sentences_en, sentences_es = zip(*pairs) # separates the pairs into 2 lists
```

Let's take a look at the first three sentence pairs:

```

>>> for i in range(3):
...     print(sentences_en[i], "=>", sentences_es[i])
...
How boring! => Qué aburrimiento!
I love sports. => Adoro el deporte.
Would you like to swap jobs? => Te gustaría que intercambiemos los trabajos?

```

Next, let's create two TextVectorization layers—one per language—and adapt them to the text:

```

vocab_size = 1000
max_length = 50
text_vec_layer_en = tf.keras.layers.TextVectorization(
    vocab_size, output_sequence_length=max_length)
text_vec_layer_es = tf.keras.layers.TextVectorization(
    vocab_size, output_sequence_length=max_length)
text_vec_layer_en.adapt(sentences_en)
text_vec_layer_es.adapt([f"startofseq {s} endofseq" for s in sentences_es])

```

There are a few things to note here:

- We limit the vocabulary size to 1,000, which is quite small. That's because the training set is not very large, and because using a small value will speed up training. State-of-the-art translation models typically use a much larger vocabulary (e.g., 30,000), a much larger training set (gigabytes), and a much larger model (hundreds or even thousands of megabytes). For example, check out the Opus-MT models by the University of Helsinki, or the M2M-100 model by Facebook.
- Since all sentences in the dataset have a maximum of 50 words, we set output_sequence_length to 50: this way the input sequences will automatically be padded with zeros until they are all 50 tokens long. If there was any sentence longer than 50 tokens in the training set, it would be cropped to 50 tokens.
- For the Spanish text, we add “startofseq” and “endofseq” to each sentence when adapting the TextVectorization layer: we will use these words as SOS and EOS tokens. You could use any other words, as long as they are not actual Spanish words.

Let's inspect the first 10 tokens in both vocabularies. They start with the padding token, the unknown token, the SOS and EOS tokens (only in the Spanish vocabulary), then the actual words, sorted by decreasing frequency:

```
>>> text_vec_layer_en.get_vocabulary()[:10]
[", '[UNK]', 'the', 'i', 'to', 'you', 'tom', 'a', 'is', 'he']
>>> text_vec_layer_es.get_vocabulary()[:10]
[", '[UNK]', 'startofseq', 'endofseq', 'de', 'que', 'a', 'no', 'tom', 'la']
```

Next, let's create the training set and the validation set (you could also create a test set if you needed it). We will use the first 100,000 sentence pairs for training, and the rest for validation. The decoder's inputs are the Spanish sentences plus an SOS token prefix. The targets are the Spanish sentences plus an EOS suffix:

```
X_train = tf.constant(sentences_en[:100_000])
X_valid = tf.constant(sentences_en[100_000:])
X_train_dec = tf.constant([f"startofseq {s}" for s in sentences_es[:100_000]])
X_valid_dec = tf.constant([f"startofseq {s}" for s in sentences_es[100_000:]])
Y_train = text_vec_layer_es([f"{s} endofseq" for s in sentences_es[:100_000]])
Y_valid = text_vec_layer_es([f"{s} endofseq" for s in sentences_es[100_000:]])
```

OK, we're now ready to build our translation model. We will use the functional API for that since the model is not sequential. It requires two text inputs—one for the encoder and one for the decoder—so let's start with that:

```
encoder_inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)
decoder_inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)
```

Next, we need to encode these sentences using the TextVectorization layers we prepared earlier, followed by an Embedding layer for each language, with `mask_zero=True` to ensure masking is handled automatically. The embedding size is a hyperparameter you can tune, as always:

```
embed_size = 128
encoder_input_ids = text_vec_layer_en(encoder_inputs)
decoder_input_ids = text_vec_layer_es(decoder_inputs)
encoder_embedding_layer = tf.keras.layers.Embedding(vocab_size, embed_size,
mask_zero=True)
```

```
decoder_embedding_layer = tf.keras.layers.Embedding(vocab_size, embed_size,
                                                    mask_zero=True)
encoder_embeddings = encoder_embedding_layer(encoder_input_ids)
decoder_embeddings = decoder_embedding_layer(decoder_input_ids)
```

TIP

When the languages share many words, you may get better performance using the same embedding layer for both the encoder and the decoder.

Now let's create the encoder and pass it the embedded inputs:

```
encoder = tf.keras.layers.LSTM(512, return_state=True)
encoder_outputs, *encoder_state = encoder(encoder_embeddings)
```

To keep things simple, we just used a single LSTM layer, but you could stack several of them. We also set `return_state=True` to get a reference to the layer's final state. Since we're using an LSTM layer, there are actually two states: the short-term state and the long-term state. The layer returns these states separately, which is why we had to write `*encoder_state` to group both states in a list. ¹⁶ Now we can use this (double) state as the initial state of the decoder:

```
decoder = tf.keras.layers.LSTM(512, return_sequences=True)
decoder_outputs = decoder(decoder_embeddings, initial_state=encoder_state)
```

Next, we can pass the decoder's outputs through a Dense layer with the softmax activation function to get the word probabilities for each step:

```
output_layer = tf.keras.layers.Dense(vocab_size, activation="softmax")
Y_proba = output_layer(decoder_outputs)
```

OPTIMIZING THE OUTPUT LAYER

When the output vocabulary is large, outputting a probability for each and every possible word can be quite slow. If the target vocabulary

contained, say, 50,000 Spanish words instead of 1,000, then the decoder would output 50,000-dimensional vectors, and computing the softmax function over such a large vector would be very computationally intensive. To avoid this, one solution is to look only at the logits output by the model for the correct word and for a random sample of incorrect words, then compute an approximation of the loss based only on these logits. This *sampled softmax* technique was introduced in 2015 by Sébastien Jean et al.¹⁷ In TensorFlow you can use the `tf.nn.sampled_softmax_loss()` function for this during training and use the normal softmax function at inference time (sampled softmax cannot be used at inference time because it requires knowing the target).

Another thing you can do to speed up training—which is compatible with sampled softmax—is to tie the weights of the output layer to the transpose of the decoder’s embedding matrix (you will see how to tie weights in Chapter 17). This significantly reduces the number of model parameters, which speeds up training and may sometimes improve the model’s accuracy as well, especially if you don’t have a lot of training data. The embedding matrix is equivalent to one-hot encoding followed by a linear layer with no bias term and no activation function that maps the one-hot vectors to the embedding space. The output layer does the reverse. So, if the model can find an embedding matrix whose transpose is close to its inverse (such a matrix is called an *orthogonal matrix*), then there’s no need to learn a separate set of weights for the output layer.

And that’s it! We just need to create the Keras Model, compile it, and train it:

```
model = tf.keras.Model(inputs=[encoder_inputs, decoder_inputs],
                       outputs=[Y_prob])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam",
              metrics=["accuracy"])
model.fit((X_train, X_train_dec), Y_train, epochs=10,
          validation_data=((X_valid, X_valid_dec), Y_valid))
```

After training, we can use the model to translate new English sentences to Spanish. But it’s not as simple as calling `model.predict()`, because the

decoder expects as input the word that was predicted at the previous time step. One way to do this is to write a custom memory cell that keeps track of the previous output and feeds it to the encoder at the next time step. However, to keep things simple, we can just call the model multiple times, predicting one extra word at each round. Let's write a little utility function for that:

```
def translate(sentence_en):
    translation = ""
    for word_idx in range(max_length):
        X = np.array([sentence_en]) # encoder input
        X_dec = np.array(["startofseq" + translation]) # decoder input
        y_proba = model.predict((X, X_dec))[0, word_idx] # last token's probas
        predicted_word_id = np.argmax(y_proba)
        predicted_word = text_vec_layer_es.get_vocabulary()[predicted_word_id]
        if predicted_word == "endofseq":
            break
        translation += " " + predicted_word
    return translation.strip()
```

The function simply keeps predicting one word at a time, gradually completing the translation, and it stops once it reaches the EOS token. Let's give it a try!

```
>>> translate("I like soccer")
'me gusta el fútbol'
```

Hurray, it works! Well, at least it does with very short sentences. If you try playing with this model for a while, you will find that it's not bilingual yet, and in particular it really struggles with longer sentences. For example:

```
>>> translate("I like soccer and also going to the beach")
'me gusta el fútbol y a veces mismo al bus'
```

The translation says "I like soccer and sometimes even the bus". So how can you improve it? One way is to increase the training set size and add more LSTM layers in both the encoder and the decoder. But this will only get you so far, so let's look at more sophisticated techniques, starting with bidirectional recurrent layers.

Bidirectional RNNs

At each time step, a regular recurrent layer only looks at past and present inputs before generating its output. In other words, it is *causal*, meaning it cannot look into the future. This type of RNN makes sense when forecasting time series, or in the decoder of a sequence-to-sequence (seq2seq) model. But for tasks like text classification, or in the encoder of a seq2seq model, it is often preferable to look ahead at the next words before encoding a given word.

For example, consider the phrases “the right arm”, “the right person”, and “the right to criticize”: to properly encode the word “right”, you need to look ahead. One solution is to run two recurrent layers on the same inputs, one reading the words from left to right and the other reading them from right to left, then combine their outputs at each time step, typically by concatenating them. This is what a *bidirectional recurrent layer* does (see [Figure 16-5](#)).

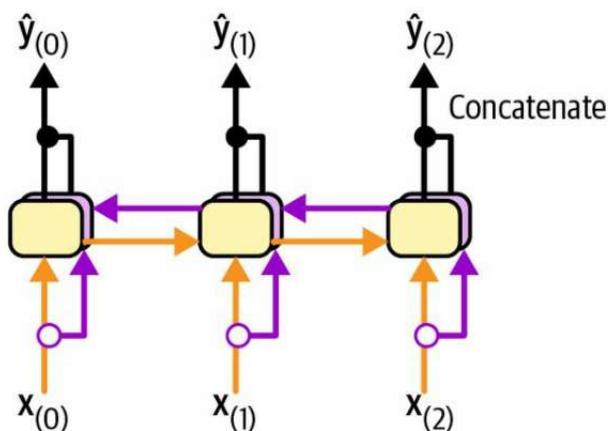


Figure 16-5. A bidirectional recurrent layer

To implement a bidirectional recurrent layer in Keras, just wrap a recurrent layer in a `tf.keras.layers.Bidirectional`. For example, the following Bidirectional layer could be used as the encoder in our translation model:

```
encoder = tf.keras.layers.Bidirectional(  
    tf.keras.layers.LSTM(256, return_state=True))
```

NOTE

The Bidirectional layer will create a clone of the GRU layer (but in the reverse direction), and it will run both and concatenate their outputs. So although the GRU layer has 10 units, the Bidirectional layer will output 20 values per time step.

There's just one problem. This layer will now return four states instead of two: the final short-term and long-term states of the forward LSTM layer, and the final short-term and long-term states of the backward LSTM layer. We cannot use this quadruple state directly as the initial state of the decoder's LSTM layer, since it expects just two states (short-term and long-term). We cannot make the decoder bidirectional, since it must remain causal: otherwise it would cheat during training and it would not work. Instead, we can concatenate the two short-term states, and also concatenate the two long-term states:

```
encoder_outputs, *encoder_state = encoder(encoder_embeddings)
encoder_state = [tf.concat(encoder_state[::2], axis=-1), # short-term (0 & 2)
                 tf.concat(encoder_state[1::2], axis=-1)] # long-term (1 & 3)
```

Now let's look at another popular technique that can greatly improve the performance of a translation model at inference time: beam search.

Beam Search

Suppose you have trained an encoder–decoder model, and you use it to translate the sentence “I like soccer” to Spanish. You are hoping that it will output the proper translation “me gusta el fútbol”, but unfortunately it outputs “me gustan los jugadores”, which means “I like the players”. Looking at the training set, you notice many sentences such as “I like cars”, which translates to “me gustan los autos”, so it wasn’t absurd for the model to output “me gustan los” after seeing “I like”. Unfortunately, in this case it was a mistake since “soccer” is singular. The model could not go back and fix it, so it tried to complete the sentence as best it could, in this case using the word “jugadores”. How can we give the model a chance to go back and fix mistakes it made earlier? One of the most common solutions is *beam search*: it keeps track of a short list of the k most promising sentences (say, the top three), and at each decoder step it tries to extend them by one word, keeping only the k most likely sentences. The parameter k is called the *beam width*.

For example, suppose you use the model to translate the sentence “I like soccer” using beam search with a beam width of 3 (see [Figure 16-6](#)). At the first decoder step, the model will output an estimated probability for each possible first word in the translated sentence. Suppose the top three words are “me” (75% estimated probability), “a” (3%), and “como” (1%). That’s our short list so far. Next, we use the model to find the next word for each sentence. For the first sentence (“me”), perhaps the model outputs a probability of 36% for the word “gustan”, 32% for the word “gusta”, 16% for the word “encanta”, and so on. Note that these are actually *conditional* probabilities, given that the sentence starts with “me”. For the second sentence (“a”), the model might output a conditional probability of 50% for the word “mi”, and so on. Assuming the vocabulary has 1,000 words, we will end up with 1,000 probabilities per sentence.

Next, we compute the probabilities of each of the 3,000 two-word sentences we considered ($3 \times 1,000$). We do this by multiplying the estimated conditional probability of each word by the estimated probability of the sentence it completes. For example, the estimated probability of the sentence

“me” was 75%, while the estimated conditional probability of the word “gustan” (given that the first word is “me”) was 36%, so the estimated probability of the sentence “me gustan” is $75\% \times 36\% = 27\%$. After computing the probabilities of all 3,000 two-word sentences, we keep only the top 3. In this example they all start with the word “me”: “me gustan” (27%), “me gusta” (24%), and “me encanta” (12%). Right now, the sentence “me gustan” is winning, but “me gusta” has not been eliminated.



Figure 16-6. Beam search, with a beam width of 3

Then we repeat the same process: we use the model to predict the next word in each of these three sentences, and we compute the probabilities of all 3,000 three-word sentences we considered. Perhaps the top three are now “me gustan los” (10%), “me gusta el” (8%), and “me gusta mucho” (2%). At the next step we may get “me gusta el fútbol” (6%), “me gusta mucho el” (1%), and “me gusta el deporte” (0.2%). Notice that “me gustan” was eliminated, and the correct translation is now ahead. We boosted our encoder–decoder model’s performance without any extra training, simply by using it more wisely.

TIP

The TensorFlow Addons library includes a full seq2seq API that lets you build encoder–decoder models with attention, including beam search, and more. However, its documentation is currently very limited. Implementing beam search is a good exercise, so give it a try! Check out this chapter’s notebook for a possible solution.

With all this, you can get reasonably good translations for fairly short sentences. Unfortunately, this model will be really bad at translating long sentences. Once again, the problem comes from the limited short-term memory of RNNs. *Attention mechanisms* are the game-changing innovation that addressed this problem.

Attention Mechanisms

Consider the path from the word “soccer” to its translation “fútbol” back in [Figure 16-3](#): it is quite long! This means that a representation of this word (along with all the other words) needs to be carried over many steps before it is actually used. Can’t we make this path shorter?

This was the core idea in a landmark [2014 paper¹⁸](#) by Dzmitry Bahdanau et al., where the authors introduced a technique that allowed the decoder to focus on the appropriate words (as encoded by the encoder) at each time step. For example, at the time step where the decoder needs to output the word “fútbol”, it will focus its attention on the word “soccer”. This means that the path from an input word to its translation is now much shorter, so the short-term memory limitations of RNNs have much less impact. Attention mechanisms revolutionized neural machine translation (and deep learning in general), allowing a significant improvement in the state of the art, especially for long sentences (e.g., over 30 words).

NOTE

The most common metric used in NMT is the *bilingual evaluation understudy* (BLEU) score, which compares each translation produced by the model with several good translations produced by humans: it counts the number of n -grams (sequences of n words) that appear in any of the target translations and adjusts the score to take into account the frequency of the produced n -grams in the target translations.

[Figure 16-7](#) shows our encoder–decoder model with an added attention mechanism. On the left, you have the encoder and the decoder. Instead of just sending the encoder’s final hidden state to the decoder, as well as the previous target word at each step (which is still done, although it is not shown in the figure), we now send all of the encoder’s outputs to the decoder as well. Since the decoder cannot deal with all these encoder outputs at once, they need to be aggregated: at each time step, the decoder’s memory cell computes a weighted sum of all the encoder outputs. This determines which

words it will focus on at this step. The weight $\alpha_{(t,i)}$ is the weight of the i^{th} encoder output at the t^{th} decoder time step. For example, if the weight $\alpha_{(3,2)}$ is much larger than the weights $\alpha_{(3,0)}$ and $\alpha_{(3,1)}$, then the decoder will pay much more attention to the encoder's output for word #2 ("soccer") than to the other two outputs, at least at this time step. The rest of the decoder works just like earlier: at each time step the memory cell receives the inputs we just discussed, plus the hidden state from the previous time step, and finally (although it is not represented in the diagram) it receives the target word from the previous time step (or at inference time, the output from the previous time step).

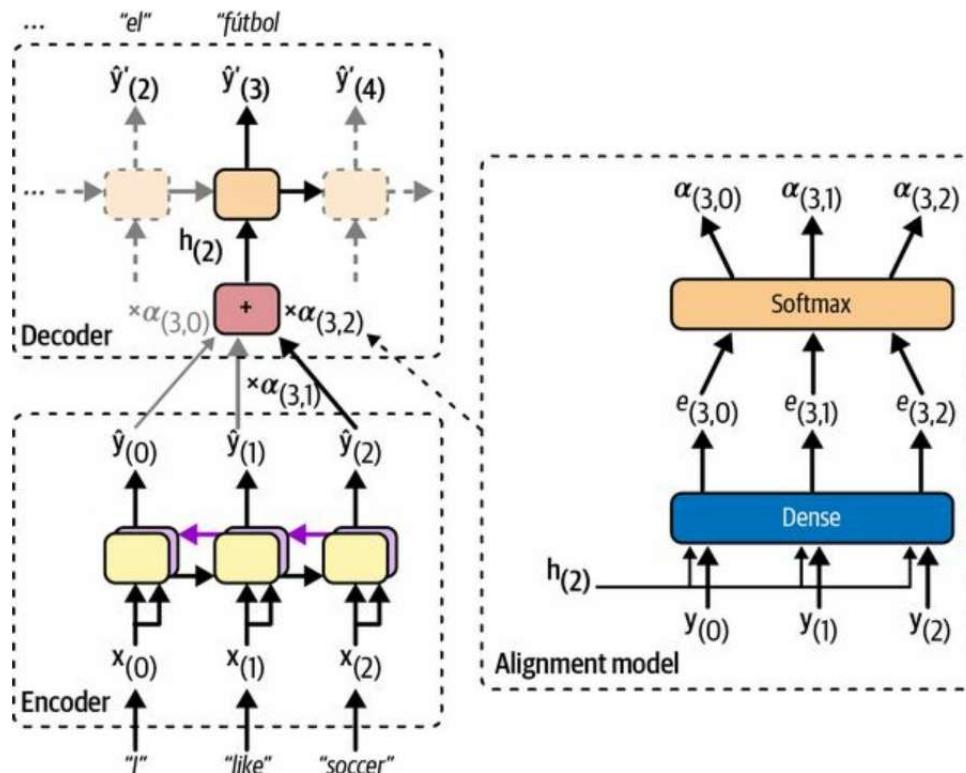


Figure 16-7. Neural machine translation using an encoder-decoder network with an attention model

But where do these $\alpha_{(t,i)}$ weights come from? Well, they are generated by a small neural network called an *alignment model* (or an *attention layer*),

which is trained jointly with the rest of the encoder–decoder model. This alignment model is illustrated on the righthand side of [Figure 16-7](#). It starts with a Dense layer composed of a single neuron that processes each of the encoder’s outputs, along with the decoder’s previous hidden state (e.g., $\mathbf{h}_{(2)}$). This layer outputs a score (or energy) for each encoder output (e.g., $e_{(3,2)}$): this score measures how well each output is aligned with the decoder’s previous hidden state. For example, in [Figure 16-7](#), the model has already output “me gusta el” (meaning “I like”), so it’s now expecting a noun: the word “soccer” is the one that best aligns with the current state, so it gets a high score. Finally, all the scores go through a softmax layer to get a final weight for each encoder output (e.g., $\alpha_{(3,2)}$). All the weights for a given decoder time step add up to 1. This particular attention mechanism is called *Bahdanau attention* (named after the 2014 paper’s first author). Since it concatenates the encoder output with the decoder’s previous hidden state, it is sometimes called *concatenative attention* (or *additive attention*).

NOTE

If the input sentence is n words long, and assuming the output sentence is about as long, then this model will need to compute about n^2 weights. Fortunately, this quadratic computational complexity is still tractable because even long sentences don’t have thousands of words.

Another common attention mechanism, known as *Luong attention* or *multiplicative attention*, was proposed shortly after, in [2015](#),¹⁹ by Minh-Thang Luong et al. Because the goal of the alignment model is to measure the similarity between one of the encoder’s outputs and the decoder’s previous hidden state, the authors proposed to simply compute the dot product (see [Chapter 4](#)) of these two vectors, as this is often a fairly good similarity measure, and modern hardware can compute it very efficiently. For this to be possible, both vectors must have the same dimensionality. The dot product gives a score, and all the scores (at a given decoder time step) go through a softmax layer to give the final weights, just like in Bahdanau attention. Another simplification Luong et al. proposed was to use the decoder’s hidden

state at the current time step rather than at the previous time step (i.e., $\mathbf{h}_{(t)}$ rather than $\mathbf{h}_{(t-1)}$), then to use the output of the attention mechanism (noted $\mathbf{h}^*(t)$) directly to compute the decoder's predictions, rather than using it to compute the decoder's current hidden state. The researchers also proposed a variant of the dot product mechanism where the encoder outputs first go through a fully connected layer (without a bias term) before the dot products are computed. This is called the “general” dot product approach. The researchers compared both dot product approaches with the concatenative attention mechanism (adding a rescaling parameter vector \mathbf{v}), and they observed that the dot product variants performed better than concatenative attention. For this reason, concatenative attention is much less used now. The equations for these three attention mechanisms are summarized in [Equation 16-1](#).

Equation 16-1. Attention mechanisms

$\mathbf{h}^*(t) = \sum_i i \alpha(t, i) \mathbf{y}(i)$ with $\alpha(t, i) = \text{exp}(e(t, i) / \sum_i e(t, i))$ and $e(t, i) = \mathbf{h}(t)^\top \mathbf{y}(i) + \mathbf{d}(t)^\top \mathbf{y}(i)$
 $\mathbf{W}\mathbf{y}(i) + \mathbf{b}$ general $\mathbf{v}^\top \tanh(\mathbf{W}[\mathbf{h}(t); \mathbf{y}(i)])$ concat

Keras provides a `tf.keras.layers.Attention` layer for Luong attention, and an `AdditiveAttention` layer for Bahdanau attention. Let's add Luong attention to our encoder–decoder model. Since we will need to pass all the encoder's outputs to the `Attention` layer, we first need to set `return_sequences=True` when creating the encoder:

```
encoder = tf.keras.layers.Bidirectional(
    tf.keras.layers.LSTM(256, return_sequences=True, return_state=True))
```

Next, we need to create the attention layer and pass it the decoder's states and the encoder's outputs. However, to access the decoder's states at each step we would need to write a custom memory cell. For simplicity, let's use the decoder's outputs instead of its states: in practice this works well too, and it's much easier to code. Then we just pass the attention layer's outputs directly to the output layer, as suggested in the Luong attention paper:

```
attention_layer = tf.keras.layers.Attention()
attention_outputs = attention_layer([decoder_outputs, encoder_outputs])
```

```
output_layer = tf.keras.layers.Dense(vocab_size, activation="softmax")
Y_proba = output_layer(attention_outputs)
```

And that's it! If you train this model, you will find that it now handles much longer sentences. For example:

```
>>> translate("I like soccer and also going to the beach")
'me gusta el fútbol y también ir a la playa'
```

In short, the attention layer provides a way to focus the attention of the model on part of the inputs. But there's another way to think of this layer: it acts as a differentiable memory retrieval mechanism.

For example, let's suppose the encoder analyzed the input sentence "I like soccer", and it managed to understand that the word "I" is the subject and the word "like" is the verb, so it encoded this information in its outputs for these words. Now suppose the decoder has already translated the subject, and it thinks that it should translate the verb next. For this, it needs to fetch the verb from the input sentence. This is analogous to a dictionary lookup: it's as if the encoder had created a dictionary {"subject": "They", "verb": "played", ...} and the decoder wanted to look up the value that corresponds to the key "verb".

However, the model does not have discrete tokens to represent the keys (like "subject" or "verb"); instead, it has vectorized representations of these concepts that it learned during training, so the query it will use for the lookup will not perfectly match any key in the dictionary. The solution is to compute a similarity measure between the query and each key in the dictionary, and then use the softmax function to convert these similarity scores to weights that add up to 1. As we saw earlier, that's exactly what the attention layer does. If the key that represents the verb is by far the most similar to the query, then that key's weight will be close to 1.

Next, the attention layer computes a weighted sum of the corresponding values: if the weight of the "verb" key is close to 1, then the weighted sum will be very close to the representation of the word "played".

This is why the Keras Attention and AdditiveAttention layers both expect a

list as input, containing two or three items: the *queries*, the *keys*, and optionally the *values*. If you do not pass any values, then they are automatically equal to the keys. So, looking at the previous code example again, the decoder outputs are the queries, and the encoder outputs are both the keys and the values. For each decoder output (i.e., each query), the attention layer returns a weighted sum of the encoder outputs (i.e., the keys/values) that are most similar to the decoder output.

The bottom line is that an attention mechanism is a trainable memory retrieval system. It is so powerful that you can actually build state-of-the-art models using only attention mechanisms. Enter the transformer architecture.

Attention Is All You Need: The Original Transformer Architecture

In a groundbreaking [2017 paper](#), ²⁰ a team of Google researchers suggested that “Attention Is All You Need”. They created an architecture called the *transformer*, which significantly improved the state-of-the-art in NMT without using any recurrent or convolutional layers, ²¹ just attention mechanisms (plus embedding layers, dense layers, normalization layers, and a few other bits and pieces). Because the model is not recurrent, it doesn’t suffer as much from the vanishing or exploding gradients problems as RNNs, it can be trained in fewer steps, it’s easier to parallelize across multiple GPUs, and it can better capture long-range patterns than RNNs. The original 2017 transformer architecture is represented in [Figure 16-8](#).

In short, the left part of [Figure 16-8](#) is the encoder, and the right part is the decoder. Each embedding layer outputs a 3D tensor of shape [*batch size*, *sequence length*, *embedding size*]. After that, the tensors are gradually transformed as they flow through the transformer, but their shape remains the same.

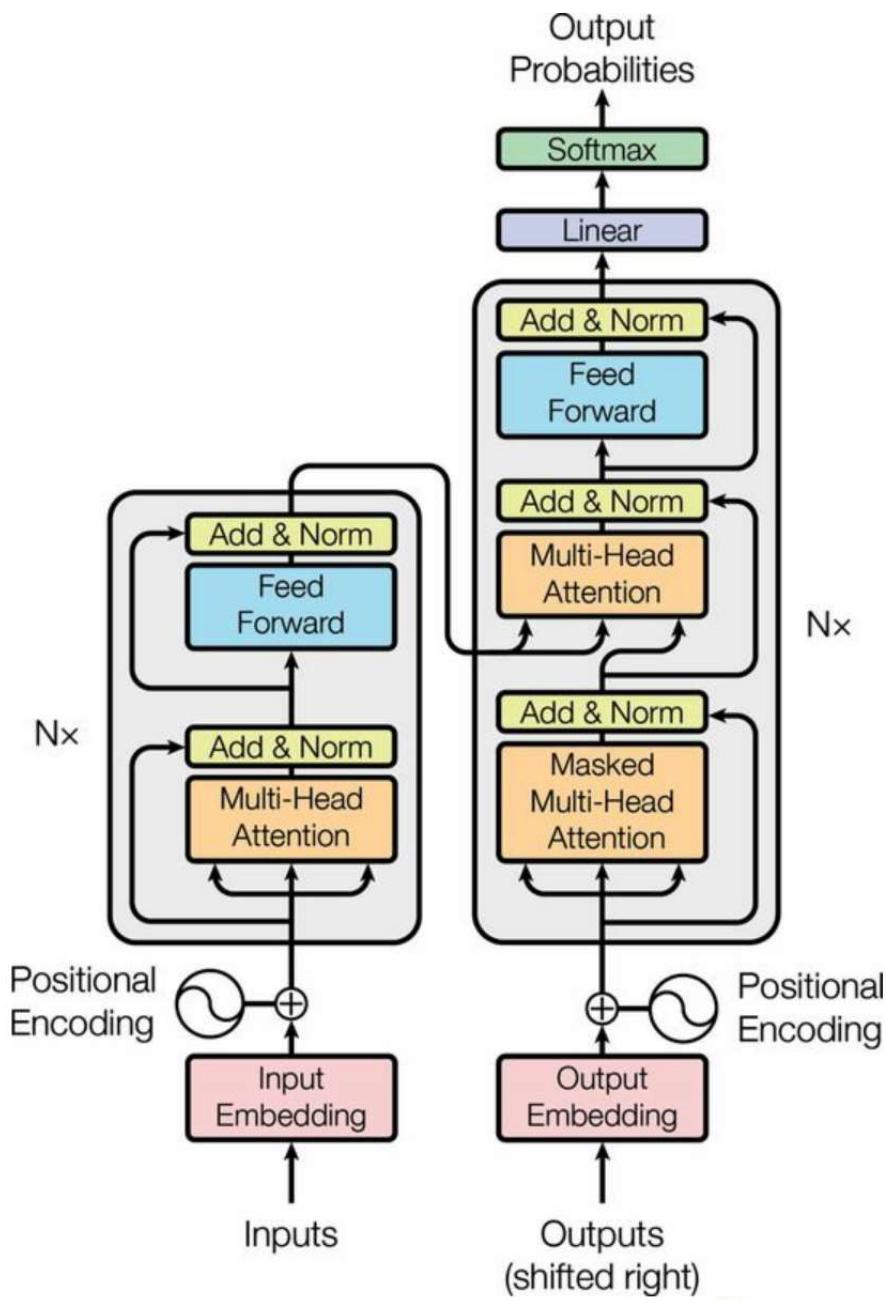


Figure 16-8. The original 2017 transformer architecture ²²

If you use the transformer for NMT, then during training you must feed the

English sentences to the encoder and the corresponding Spanish translations to the decoder, with an extra SOS token inserted at the start of each sentence. At inference time, you must call the transformer multiple times, producing the translations one word at a time and feeding the partial translations to the decoder at each round, just like we did earlier in the `translate()` function.

The encoder’s role is to gradually transform the inputs—word representations of the English sentence—until each word’s representation perfectly captures the meaning of the word, in the context of the sentence. For example, if you feed the encoder with the sentence “I like soccer”, then the word “like” will start off with a rather vague representation, since this word could mean different things in different contexts: think of “I like soccer” versus “It’s like that”. But after going through the encoder, the word’s representation should capture the correct meaning of “like” in the given sentence (i.e., to be fond of), as well as any other information that may be required for translation (e.g., it’s a verb).

The decoder’s role is to gradually transform each word representation in the translated sentence into a word representation of the next word in the translation. For example, if the sentence to translate is “I like soccer”, and the decoder’s input sentence is “<SOS> me gusta el fútbol”, then after going through the decoder, the word representation of the word “el” will end up transformed into a representation of the word “fútbol”. Similarly, the representation of the word “fútbol” will be transformed into a representation of the EOS token.

After going through the decoder, each word representation goes through a final Dense layer with a softmax activation function, which will hopefully output a high probability for the correct next word and a low probability for all other words. The predicted sentence should be “me gusta el fútbol <EOS>”.

That was the big picture; now let’s walk through [Figure 16-8](#) in more detail:

- First, notice that both the encoder and the decoder contain modules that are stacked N times. In the paper, $N = 6$. The final outputs of the whole encoder stack are fed to the decoder at each of these N levels.

- Zooming in, you can see that you are already familiar with most components: there are two embedding layers; several skip connections, each of them followed by a layer normalization layer; several feedforward modules that are composed of two dense layers each (the first one using the ReLU activation function, the second with no activation function); and finally the output layer is a dense layer using the softmax activation function. You can also sprinkle a bit of dropout after the attention layers and the feedforward modules, if needed. Since all of these layers are time-distributed, each word is treated independently from all the others. But how can we translate a sentence by looking at the words completely separately? Well, we can't, so that's where the new components come in:

- The encoder's *multi-head attention* layer updates each word representation by attending to (i.e., paying attention to) all other words in the same sentence. That's where the vague representation of the word "like" becomes a richer and more accurate representation, capturing its precise meaning in the given sentence. We will discuss exactly how this works shortly.
- The decoder's *masked multi-head attention* layer does the same thing, but when it processes a word, it doesn't attend to words located after it: it's a causal layer. For example, when it processes the word "gusta", it only attends to the words "<SOS> me gusta", and it ignores the words "el fútbol" (or else that would be cheating).
- The decoder's upper *multi-head attention* layer is where the decoder pays attention to the words in the English sentence. This is called *cross-attention*, not *self-attention* in this case. For example, the decoder will probably pay close attention to the word "soccer" when it processes the word "el" and transforms its representation into a representation of the word "fútbol".
- The *positional encodings* are dense vectors (much like word embeddings) that represent the position of each word in the

sentence. The n^{th} positional encoding is added to the word embedding of the n^{th} word in each sentence. This is needed because all layers in the transformer architecture ignore word positions: without positional encodings, you could shuffle the input sequences, and it would just shuffle the output sequences in the same way. Obviously, the order of words matters, which is why we need to give positional information to the transformer somehow: adding positional encodings to the word representations is a good way to achieve this.

NOTE

The first two arrows going into each multi-head attention layer in [Figure 16-8](#) represent the keys and values, and the third arrow represents the queries. In the self-attention layers, all three are equal to the word representations output by the previous layer, while in the decoder's upper attention layer, the keys and values are equal to the encoder's final word representations, and the queries are equal to the word representations output by the previous layer.

Let's go through the novel components of the transformer architecture in more detail, starting with the positional encodings.

Positional encodings

A positional encoding is a dense vector that encodes the position of a word within a sentence: the i^{th} positional encoding is added to the word embedding of the i^{th} word in the sentence. The easiest way to implement this is to use an Embedding layer and make it encode all the positions from 0 to the maximum sequence length in the batch, then add the result to the word embeddings. The rules of broadcasting will ensure that the positional encodings get applied to every input sequence. For example, here is how to add positional encodings to the encoder and decoder inputs:

```
max_length = 50 # max length in the whole training set
embed_size = 128
pos_embed_layer = tf.keras.layers.Embedding(max_length, embed_size)
batch_max_len_enc = tf.shape(encoder_embeddings)[1]
```

```

encoder_in = encoder_embeddings + pos_embed_layer(tf.range(batch_max_len_enc))
batch_max_len_dec = tf.shape(decoder_embeddings)[1]
decoder_in = decoder_embeddings + pos_embed_layer(tf.range(batch_max_len_dec))

```

Note that this implementation assumes that the embeddings are represented as regular tensors, not ragged tensors.²³ The encoder and the decoder share the same Embedding layer for the positional encodings, since they have the same embedding size (this is often the case).

Instead of using trainable positional encodings, the authors of the transformer paper chose to use fixed positional encodings, based on the sine and cosine functions at different frequencies. The positional encoding matrix \mathbf{P} is defined in [Equation 16-2](#) and represented at the top of [Figure 16-9](#) (transposed), where $P_{p,i}$ is the i^{th} component of the encoding for the word located at the p^{th} position in the sentence.

Equation 16-2. Sine/cosine positional encodings

$$P_{p,i} = \sin(p/10000i/d) \text{ if } i \text{ is even} \\ \cos(p/10000(i-1)/d) \text{ if } i \text{ is odd}$$

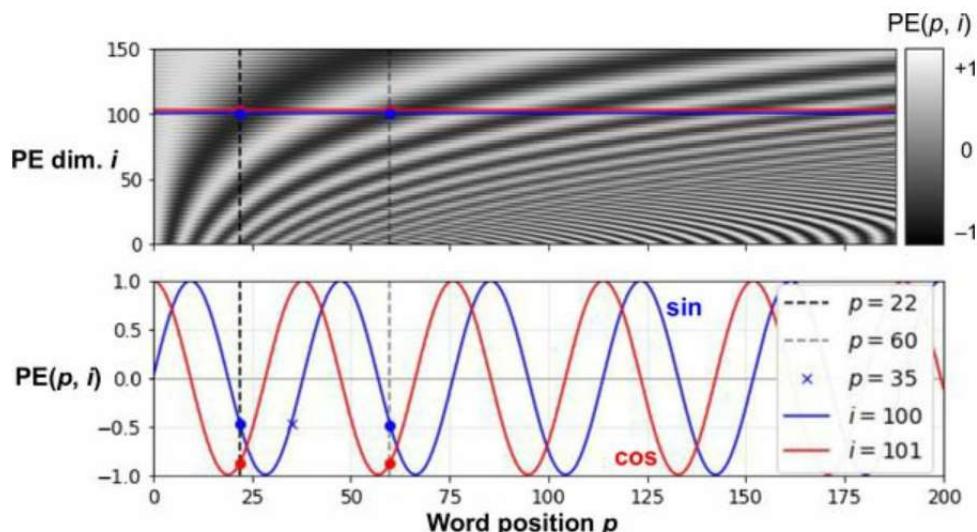


Figure 16-9. Sine/cosine positional encoding matrix (transposed, top) with a focus on two values of i (bottom)

This solution can give the same performance as trainable positional encodings, and it can extend to arbitrarily long sentences without adding any

parameters to the model (however, when there is a large amount of pretraining data, trainable positional encodings are usually favored). After these positional encodings are added to the word embeddings, the rest of the model has access to the absolute position of each word in the sentence because there is a unique positional encoding for each position (e.g., the positional encoding for the word located at the 22nd position in a sentence is represented by the vertical dashed line at the top left of [Figure 16-9](#), and you can see that it is unique to that position). Moreover, the choice of oscillating functions (sine and cosine) makes it possible for the model to learn relative positions as well. For example, words located 38 words apart (e.g., at positions $p = 22$ and $p = 60$) always have the same positional encoding values in the encoding dimensions $i = 100$ and $i = 101$, as you can see in [Figure 16-9](#). This explains why we need both the sine and the cosine for each frequency: if we only used the sine (the blue wave at $i = 100$), the model would not be able to distinguish positions $p = 22$ and $p = 35$ (marked by a cross).

There is no PositionalEncoding layer in TensorFlow, but it is not too hard to create one. For efficiency reasons, we precompute the positional encoding matrix in the constructor. The call() method just truncates this encoding matrix to the max length of the input sequences, and it adds them to the inputs. We also set supports_masking=True to propagate the input's automatic mask to the next layer:

```
class PositionalEncoding(tf.keras.layers.Layer):
    def __init__(self, max_length, embed_size, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)
        assert embed_size % 2 == 0, "embed_size must be even"
        p, i = np.meshgrid(np.arange(max_length),
                           2 * np.arange(embed_size // 2))
        pos_emb = np.empty((1, max_length, embed_size))
        pos_emb[0, :, ::2] = np.sin(p / 10_000 ** (i / embed_size)).T
        pos_emb[0, :, 1::2] = np.cos(p / 10_000 ** (i / embed_size)).T
        self.pos_encodings = tf.constant(pos_emb.astype(self.dtype))
        self.supports_masking = True

    def call(self, inputs):
        batch_max_length = tf.shape(inputs)[1]
        return inputs + self.pos_encodings[:, :batch_max_length]
```

Let's use this layer to add the positional encoding to the encoder's inputs:

```
pos_embed_layer = PositionalEncoding(max_length, embed_size)
encoder_in = pos_embed_layer(encoder_embeddings)
decoder_in = pos_embed_layer(decoder_embeddings)
```

Now let's look deeper into the heart of the transformer model, at the multi-head attention layer.

Multi-head attention

To understand how a multi-head attention layer works, we must first understand the *scaled dot-product attention* layer, which it is based on. Its equation is shown in [Equation 16-3](#), in a vectorized form. It's the same as Luong attention, except for a scaling factor.

Equation 16-3. Scaled dot-product attention

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^\top d_{\text{keys}})V$$

In this equation:

- **Q** is a matrix containing one row per *query*. Its shape is $[n_{\text{queries}}, d_{\text{keys}}]$, where n_{queries} is the number of queries and d_{keys} is the number of dimensions of each query and each key.
- **K** is a matrix containing one row per *key*. Its shape is $[n_{\text{keys}}, d_{\text{keys}}]$, where n_{keys} is the number of keys and values.
- **V** is a matrix containing one row per *value*. Its shape is $[n_{\text{keys}}, d_{\text{values}}]$, where d_{values} is the number of dimensions of each value.
- The shape of QK^\top is $[n_{\text{queries}}, n_{\text{keys}}]$: it contains one similarity score for each query/key pair. To prevent this matrix from being huge, the input sequences must not be too long (we will discuss how to overcome this limitation later in this chapter). The output of the softmax function has the same shape, but all rows sum up to 1. The final output has a shape of $[n_{\text{queries}}, d_{\text{values}}]$: there is one row per query, where each row represents the query result (a weighted sum of the values).

- The scaling factor $1 / (\text{d keys})$ scales down the similarity scores to avoid saturating the softmax function, which would lead to tiny gradients.
- It is possible to mask out some key/value pairs by adding a very large negative value to the corresponding similarity scores, just before computing the softmax. This is useful in the masked multi-head attention layer.

If you set `use_scale=True` when creating a `tf.keras.layers.Attention` layer, then it will create an additional parameter that lets the layer learn how to properly downscale the similarity scores. The scaled dot-product attention used in the transformer model is almost the same, except it always scales the similarity scores by the same factor, $1 / (\text{d keys})$.

Note that the Attention layer's inputs are just like **Q**, **K**, and **V**, except with an extra batch dimension (the first dimension). Internally, the layer computes all the attention scores for all sentences in the batch with just one call to `tf.matmul(queries, keys)`: this makes it extremely efficient. Indeed, in TensorFlow, if A and B are tensors with more than two dimensions—say, of shape [2, 3, 4, 5] and [2, 3, 5, 6], respectively—then `tf.matmul(A, B)` will treat these tensors as 2×3 arrays where each cell contains a matrix, and it will multiply the corresponding matrices: the matrix at the i^{th} row and j^{th} column in A will be multiplied by the matrix at the i^{th} row and j^{th} column in B. Since the product of a 4×5 matrix with a 5×6 matrix is a 4×6 matrix, `tf.matmul(A, B)` will return an array of shape [2, 3, 4, 6].

Now we're ready to look at the multi-head attention layer. Its architecture is shown in [Figure 16-10](#).

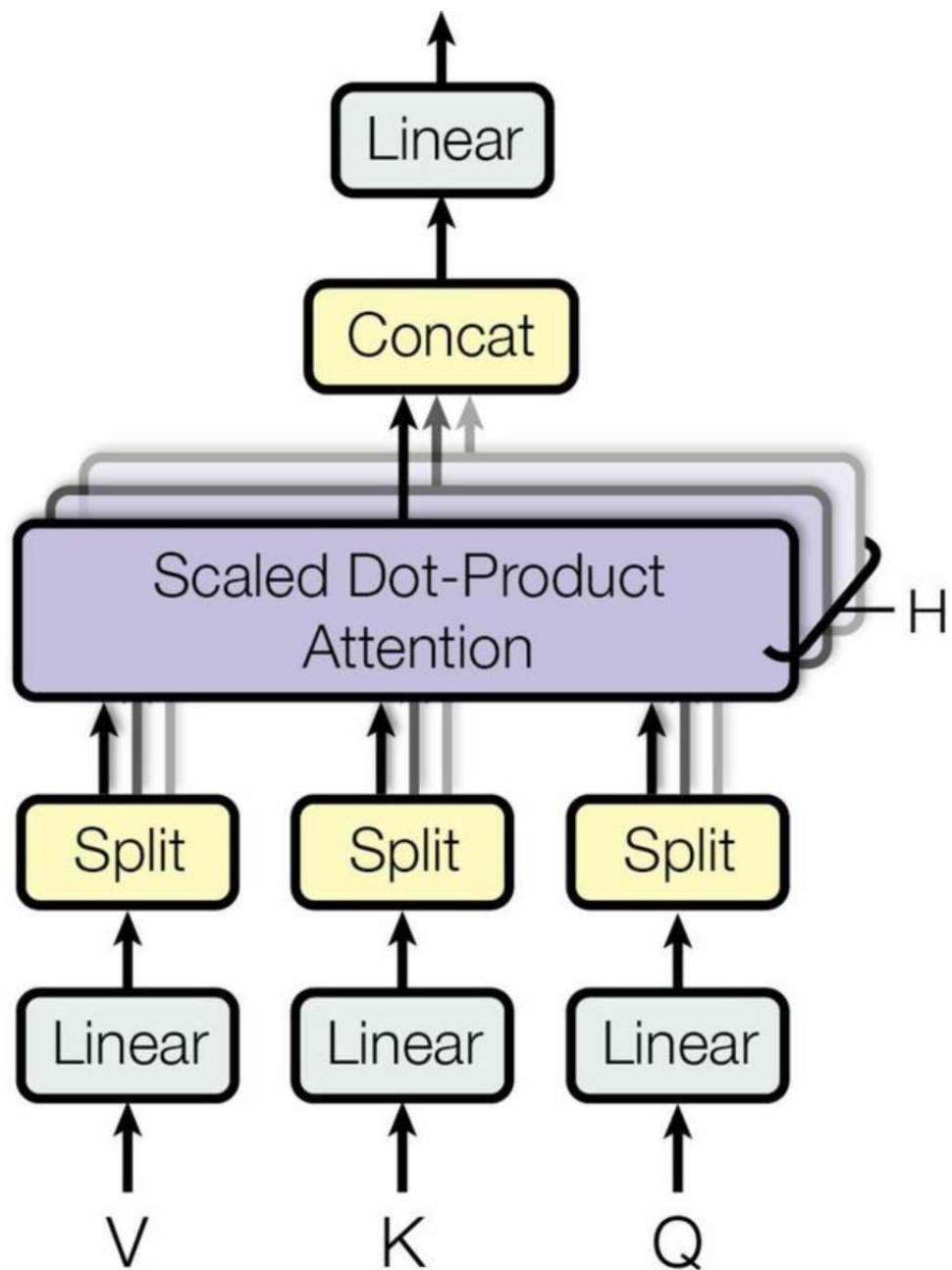


Figure 16-10. Multi-head attention layer architecture ²⁴

As you can see, it is just a bunch of scaled dot-product attention layers, each

preceded by a linear transformation of the values, keys, and queries (i.e., a time-distributed dense layer with no activation function). All the outputs are simply concatenated, and they go through a final linear transformation (again, time-distributed).

But why? What is the intuition behind this architecture? Well, consider once again the word “like” in the sentence “I like soccer”. The encoder was smart enough to encode the fact that it is a verb. But the word representation also includes its position in the text, thanks to the positional encodings, and it probably includes many other features that are useful for its translation, such as the fact that it is in the present tense. In short, the word representation encodes many different characteristics of the word. If we just used a single scaled dot-product attention layer, we would only be able to query all of these characteristics in one shot.

This is why the multi-head attention layer applies *multiple* different linear transformations of the values, keys, and queries: this allows the model to apply many different projections of the word representation into different subspaces, each focusing on a subset of the word’s characteristics. Perhaps one of the linear layers will project the word representation into a subspace where all that remains is the information that the word is a verb, another linear layer will extract just the fact that it is present tense, and so on. Then the scaled dot-product attention layers implement the lookup phase, and finally we concatenate all the results and project them back to the original space.

Keras includes a `tf.keras.layers.MultiHeadAttention` layer, so we now have everything we need to build the rest of the transformer. Let’s start with the full encoder, which is exactly like in [Figure 16-8](#), except we use a stack of two blocks ($N = 2$) instead of six, since we don’t have a huge training set, and we add a bit of dropout as well:

```
N = 2 # instead of 6
num_heads = 8
dropout_rate = 0.1
n_units = 128 # for the first dense layer in each feedforward block
encoder_pad_mask = tf.math.not_equal(encoder_input_ids, 0)[:, tf.newaxis]
Z = encoder_in
```

```

for _ in range(N):
    skip = Z
    attn_layer = tf.keras.layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=embed_size, dropout=dropout_rate)
    Z = attn_layer(Z, value=Z, attention_mask=encoder_pad_mask)
    Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip]))
    skip = Z
    Z = tf.keras.layers.Dense(n_units, activation="relu")(Z)
    Z = tf.keras.layers.Dense(embed_size)(Z)
    Z = tf.keras.layers.Dropout(dropout_rate)(Z)
    Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip]))

```

This code should be mostly straightforward, except for one thing: masking. As of the time of writing, the MultiHeadAttention layer does not support automatic masking,²⁵ so we must handle it manually. How can we do that?

The MultiHeadAttention layer accepts an attention_mask argument, which is a Boolean tensor of shape [*batch size, max query length, max value length*]: for every token in every query sequence, this mask indicates which tokens in the corresponding value sequence should be attended to. We want to tell the MultiHeadAttention layer to ignore all the padding tokens in the values. So, we first compute the padding mask using

`tf.math.not_equal(encoder_input_ids, 0)`. This returns a Boolean tensor of shape [*batch size, max sequence length*]. We then insert a second axis using `[:, tf.newaxis]`, to get a mask of shape [*batch size, 1, max sequence length*]. This allows us to use this mask as the attention_mask when calling the MultiHeadAttention layer: thanks to broadcasting, the same mask will be used for all tokens in each query. This way, the padding tokens in the values will be ignored correctly.

However, the layer will compute outputs for every single query token, including the padding tokens. We need to mask the outputs that correspond to these padding tokens. Recall that we used `mask_zero` in the Embedding layers, and we set `supports_masking` to True in the PositionalEncoding layer, so the automatic mask was propagated all the way to the MultiHeadAttention layer's inputs (`encoder_in`). We can use this to our advantage in the skip connection: indeed, the Add layer supports automatic masking, so when we add `Z` and `skip` (which is initially equal to `encoder_in`), the outputs get

automatically masked correctly.²⁶ Yikes! Masking required much more explanation than code.

Now on to the decoder! Once again, masking is going to be the only tricky part, so let's start with that. The first multi-head attention layer is a self-attention layer, like in the encoder, but it is a *masked* multi-head attention layer, meaning it is causal: it should ignore all tokens in the future. So, we need two masks: a padding mask and a causal mask. Let's create them:

```
decoder_pad_mask = tf.math.not_equal(decoder_input_ids, 0)[:, tf.newaxis]
causal_mask = tf.linalg.band_part( # creates a lower triangular matrix
    tf.ones((batch_max_len_dec, batch_max_len_dec), tf.bool), -1, 0)
```

The padding mask is exactly like the one we created for the encoder, except it's based on the decoder's inputs rather than the encoder's. The causal mask is created using the `tf.linalg.band_part()` function, which takes a tensor and returns a copy with all the values outside a diagonal band set to zero. With these arguments, we get a square matrix of size `batch_max_len_dec` (the max length of the input sequences in the batch), with 1s in the lower-left triangle and 0s in the upper right. If we use this mask as the attention mask, we will get exactly what we want: the first query token will only attend to the first value token, the second will only attend to the first two, the third will only attend to the first three, and so on. In other words, query tokens cannot attend to any value token in the future.

Let's now build the decoder:

```
encoder_outputs = Z # let's save the encoder's final outputs
Z = decoder_in # the decoder starts with its own inputs
for _ in range(N):
    skip = Z
    attn_layer = tf.keras.layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=embed_size, dropout=dropout_rate)
    Z = attn_layer(Z, value=Z, attention_mask=causal_mask & decoder_pad_mask)
    Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()(Z, skip))
    skip = Z
    attn_layer = tf.keras.layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=embed_size, dropout=dropout_rate)
    Z = attn_layer(Z, value=encoder_outputs, attention_mask=encoder_pad_mask)
    Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()(Z, skip))
```

```
skip = Z
Z = tf.keras.layers.Dense(n_units, activation="relu")(Z)
Z = tf.keras.layers.Dense(embed_size)(Z)
Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip]))
```

For the first attention layer, we use causal_mask & decoder_pad_mask to mask both the padding tokens and future tokens. The causal mask only has two dimensions: it's missing the batch dimension, but that's okay since broadcasting ensures that it gets copied across all the instances in the batch.

For the second attention layer, there's nothing special. The only thing to note is that we are using encoder_pad_mask, not decoder_pad_mask, because this attention layer uses the encoder's final outputs as its values.

We're almost done. We just need to add the final output layer, create the model, compile it, and train it:

```
Y_proba = tf.keras.layers.Dense(vocab_size, activation="softmax")(Z)
model = tf.keras.Model(inputs=[encoder_inputs, decoder_inputs],
                       outputs=[Y_proba])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam",
              metrics=["accuracy"])
model.fit((X_train, X_train_dec), Y_train, epochs=10,
          validation_data=((X_valid, X_valid_dec), Y_valid))
```

Congratulations! You've built a full transformer from scratch, and trained it for automatic translation. This is getting quite advanced!

TIP

The Keras team has created a new [Keras NLP project](#), including an API to build a transformer more easily. You may also be interested in the new [Keras CV project](#) for computer vision.

But the field didn't stop there. Let's now explore some of the recent advances.

An Avalanche of Transformer Models

The year 2018 has been called the “ImageNet moment for NLP”. Since then, progress has been astounding, with larger and larger transformer-based architectures trained on immense datasets.

First, the [GPT paper²⁷](#) by Alec Radford and other OpenAI researchers once again demonstrated the effectiveness of unsupervised pretraining, like the ELMo and ULMFiT papers before it, but this time using a transformer-like architecture. The authors pretrained a large but fairly simple architecture composed of a stack of 12 transformer modules using only masked multi-head attention layers, like in the original transformer’s decoder. They trained it on a very large dataset, using the same autoregressive technique we used for our Shakespearean char-RNN: just predict the next token. This is a form of self-supervised learning. Then they fine-tuned it on various language tasks, using only minor adaptations for each task. The tasks were quite diverse: they included text classification, *entailment* (whether sentence A imposes, involves, or implies sentence B as a necessary consequence), ²⁸ similarity (e.g., “Nice weather today” is very similar to “It is sunny”), and question answering (given a few paragraphs of text giving some context, the model must answer some multiple-choice questions).

Then Google’s [BERT paper²⁹](#) came out: it also demonstrated the effectiveness of self-supervised pretraining on a large corpus, using a similar architecture to GPT but with nonmasked multi-head attention layers only, like in the original transformer’s encoder. This means that the model is naturally bidirectional; hence the B in BERT (*Bidirectional Encoder Representations from Transformers*). Most importantly, the authors proposed two pretraining tasks that explain most of the model’s strength:

Masked language model (MLM)

Each word in a sentence has a 15% probability of being masked, and the model is trained to predict the masked words. For example, if the original sentence is “She had fun at the birthday party”, then the model may be

given the sentence “She <mask> fun at the <mask> party” and it must predict the words “had” and “birthday” (the other outputs will be ignored). To be more precise, each selected word has an 80% chance of being masked, a 10% chance of being replaced by a random word (to reduce the discrepancy between pretraining and fine-tuning, since the model will not see <mask> tokens during fine-tuning), and a 10% chance of being left alone (to bias the model toward the correct answer).

Next sentence prediction (NSP)

The model is trained to predict whether two sentences are consecutive or not. For example, it should predict that “The dog sleeps” and “It snores loudly” are consecutive sentences, while “The dog sleeps” and “The Earth orbits the Sun” are not consecutive. Later research showed that NSP was not as important as was initially thought, so it was dropped in most later architectures.

The model is trained on these two tasks simultaneously (see [Figure 16-11](#)). For the NSP task, the authors inserted a class token (<CLS>) at the start of every input, and the corresponding output token represents the model’s prediction: sentence B follows sentence A, or it does not. The two input sentences are concatenated, separated only by a special separation token (<SEP>), and they are fed as input to the model. To help the model know which sentence each input token belongs to, a *segment embedding* is added on top of each token’s positional embeddings: there are just two possible segment embeddings, one for sentence A and one for sentence B. For the MLM task, some input words are masked (as we just saw) and the model tries to predict what those words were. The loss is only computed on the NSP prediction and the masked tokens, not on the unmasked ones.

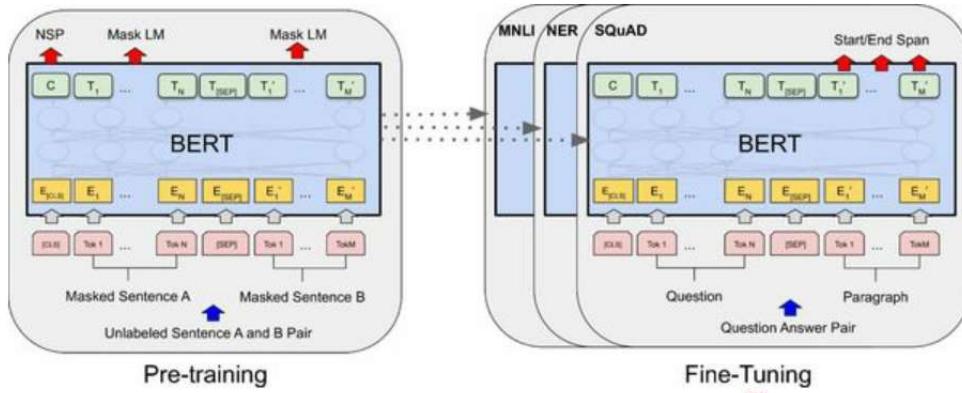


Figure 16-11. BERT training and fine-tuning process ³⁰

After this unsupervised pretraining phase on a very large corpus of text, the model is then fine-tuned on many different tasks, changing very little for each task. For example, for text classification such as sentiment analysis, all output tokens are ignored except for the first one, corresponding to the class token, and a new output layer replaces the previous one, which was just a binary classification layer for NSP.

In February 2019, just a few months after BERT was published, Alec Radford, Jeffrey Wu, and other OpenAI researchers published the [GPT-2 paper](#), ³¹ which proposed a very similar architecture to GPT, but larger still (with over 1.5 billion parameters!). The researchers showed that the new and improved GPT model could perform *zero-shot learning* (ZSL), meaning it could achieve good performance on many tasks without any fine-tuning. This was just the start of a race toward larger and larger models: Google's [Switch Transformers](#)³² (introduced in January 2021) used 1 trillion parameters, and soon much larger models came out, such as the Wu Dao 2.0 model by the Beijing Academy of Artificial Intelligence (BAII), announced in June 2021.

An unfortunate consequence of this trend toward gigantic models is that only well-funded organizations can afford to train such models: it can easily cost hundreds of thousands of dollars or more. And the energy required to train a single model corresponds to an American household's electricity consumption for several years; it's not eco-friendly at all. Many of these models are just too big to even be used on regular hardware: they wouldn't fit

in RAM, and they would be horribly slow. Lastly, some are so costly that they are not released publicly.

Luckily, ingenious researchers are finding new ways to downsize transformers and make them more data-efficient. For example, the [DistilBERT model](#),³³ introduced in October 2019 by Victor Sanh et al. from Hugging Face, is a small and fast transformer model based on BERT. It is available on Hugging Face’s excellent model hub, along with thousands of others—you’ll see an example later in this chapter.

DistilBERT was trained using *distillation* (hence the name): this means transferring knowledge from a teacher model to a student one, which is usually much smaller than the teacher model. This is typically done by using the teacher’s predicted probabilities for each training instance as targets for the student. Surprisingly, distillation often works better than training the student from scratch on the same dataset as the teacher! Indeed, the student benefits from the teacher’s more nuanced labels.

Many more transformer architectures came out after BERT, almost on a monthly basis, often improving on the state of the art across all NLP tasks: XLNet (June 2019), RoBERTa (July 2019), StructBERT (August 2019), ALBERT (September 2019), T5 (October 2019), ELECTRA (March 2020), GPT3 (May 2020), DeBERTa (June 2020), Switch Transformers (January 2021), Wu Dao 2.0 (June 2021), Gopher (December 2021), GPT-NeoX-20B (February 2022), Chinchilla (March 2022), OPT (May 2022), and the list goes on and on. Each of these models brought new ideas and techniques,³⁴ but I particularly like the [T5 paper](#)³⁵ by Google researchers: it frames all NLP tasks as text-to-text, using an encoder–decoder transformer. For example, to translate “I like soccer” to Spanish, you can just call the model with the input sentence “translate English to Spanish: I like soccer” and it outputs “me gusta el fútbol”. To summarize a paragraph, you just enter “summarize:” followed by the paragraph, and it outputs the summary. For classification, you only need to change the prefix to “classify:” and the model outputs the class name, as text. This simplifies using the model, and it also makes it possible to pretrain it on even more tasks.

Last but not least, in April 2022, Google researchers used a new large-scale training platform named *Pathways* (which we will briefly discuss in [Chapter 19](#)) to train a humongous language model named the *Pathways Language Model (PaLM)*, ³⁶ with a whopping 540 billion parameters, using over 6,000 TPUs. Other than its incredible size, this model is a standard transformer, using decoders only (i.e., with masked multi-head attention layers), with just a few tweaks (see the paper for details). This model achieved incredible performance on all sorts of NLP tasks, particularly in natural language understanding (NLU). It's capable of impressive feats, such as explaining jokes, giving detailed step-by-step answers to questions, and even coding. This is in part due to the model's size, but also thanks to a technique called *Chain of thought prompting*, ³⁷ which was introduced a couple months earlier by another team of Google researchers.

In question answering tasks, regular prompting typically includes a few examples of questions and answers, such as: "Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now? A: 11." The prompt then continues with the actual question, such as "Q: John takes care of 10 dogs. Each dog takes .5 hours a day to walk and take care of their business. How many hours a week does he spend taking care of dogs? A:", and the model's job is to append the answer: in this case, "35."

But with chain of thought prompting, the example answers include all the reasoning steps that lead to the conclusion. For example, instead of "A: 11", the prompt contains "A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. $5 + 6 = 11$." This encourages the model to give a detailed answer to the actual question, such as "John takes care of 10 dogs. Each dog takes .5 hours a day to walk and take care of their business. So that is $10 \times .5 = 5$ hours a day. 5 hours a day $\times 7$ days a week = 35 hours a week. The answer is 35 hours a week." This is an actual example from the paper!

Not only does the model give the right answer much more frequently than using regular prompting—we're encouraging the model to think things through—but it also provides all the reasoning steps, which can be useful to better understand the rationale behind a model's answer.

Transformers have taken over NLP, but they didn't stop there: they soon expanded to computer vision as well.

Vision Transformers

One of the first applications of attention mechanisms beyond NMT was in generating image captions using **visual attention**: ³⁸ a convolutional neural network first processes the image and outputs some feature maps, then a decoder RNN equipped with an attention mechanism generates the caption, one word at a time.

At each decoder time step (i.e., each word), the decoder uses the attention model to focus on just the right part of the image. For example, in [Figure 16-12](#), the model generated the caption “A woman is throwing a frisbee in a park”, and you can see what part of the input image the decoder focused its attention on when it was about to output the word “frisbee”: clearly, most of its attention was focused on the frisbee.



Figure 16-12. Visual attention: an input image (left) and the model’s focus before producing the word “frisbee” (right) ³⁹

EXPLAINABILITY

One extra benefit of attention mechanisms is that they make it easier to understand what led the model to produce its output. This is called *explainability*. It can be especially useful when the model makes a

mistake: for example, if an image of a dog walking in the snow is labeled as “a wolf walking in the snow”, then you can go back and check what the model focused on when it output the word “wolf”. You may find that it was paying attention not only to the dog, but also to the snow, hinting at a possible explanation: perhaps the way the model learned to distinguish dogs from wolves is by checking whether or not there’s a lot of snow around. You can then fix this by training the model with more images of wolves without snow, and dogs with snow. This example comes from a great [2016 paper⁴⁰](#) by Marco Tulio Ribeiro et al. that uses a different approach to explainability: learning an interpretable model locally around a classifier’s prediction.

In some applications, explainability is not just a tool to debug a model; it can be a legal requirement—think of a system deciding whether or not it should grant you a loan.

When transformers came out in 2017 and people started to experiment with them beyond NLP, they were first used alongside CNNs, without replacing them. Instead, transformers were generally used to replace RNNs, for example, in image captioning models. Transformers became slightly more visual in a [2020 paper⁴¹](#) by Facebook researchers, which proposed a hybrid CNN–transformer architecture for object detection. Once again, the CNN first processes the input images and outputs a set of feature maps, then these feature maps are converted to sequences and fed to a transformer, which outputs bounding box predictions. But again, most of the visual work is still done by the CNN.

Then, in October 2020, a team of Google researchers released [a paper⁴²](#) that introduced a fully transformer-based vision model, called a *vision transformer* (ViT). The idea is surprisingly simple: just chop the image into little 16×16 squares, and treat the sequence of squares as if it were a sequence of word representations. To be more precise, the squares are first flattened into $16 \times 16 \times 3 = 768$ -dimensional vectors—the 3 is for the RGB color channels—then these vectors go through a linear layer that transforms them but retains their dimensionality. The resulting sequence of vectors can

then be treated just like a sequence of word embeddings: this means adding positional embeddings, and passing the result to the transformer. That's it! This model beat the state of the art on ImageNet image classification, but to be fair the authors had to use over 300 million additional images for training. This makes sense since transformers don't have as many *inductive biases* as convolution neural nets, so they need extra data just to learn things that CNNs implicitly assume.

NOTE

An inductive bias is an implicit assumption made by the model, due to its architecture. For example, linear models implicitly assume that the data is, well, linear. CNNs implicitly assume that patterns learned in one location will likely be useful in other locations as well. RNNs implicitly assume that the inputs are ordered, and that recent tokens are more important than older ones. The more inductive biases a model has, assuming they are correct, the less training data the model will require. But if the implicit assumptions are wrong, then the model may perform poorly even if it is trained on a large dataset.

Just two months later, a team of Facebook researchers released a paper⁴³ that introduced *data-efficient image transformers* (DeiT). Their model achieved competitive results on ImageNet without requiring any additional data for training. The model's architecture is virtually the same as the original ViT, but the authors used a distillation technique to transfer knowledge from state-of-the-art CNN models to their model.

Then, in March 2021, DeepMind released an important paper⁴⁴ that introduced the *Perceiver* architecture. It is a *multimodal* transformer, meaning you can feed it text, images, audio, or virtually any other modality. Until then, transformers had been restricted to fairly short sequences because of the performance and RAM bottleneck in the attention layers. This excluded modalities such as audio or video, and it forced researchers to treat images as sequences of patches, rather than sequences of pixels. The bottleneck is due to self-attention, where every token must attend to every other token: if the input sequence has M tokens, then the attention layer must compute an $M \times M$ matrix, which can be huge if M is very large. The

Perceiver solves this problem by gradually improving a fairly short *latent representation* of the inputs, composed of N tokens—typically just a few hundred. (The word *latent* means hidden, or internal.) The model uses cross-attention layers only, feeding them the latent representation as the queries, and the (possibly large) inputs as the values. This only requires computing an $M \times N$ matrix, so the computational complexity is linear with regard to M , instead of quadratic. After going through several cross-attention layers, if everything goes well, the latent representation ends up capturing everything that matters in the inputs. The authors also suggested sharing the weights between consecutive cross-attention layers: if you do that, then the Perceiver effectively becomes an RNN. Indeed, the shared cross-attention layers can be seen as the same memory cell at different time steps, and the latent representation corresponds to the cell’s context vector. The same inputs are repeatedly fed to the memory cell at every time step. It looks like RNNs are not dead after all!

Just a month later, Mathilde Caron et al. introduced **DINO**,⁴⁵ an impressive vision transformer trained entirely without labels, using self-supervision, and capable of high-accuracy semantic segmentation. The model is duplicated during training, with one network acting as a teacher and the other acting as a student. Gradient descent only affects the student, while the teacher’s weights are just an exponential moving average of the student’s weights. The student is trained to match the teacher’s predictions: since they’re almost the same model, this is called *self-distillation*. At each training step, the input images are augmented in different ways for the teacher and the student, so they don’t see the exact same image, but their predictions must match. This forces them to come up with high-level representations. To prevent *mode collapse*, where both the student and the teacher would always output the same thing, completely ignoring the inputs, DINO keeps track of a moving average of the teacher’s outputs, and it tweaks the teacher’s predictions to ensure that they remain centered on zero, on average. DINO also forces the teacher to have high confidence in its predictions: this is called *sharpening*. Together, these techniques preserve diversity in the teacher’s outputs.

In a 2021 paper,⁴⁶ Google researchers showed how to scale ViTs up or

down, depending on the amount of data. They managed to create a huge 2 billion parameter model that reached over 90.4% top-1 accuracy on ImageNet. Conversely, they also trained a scaled-down model that reached over 84.8% top-1 accuracy on ImageNet, using only 10,000 images: that's just 10 images per class!

And progress in visual transformers has continued steadily to this day. For example, in March 2022, a [paper⁴⁷](#) by Mitchell Wortsman et al. demonstrated that it's possible to first train multiple transformers, then average their weights to create a new and improved model. This is similar to an ensemble (see [Chapter 7](#)), except there's just one model in the end, which means there's no inference time penalty.

The latest trend in transformers consists in building large multimodal models, often capable of zero-shot or few-shot learning. For example, [OpenAI's 2021 CLIP paper⁴⁸](#) proposed a large transformer model pretrained to match captions with images: this task allows it to learn excellent image representations, and the model can then be used directly for tasks such as image classification using simple text prompts such as "a photo of a cat". Soon after, OpenAI announced [DALL·E](#), ⁴⁹ capable of generating amazing images based on text prompts. The [DALL·E 2](#), ⁵⁰ which generates even higher quality images using a diffusion model (see [Chapter 17](#)).

In April 2022, DeepMind released the [Flamingo paper](#), ⁵¹ which introduced a family of models pretrained on a wide variety of tasks across multiple modalities, including text, images, and videos. A single model can be used across very different tasks, such as question answering, image captioning, and more. Soon after, in May 2022, DeepMind introduced [GATO](#), ⁵² a multimodal model that can be used as a policy for a reinforcement learning agent (RL will be introduced in [Chapter 18](#)). The same transformer can chat with you, caption images, play Atari games, control (simulated) robotic arms, and more, all with "only" 1.2 billion parameters. And the adventure continues!

NOTE

These astounding advances have led some researchers to claim that human-level AI is near, that “scale is all you need”, and that some of these models may be “slightly conscious”. Others point out that despite the amazing progress, these models still lack the reliability and adaptability of human intelligence, our ability to reason symbolically, to generalize based on a single example, and more.

As you can see, transformers are everywhere! And the good news is that you generally won’t have to implement transformers yourself since many excellent pretrained models are readily available for download via TensorFlow Hub or Hugging Face’s model hub. You’ve already seen how to use a model from TF Hub, so let’s close this chapter by taking a quick look at Hugging Face’s ecosystem.

Hugging Face's Transformers Library

It's impossible to talk about transformers today without mentioning Hugging Face, an AI company that has built a whole ecosystem of easy-to-use open source tools for NLP, vision, and beyond. The central component of their ecosystem is the Transformers library, which allows you to easily download a pretrained model, including its corresponding tokenizer, and then fine-tune it on your own dataset, if needed. Plus, the library supports TensorFlow, PyTorch, and JAX (with the Flax library).

The simplest way to use the Transformers library is to use the `transformers.pipeline` function: you just specify which task you want, such as sentiment analysis, and it downloads a default pretrained model, ready to be used—it really couldn't be any simpler:

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis") # many other tasks are available
result = classifier("The actors were very convincing.")
```

The result is a Python list containing one dictionary per input text:

```
>>> result
[{'label': 'POSITIVE', 'score': 0.9998071789741516}]
```

In this example, the model correctly found that the sentence is positive, with around 99.98% confidence. Of course, you can also pass a batch of sentences to the model:

```
>>> classifier(["I am from India.", "I am from Iraq."])
[{'label': 'POSITIVE', 'score': 0.9896161556243896},
 {'label': 'NEGATIVE', 'score': 0.9811071157455444}]
```

BIAS AND FAIRNESS

As the output suggests, this specific classifier loves Indians, but is

severely biased against Iraqis. You can try this code with your own country or city. Such an undesirable bias generally comes in large part from the training data itself: in this case, there were plenty of negative sentences related to the wars in Iraq in the training data. This bias was then amplified during the fine-tuning process since the model was forced to choose between just two classes: positive or negative. If you add a neutral class when fine-tuning, then the country bias mostly disappears. But the training data is not the only source of bias: the model's architecture, the type of loss or regularization used for training, the optimizer; all of these can affect what the model ends up learning. Even a mostly unbiased model can be *used* in a biased way, much like survey questions can be biased.

Understanding bias in AI and mitigating its negative effects is still an area of active research, but one thing is certain: you should pause and think before you rush to deploy a model to production. Ask yourself how the model could do harm, even indirectly. For example, if the model's predictions are used to decide whether or not to give someone a loan, the process should be fair. So, make sure you evaluate the model's performance not just on average over the whole test set, but across various subsets as well: for example, you may find that although the model works very well on average, its performance is abysmal for some categories of people. You may also want to run counterfactual tests: for example, you may want to check that the model's predictions do not change when you simply switch someone's gender.

If the model works well on average, it's tempting to push it to production and move on to something else, especially if it's just one component of a much larger system. But in general, if you don't fix such issues, no one else will, and your model may end up doing more harm than good. The solution depends on the problem: it may require rebalancing the dataset, fine-tuning on a different dataset, switching to another pretrained model, tweaking the model's architecture or hyperparameters, etc.

The pipeline() function uses the default model for the given task. For

example, for text classification tasks such as sentiment analysis, at the time of writing, it defaults to distilbert-base-uncased-finetuned-sst-2-english—a DistilBERT model with an uncased tokenizer, trained on English Wikipedia and a corpus of English books, and fine-tuned on the Stanford Sentiment Treebank v2 (SST 2) task. It’s also possible to manually specify a different model. For example, you could use a DistilBERT model fine-tuned on the Multi-Genre Natural Language Inference (MultiNLI) task, which classifies two sentences into three classes: contradiction, neutral, or entailment. Here is how:

```
>>> model_name = "huggingface/distilbert-base-uncased-finetuned-mnli"
>>> classifier_mnli = pipeline("text-classification", model=model_name)
>>> classifier_mnli("She loves me. [SEP] She loves me not.")
[{'label': 'contradiction', 'score': 0.9790192246437073}]
```

TIP

You can find the available models at <https://huggingface.co/models>, and the list of tasks at <https://huggingface.co/tasks>.

The pipeline API is very simple and convenient, but sometimes you will need more control. For such cases, the Transformers library provides many classes, including all sorts of tokenizers, models, configurations, callbacks, and much more. For example, let’s load the same DistilBERT model, along with its corresponding tokenizer, using the TFAutoModelForSequenceClassification and AutoTokenizer classes:

```
from transformers import AutoTokenizer, TFAutoModelForSequenceClassification
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = TFAutoModelForSequenceClassification.from_pretrained(model_name)
```

Next, let’s tokenize a couple of pairs of sentences. In this code, we activate padding and specify that we want TensorFlow tensors instead of Python lists:

```
token_ids = tokenizer(["I like soccer. [SEP] We all love soccer!",
```

```
"Joe lived for a very long time. [SEP] Joe is old."],  
padding=True, return_tensors="tf")
```

TIP

Instead of passing "Sentence 1 [SEP] Sentence 2" to the tokenizer, you can equivalently pass it a tuple: ("Sentence 1", "Sentence 2").

The output is a dictionary-like instance of the BatchEncoding class, which contains the sequences of token IDs, as well as a mask containing 0s for the padding tokens:

```
>>> token_ids  
{'input_ids': <tf.Tensor: shape=(2, 15), dtype=int32, numpy=  
array([[ 101, 1045, 2066, 4715, 1012, 102, 2057, 2035, 2293, 4715, 999,  
       102, 0, 0, 0],  
      [ 101, 3533, 2973, 2005, 1037, 2200, 2146, 2051, 1012, 102, 3533,  
       2003, 2214, 1012, 102]], dtype=int32>},  
'attention_mask': <tf.Tensor: shape=(2, 15), dtype=int32, numpy=  
array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],  
      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]], dtype=int32>}
```

If you set `return_token_type_ids=True` when calling the tokenizer, you will also get an extra tensor that indicates which sentence each token belongs to. This is needed by some models, but not DistilBERT.

Next, we can directly pass this BatchEncoding object to the model; it returns a TFSequenceClassifierOutput object containing its predicted class logits:

```
>>> outputs = model(token_ids)  
>>> outputs  
TFSequenceClassifierOutput(loss=None, logits=[<tf.Tensor: [...] numpy=  
array([-2.1123817, 1.1786783, 1.4101017],  
      [-0.01478387, 1.0962474, -0.9919954]], dtype=float32>], [...])
```

Lastly, we can apply the softmax activation function to convert these logits to class probabilities, and use the `argmax()` function to predict the class with the highest probability for each input sentence pair:

```

>>> Y_probas = tf.keras.activations.softmax(outputs.logits)
>>> Y_probas
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[0.01619702, 0.43523544, 0.5485676 ],
       [0.08672056, 0.85204804, 0.06123142]], dtype=float32)>
>>> Y_pred = tf.argmax(Y_probas, axis=1)
>>> Y_pred # 0 = contradiction, 1 = entailment, 2 = neutral
<tf.Tensor: shape=(2,), dtype=int64, numpy=array([2, 1])>

```

In this example, the model correctly classifies the first sentence pair as neutral (the fact that I like soccer does not imply that everyone else does) and the second pair as an entailment (Joe must indeed be quite old).

If you wish to fine-tune this model on your own dataset, you can train the model as usual with Keras since it's just a regular Keras model with a few extra methods. However, because the model outputs logits instead of probabilities, you must use the

`tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)` loss instead of the usual "sparse_categorical_crossentropy" loss. Moreover, the model does not support BatchEncoding inputs during training, so you must use its data attribute to get a regular dictionary instead:

```

sentences = [("Sky is blue", "Sky is red"), ("I love her", "She loves me")]
X_train = tokenizer(sentences, padding=True, return_tensors="tf").data
y_train = tf.constant([0, 2]) # contradiction, neutral
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(loss=loss, optimizer="adam", metrics=["accuracy"])
history = model.fit(X_train, y_train, epochs=2)

```

Hugging Face has also built a Datasets library that you can use to easily download a standard dataset (such as IMDb) or a custom one, and use it to fine-tune your model. It's similar to TensorFlow Datasets, but it also provides tools to perform common preprocessing tasks on the fly, such as masking. The list of datasets is available at <https://huggingface.co/datasets>.

This should get you started with Hugging Face's ecosystem. To learn more, you can head over to <https://huggingface.co/docs> for the documentation, which includes many tutorial notebooks, videos, the full API, and more. I also recommend you check out the O'Reilly book *Natural Language*

Processing with Transformers: Building Language Applications with Hugging Face by Lewis Tunstall, Leandro von Werra, and Thomas Wolf—all from the Hugging Face team.

In the next chapter we will discuss how to learn deep representations in an unsupervised way using autoencoders, and we will use generative adversarial networks to produce images and more!

Exercises

1. What are the pros and cons of using a stateful RNN versus a stateless RNN?
2. Why do people use encoder–decoder RNNs rather than plain sequence-to-sequence RNNs for automatic translation?
3. How can you deal with variable-length input sequences? What about variable-length output sequences?
4. What is beam search, and why would you use it? What tool can you use to implement it?
5. What is an attention mechanism? How does it help?
6. What is the most important layer in the transformer architecture? What is its purpose?
7. When would you need to use sampled softmax?
8. *Embedded Reber grammars* were used by Hochreiter and Schmidhuber in [their paper](#) about LSTMs. They are artificial grammars that produce strings such as “BPBTSXXVPSEPE”. Check out Jenny Orr’s [nice introduction](#) to this topic, then choose a particular embedded Reber grammar (such as the one represented on Orr’s page), then train an RNN to identify whether a string respects that grammar or not. You will first need to write a function capable of generating a training batch containing about 50% strings that respect the grammar, and 50% that don’t.
9. Train an encoder–decoder model that can convert a date string from one format to another (e.g., from “April 22, 2019” to “2019-04-22”).
10. Go through the example on the Keras website for [“Natural language image search with a Dual Encoder”](#). You will learn how to build a model capable of representing both images and text within the same

embedding space. This makes it possible to search for images using a text prompt, like in the CLIP model by OpenAI.

11. Use the Hugging Face Transformers library to download a pretrained language model capable of generating text (e.g., GPT), and try generating more convincing Shakespearean text. You will need to use the model’s `generate()` method—see Hugging Face’s documentation for more details.

Solutions to these exercises are available at the end of this chapter’s notebook, at <https://homl.info/colab3>.

-
- 1 Alan Turing, “Computing Machinery and Intelligence”, *Mind* 49 (1950): 433–460.
 - 2 Of course, the word *chatbot* came much later. Turing called his test the *imitation game*: machine A and human B chat with human interrogator C via text messages; the interrogator asks questions to figure out which one is the machine (A or B). The machine passes the test if it can fool the interrogator, while the human B must try to help the interrogator.
 - 3 Since the input windows overlap, the concept of *epoch* is not so clear in this case: during each epoch (as implemented by Keras), the model will actually see the same character multiple times.
 - 4 Alec Radford et al., “Learning to Generate Reviews and Discovering Sentiment”, arXiv preprint arXiv:1704.01444 (2017).
 - 5 Rico Sennrich et al., “Neural Machine Translation of Rare Words with Subword Units”, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* 1 (2016): 1715–1725.
 - 6 Taku Kudo, “Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates”, arXiv preprint arXiv:1804.10959 (2018).
 - 7 Taku Kudo and John Richardson, “SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing”, arXiv preprint arXiv:1808.06226 (2018).
 - 8 Yonghui Wu et al., “Google’s Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation”, arXiv preprint arXiv:1609.08144 (2016).
 - 9 Ragged tensors were introduced in [Chapter 12](#), and they are detailed in [Appendix C](#).
 - 10 Matthew Peters et al., “Deep Contextualized Word Representations”, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2018): 2227–2237.
 - 11 Jeremy Howard and Sebastian Ruder, “Universal Language Model Fine-Tuning for Text Classification”, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics* (2018): 3287–3297.

Linguistics 1 (2018): 328–339.

- 12 Daniel Cer et al., “Universal Sentence Encoder”, arXiv preprint arXiv:1803.11175 (2018).
- 13 Ilya Sutskever et al., “Sequence to Sequence Learning with Neural Networks”, arXiv preprint (2014).
- 14 Samy Bengio et al., “Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks”, arXiv preprint arXiv:1506.03099 (2015).
- 15 This dataset is composed of sentence pairs created by contributors of the [Tatoeba project](#). About 120,000 sentence pairs were selected by the authors of the website <https://manythings.org/anki>. This dataset is released under the Creative Commons Attribution 2.0 France license. Other language pairs are available.
- 16 In Python, if you run `a, *b = [1, 2, 3, 4]`, then `a` equals 1 and `b` equals `[2, 3, 4]`.
- 17 Sébastien Jean et al., “On Using Very Large Target Vocabulary for Neural Machine Translation”, *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing* 1 (2015): 1–10.
- 18 Dzmitry Bahdanau et al., “Neural Machine Translation by Jointly Learning to Align and Translate”, arXiv preprint arXiv:1409.0473 (2014).
- 19 Minh-Thang Luong et al., “Effective Approaches to Attention-Based Neural Machine Translation”, *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (2015): 1412–1421.
- 20 Ashish Vaswani et al., “Attention Is All You Need”, *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 6000–6010.
- 21 Since the transformer uses time-distributed dense layers, you could argue that it uses 1D convolutional layers with a kernel size of 1.
- 22 This is figure 1 from the “Attention Is All You Need” paper, reproduced with the kind permission of the authors.
- 23 It’s possible to use ragged tensors instead, if you are using the latest version of TensorFlow.
- 24 This is the righthand part of figure 2 from “Attention Is All You Need”, reproduced with the kind authorization of the authors.
- 25 This will most likely change by the time you read this; check out [Keras issue #16248](#) for more details. When this happens, there will be no need to set the `attention_mask` argument, and therefore no need to create `encoder_pad_mask`.
- 26 Currently `Z + skip` does not support automatic masking, which is why we had to write `tf.keras.layers.Add()([Z, skip])` instead. Again, this may change by the time you read this.
- 27 Alec Radford et al., “Improving Language Understanding by Generative Pre-Training” (2018).
- 28 For example, the sentence “Jane had a lot of fun at her friend’s birthday party” entails “Jane enjoyed the party”, but it is contradicted by “Everyone hated the party” and it is unrelated to “The Earth is flat”.

- 29 Jacob Devlin et al., “BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding”, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2019).
- 30 This is figure 1 from the paper, reproduced with the kind authorization of the authors.
- 31 Alec Radford et al., “Language Models Are Unsupervised Multitask Learners” (2019).
- 32 William Fedus et al., “Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity” (2021).
- 33 Victor Sanh et al., “DistilBERT, A Distilled Version of Bert: Smaller, Faster, Cheaper and Lighter”, arXiv preprint arXiv:1910.01108 (2019).
- 34 Mariya Yao summarized many of these models in this post: <https://homl.info/yaopost>.
- 35 Colin Raffel et al., “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”, arXiv preprint arXiv:1910.10683 (2019).
- 36 Aakanksha Chowdhery et al., “PaLM: Scaling Language Modeling with Pathways”, arXiv preprint arXiv:2204.02311 (2022).
- 37 Jason Wei et al., “Chain of Thought Prompting Elicits Reasoning in Large Language Models”, arXiv preprint arXiv:2201.11903 (2022).
- 38 Kelvin Xu et al., “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention”, *Proceedings of the 32nd International Conference on Machine Learning* (2015): 2048–2057.
- 39 This is a part of figure 3 from the paper. It is reproduced with the kind authorization of the authors.
- 40 Marco Tulio Ribeiro et al., “‘Why Should I Trust You?’: Explaining the Predictions of Any Classifier”, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016): 1135–1144.
- 41 Nicolas Carion et al., “End-to-End Object Detection with Transformers”, arXiv preprint arxiv:2005.12872 (2020).
- 42 Alexey Dosovitskiy et al., “An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale”, arXiv preprint arxiv:2010.11929 (2020).
- 43 Hugo Touvron et al., “Training Data-Efficient Image Transformers & Distillation Through Attention”, arXiv preprint arxiv:2012.12877 (2020).
- 44 Andrew Jaegle et al., “Perceiver: General Perception with Iterative Attention”, arXiv preprint arxiv:2103.03206 (2021).
- 45 Mathilde Caron et al., “Emerging Properties in Self-Supervised Vision Transformers”, arXiv preprint arxiv:2104.14294 (2021).
- 46 Xiaohua Zhai et al., “Scaling Vision Transformers”, arXiv preprint arxiv:2106.04560v1 (2021).
- 47 Mitchell Wortsman et al., “Model Soups: Averaging Weights of Multiple Fine-tuned Models Improves Accuracy Without Increasing Inference Time”, arXiv preprint arxiv:2203.05482v1 (2022).

- 48 Alec Radford et al., “Learning Transferable Visual Models From Natural Language Supervision”, arXiv preprint arxiv:2103.00020 (2021).
- 49 Aditya Ramesh et al., “Zero-Shot Text-to-Image Generation”, arXiv preprint arxiv:2102.12092 (2021).
- 50 Aditya Ramesh et al., “Hierarchical Text-Conditional Image Generation with CLIP Latents”, arXiv preprint arxiv:2204.06125 (2022).
- 51 Jean-Baptiste Alayrac et al., “Flamingo: a Visual Language Model for Few-Shot Learning”, arXiv preprint arxiv:2204.14198 (2022).
- 52 Scott Reed et al., “A Generalist Agent”, arXiv preprint arxiv:2205.06175 (2022).

Chapter 17. Autoencoders, GANs, and Diffusion Models

Autoencoders are artificial neural networks capable of learning dense representations of the input data, called *latent representations* or *codings*, without any supervision (i.e., the training set is unlabeled). These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction (see [Chapter 8](#)), especially for visualization purposes. Autoencoders also act as feature detectors, and they can be used for unsupervised pretraining of deep neural networks (as we discussed in [Chapter 11](#)). Lastly, some autoencoders are *generative models*: they are capable of randomly generating new data that looks very similar to the training data. For example, you could train an autoencoder on pictures of faces, and it would then be able to generate new faces.

Generative adversarial networks (GANs) are also neural nets capable of generating data. In fact, they can generate pictures of faces so convincing that it is hard to believe the people they represent do not exist. You can judge so for yourself by visiting <https://thispersondoesnotexist.com>, a website that shows faces generated by a GAN architecture called *StyleGAN*. You can also check out <https://thisrentaldoesnotexist.com> to see some generated Airbnb listings. GANs are now widely used for super resolution (increasing the resolution of an image), [colorization](#), powerful image editing (e.g., replacing photo bombers with realistic background), turning simple sketches into photorealistic images, predicting the next frames in a video, augmenting a dataset (to train other models), generating other types of data (such as text, audio, and time series), identifying the weaknesses in other models to strengthen them, and more.

A more recent addition to the generative learning party is *diffusion models*. In 2021, they managed to generate more diverse and higher-quality images than GANs, while also being much easier to train. However, diffusion models are

much slower to run.

Autoencoders, GANs, and diffusion models are all unsupervised, they all learn latent representations, they can all be used as generative models, and they have many similar applications. However, they work very differently:

- Autoencoders simply learn to copy their inputs to their outputs. This may sound like a trivial task, but as you will see, constraining the network in various ways can make it rather difficult. For example, you can limit the size of the latent representations, or you can add noise to the inputs and train the network to recover the original inputs. These constraints prevent the autoencoder from trivially copying the inputs directly to the outputs, which forces it to learn efficient ways of representing the data. In short, the codings are byproducts of the autoencoder learning the identity function under some constraints.
- GANs are composed of two neural networks: a *generator* that tries to generate data that looks similar to the training data, and a *discriminator* that tries to tell real data from fake data. This architecture is very original in deep learning in that the generator and the discriminator compete against each other during training: the generator is often compared to a criminal trying to make realistic counterfeit money, while the discriminator is like the police investigator trying to tell real money from fake. *Adversarial training* (training competing neural networks) is widely considered one of the most important innovations of the 2010s. In 2016, Yann LeCun even said that it was “the most interesting idea in the last 10 years in machine learning”.
- A *denoising diffusion probabilistic model* (DDPM) is trained to remove a tiny bit of noise from an image. If you then take an image entirely full of Gaussian noise and repeatedly run the diffusion model on that image, a high-quality image will gradually emerge, similar to the training images (but not identical).

In this chapter we will start by exploring in more depth how autoencoders work and how to use them for dimensionality reduction, feature extraction,

unsupervised pretraining, or as generative models. This will naturally lead us to GANs. We will build a simple GAN to generate fake images, but we will see that training is often quite difficult. We will discuss the main difficulties you will encounter with adversarial training, as well as some of the main techniques to work around these difficulties. And lastly, we will build and train a DDPM and use it to generate images. Let's start with autoencoders!

Efficient Data Representations

Which of the following number sequences do you find the easiest to memorize?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14

At first glance, it would seem that the first sequence should be easier, since it is much shorter. However, if you look carefully at the second sequence, you will notice that it is just the list of even numbers from 50 down to 14. Once you notice this pattern, the second sequence becomes much easier to memorize than the first because you only need to remember the pattern (i.e., decreasing even numbers) and the starting and ending numbers (i.e., 50 and 14). Note that if you could quickly and easily memorize very long sequences, you would not care much about the existence of a pattern in the second sequence. You would just learn every number by heart, and that would be that. The fact that it is hard to memorize long sequences is what makes it useful to recognize patterns, and hopefully this clarifies why constraining an autoencoder during training pushes it to discover and exploit patterns in the data.

The relationship between memory, perception, and pattern matching was famously studied by [William Chase and Herbert Simon¹](#) in the early 1970s. They observed that expert chess players were able to memorize the positions of all the pieces in a game by looking at the board for just five seconds, a task that most people would find impossible. However, this was only the case when the pieces were placed in realistic positions (from actual games), not when the pieces were placed randomly. Chess experts don't have a much better memory than you and I; they just see chess patterns more easily, thanks to their experience with the game. Noticing patterns helps them store information efficiently.

Just like the chess players in this memory experiment, an autoencoder looks

at the inputs, converts them to an efficient latent representation, and then spits out something that (hopefully) looks very close to the inputs. An autoencoder is always composed of two parts: an *encoder* (or) that converts the inputs to a latent representation, followed by a *decoder* (or *generative network*) that converts the internal representation to the outputs (see Figure 17-1).

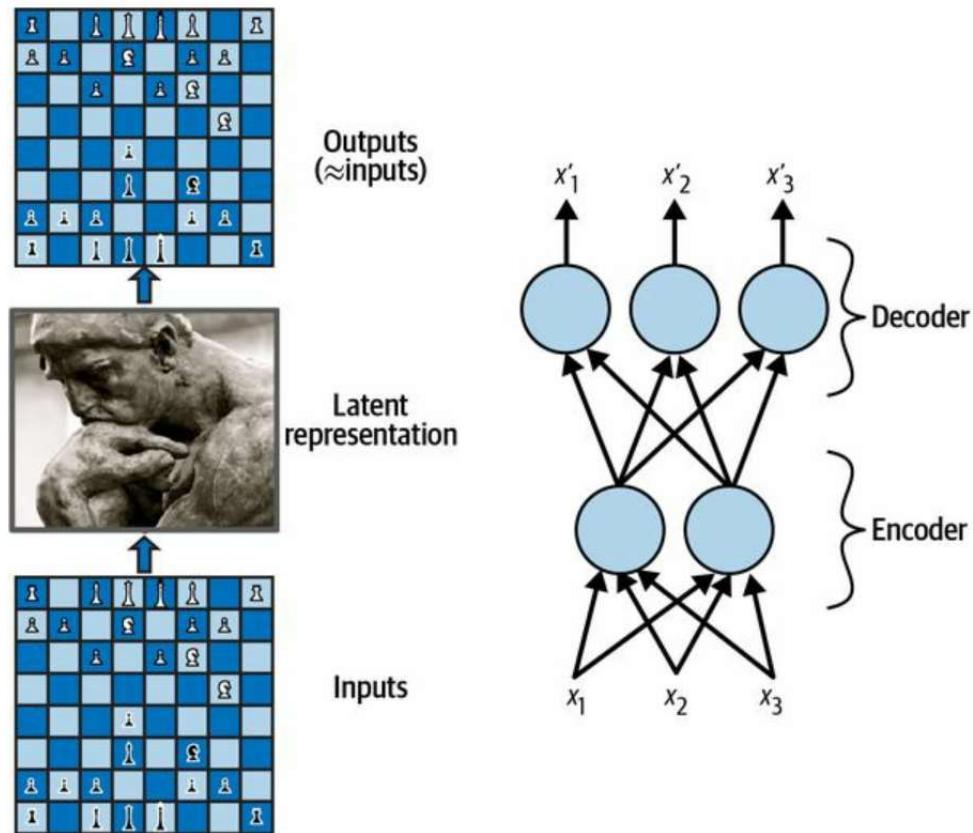


Figure 17-1. The chess memory experiment (left) and a simple autoencoder (right)

As you can see, an autoencoder typically has the same architecture as a multilayer perceptron (MLP; see Chapter 10), except that the number of neurons in the output layer must be equal to the number of inputs. In this example, there is just one hidden layer composed of two neurons (the encoder), and one output layer composed of three neurons (the decoder). The

outputs are often called the *reconstructions* because the autoencoder tries to reconstruct the inputs. The cost function contains a *reconstruction loss* that penalizes the model when the reconstructions are different from the inputs.

Because the internal representation has a lower dimensionality than the input data (it is 2D instead of 3D), the autoencoder is said to be *undercomplete*. An undercomplete autoencoder cannot trivially copy its inputs to the codings, yet it must find a way to output a copy of its inputs. It is forced to learn the most important features in the input data (and drop the unimportant ones).

Let's see how to implement a very simple undercomplete autoencoder for dimensionality reduction.

Performing PCA with an Undercomplete Linear Autoencoder

If the autoencoder uses only linear activations and the cost function is the mean squared error (MSE), then it ends up performing principal component analysis (PCA; see [Chapter 8](#)).

The following code builds a simple linear autoencoder to perform PCA on a 3D dataset, projecting it to 2D:

```
import tensorflow as tf

encoder = tf.keras.Sequential([tf.keras.layers.Dense(2)])
decoder = tf.keras.Sequential([tf.keras.layers.Dense(3)])
autoencoder = tf.keras.Sequential([encoder, decoder])

optimizer = tf.keras.optimizers.SGD(learning_rate=0.5)
autoencoder.compile(loss="mse", optimizer=optimizer)
```

This code is really not very different from all the MLPs we built in past chapters, but there are a few things to note:

- We organized the autoencoder into two subcomponents: the encoder and the decoder. Both are regular Sequential models with a single Dense layer each, and the autoencoder is a Sequential model containing the encoder followed by the decoder (remember that a model can be used as a layer in another model).
- The autoencoder's number of outputs is equal to the number of inputs (i.e., 3).
- To perform PCA, we do not use any activation function (i.e., all neurons are linear), and the cost function is the MSE. That's because PCA is a linear transformation. We will see more complex and nonlinear autoencoders shortly.

Now let's train the model on the same simple generated 3D dataset we used

in [Chapter 8](#) and use it to encode that dataset (i.e., project it to 2D):

```
X_train = [...] # generate a 3D dataset, like in Chapter 8  
history = autoencoder.fit(X_train, X_train, epochs=500, verbose=False)  
codings = encoder.predict(X_train)
```

Note that `X_train` is used as both the inputs and the targets. [Figure 17-2](#) shows the original 3D dataset (on the left) and the output of the autoencoder's hidden layer (i.e., the coding layer, on the right). As you can see, the autoencoder found the best 2D plane to project the data onto, preserving as much variance in the data as it could (just like PCA).

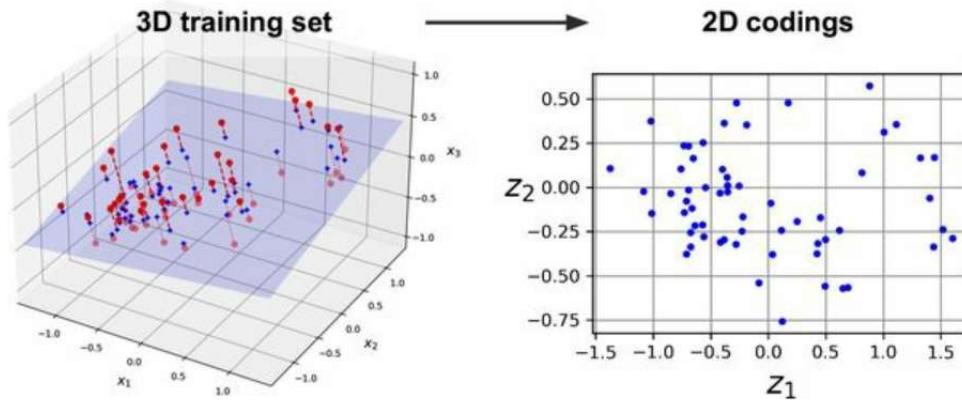


Figure 17-2. Approximate PCA performed by an undercomplete linear autoencoder

NOTE

You can think of an autoencoder as performing a form of self-supervised learning, since it is based on a supervised learning technique with automatically generated labels (in this case simply equal to the inputs).

Stacked Autoencoders

Just like other neural networks we have discussed, autoencoders can have multiple hidden layers. In this case they are called *stacked autoencoders* (or *deep autoencoders*). Adding more layers helps the autoencoder learn more complex codings. That said, one must be careful not to make the autoencoder too powerful. Imagine an encoder so powerful that it just learns to map each input to a single arbitrary number (and the decoder learns the reverse mapping). Obviously such an autoencoder will reconstruct the training data perfectly, but it will not have learned any useful data representation in the process, and it is unlikely to generalize well to new instances.

The architecture of a stacked autoencoder is typically symmetrical with regard to the central hidden layer (the coding layer). To put it simply, it looks like a sandwich. For example, an autoencoder for Fashion MNIST (introduced in [Chapter 10](#)) may have 784 inputs, followed by a hidden layer with 100 neurons, then a central hidden layer of 30 neurons, then another hidden layer with 100 neurons, and an output layer with 784 neurons. This stacked autoencoder is represented in [Figure 17-3](#).

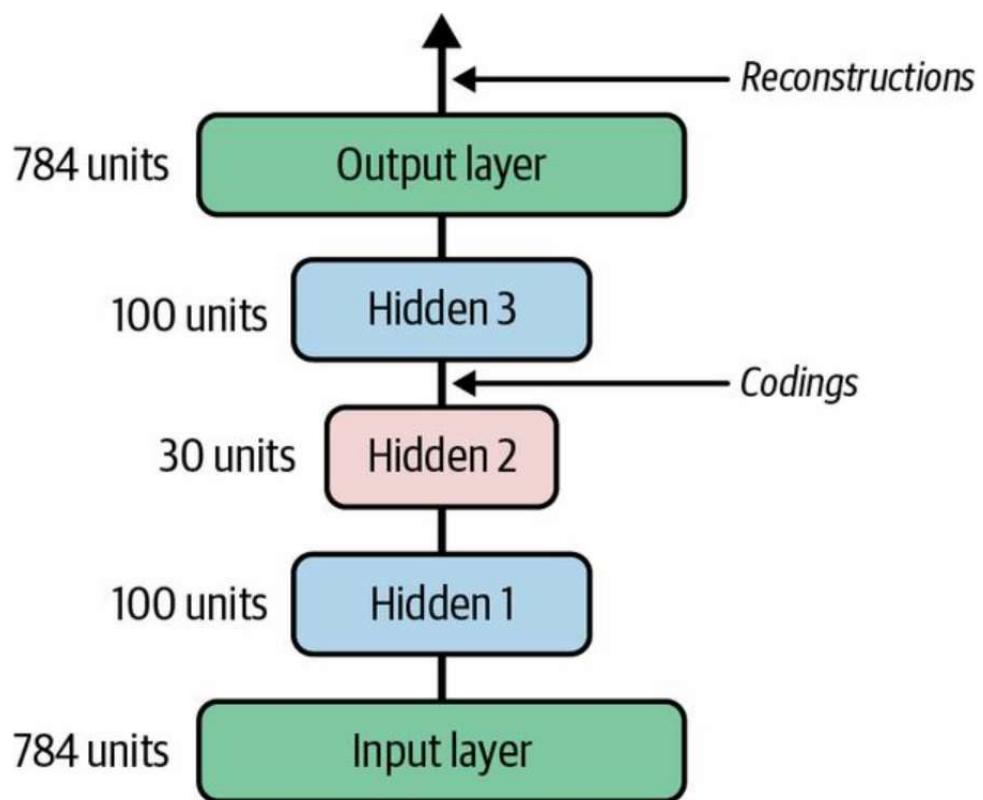


Figure 17-3. Stacked autoencoder

Implementing a Stacked Autoencoder Using Keras

You can implement a stacked autoencoder very much like a regular deep MLP:

```
stacked_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(30, activation="relu"),
])
stacked_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
])
stacked_ae = tf.keras.Sequential([stacked_encoder, stacked_decoder])

stacked_ae.compile(loss="mse", optimizer="adam")
history = stacked_ae.fit(X_train, X_train, epochs=20,
                         validation_data=(X_valid, X_valid))
```

Let's go through this code:

- Just like earlier, we split the autoencoder model into two submodels: the encoder and the decoder.
- The encoder takes 28×28 -pixel grayscale images, flattens them so that each image is represented as a vector of size 784, then processes these vectors through two Dense layers of diminishing sizes (100 units then 30 units), both using the ReLU activation function. For each input image, the encoder outputs a vector of size 30.
- The decoder takes codings of size 30 (output by the encoder) and processes them through two Dense layers of increasing sizes (100 units then 784 units), and it reshapes the final vectors into 28×28 arrays so the decoder's outputs have the same shape as the encoder's inputs.
- When compiling the stacked autoencoder, we use the MSE loss and Nadam optimization.

- Finally, we train the model using `X_train` as both the inputs and the targets. Similarly, we use `X_valid` as both the validation inputs and targets.

Visualizing the Reconstructions

One way to ensure that an autoencoder is properly trained is to compare the inputs and the outputs: the differences should not be too significant. Let's plot a few images from the validation set, as well as their reconstructions:

```
import numpy as np

def plot_reconstructions(model, images=X_valid, n_images=5):
    reconstructions = np.clip(model.predict(images[:n_images]), 0, 1)
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plt.imshow(images[image_index], cmap="binary")
        plt.axis("off")
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plt.imshow(reconstructions[image_index], cmap="binary")
        plt.axis("off")

plot_reconstructions(stacked_ae)
plt.show()
```

Figure 17-4 shows the resulting images.



Figure 17-4. Original images (top) and their reconstructions (bottom)

The reconstructions are recognizable, but a bit too lossy. We may need to train the model for longer, or make the encoder and decoder deeper, or make the codings larger. But if we make the network too powerful, it will manage to make perfect reconstructions without having learned any useful patterns in

the data. For now, let's go with this model.

Visualizing the Fashion MNIST Dataset

Now that we have trained a stacked autoencoder, we can use it to reduce the dataset's dimensionality. For visualization, this does not give great results compared to other dimensionality reduction algorithms (such as those we discussed in [Chapter 8](#)), but one big advantage of autoencoders is that they can handle large datasets with many instances and many features. So, one strategy is to use an autoencoder to reduce the dimensionality down to a reasonable level, then use another dimensionality reduction algorithm for visualization. Let's use this strategy to visualize Fashion MNIST. First we'll use the encoder from our stacked autoencoder to reduce the dimensionality down to 30, then we'll use Scikit-Learn's implementation of the t-SNE algorithm to reduce the dimensionality down to 2 for visualization:

```
from sklearn.manifold import TSNE

X_valid_compressed = stacked_encoder.predict(X_valid)
tsne = TSNE(init="pca", learning_rate="auto", random_state=42)
X_valid_2D = tsne.fit_transform(X_valid_compressed)
```

Now we can plot the dataset:

```
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap="tab10")
plt.show()
```

[Figure 17-5](#) shows the resulting scatterplot, beautified a bit by displaying some of the images. The t-SNE algorithm identified several clusters that match the classes reasonably well (each class is represented by a different color).

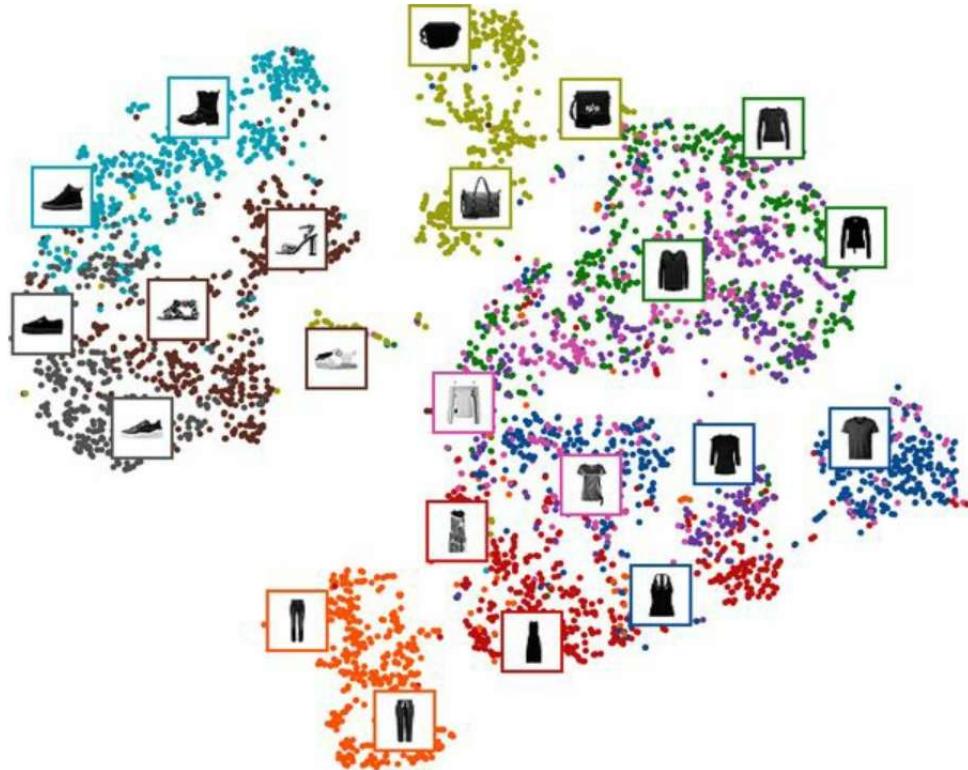


Figure 17-5. Fashion MNIST visualization using an autoencoder followed by t-SNE

So, autoencoders can be used for dimensionality reduction. Another application is for unsupervised pretraining.

Unsupervised Pretraining Using Stacked Autoencoders

As we discussed in [Chapter 11](#), if you are tackling a complex supervised task but you do not have a lot of labeled training data, one solution is to find a neural network that performs a similar task and reuse its lower layers. This makes it possible to train a high-performance model using little training data because your neural network won't have to learn all the low-level features; it will just reuse the feature detectors learned by the existing network.

Similarly, if you have a large dataset but most of it is unlabeled, you can first train a stacked autoencoder using all the data, then reuse the lower layers to create a neural network for your actual task and train it using the labeled data. For example, [Figure 17-6](#) shows how to use a stacked autoencoder to perform unsupervised pretraining for a classification neural network. When training the classifier, if you really don't have much labeled training data, you may want to freeze the pretrained layers (at least the lower ones).

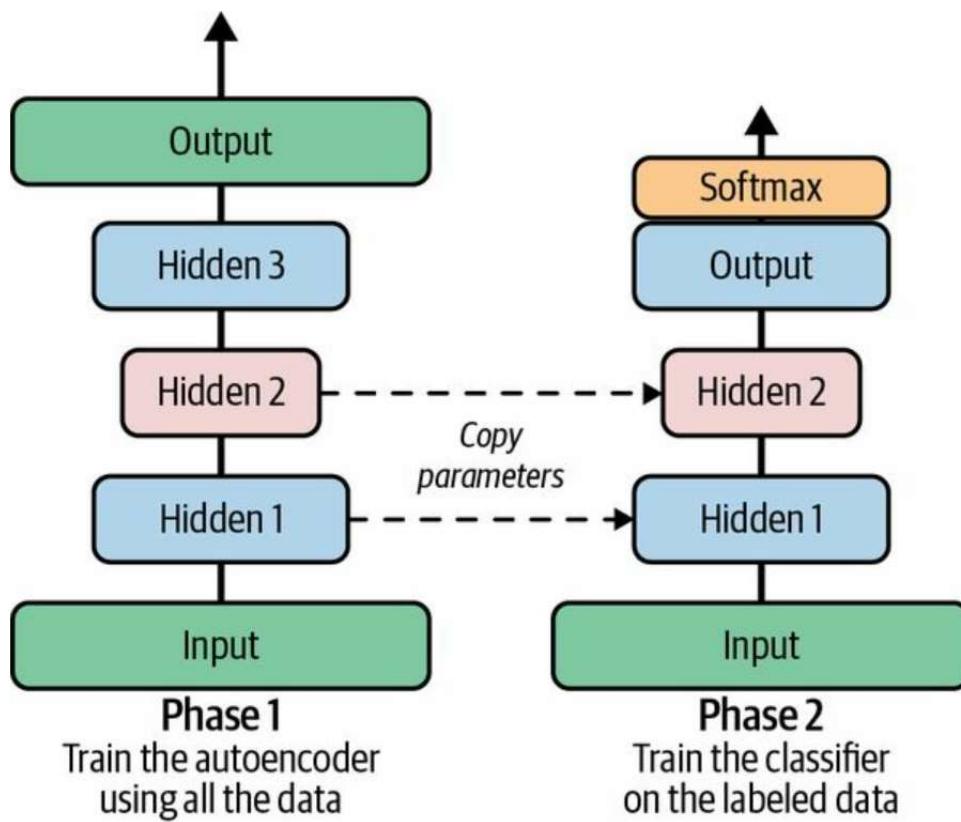


Figure 17-6. Unsupervised pretraining using autoencoders

NOTE

Having plenty of unlabeled data and little labeled data is common. Building a large unlabeled dataset is often cheap (e.g., a simple script can download millions of images off the internet), but labeling those images (e.g., classifying them as cute or not) can usually be done reliably only by humans. Labeling instances is time-consuming and costly, so it's normal to have only a few thousand human-labeled instances, or even less.

There is nothing special about the implementation: just train an autoencoder using all the training data (labeled plus unlabeled), then reuse its encoder layers to create a new neural network (see the exercises at the end of this chapter for an example).

Next, let's look at a few techniques for training stacked autoencoders.

Tying Weights

When an autoencoder is neatly symmetrical, like the one we just built, a common technique is to *tie* the weights of the decoder layers to the weights of the encoder layers. This halves the number of weights in the model, speeding up training and limiting the risk of overfitting. Specifically, if the autoencoder has a total of N layers (not counting the input layer), and \mathbf{W}_L represents the connection weights of the L^{th} layer (e.g., layer 1 is the first hidden layer, layer $N/2$ is the coding layer, and layer N is the output layer), then the decoder layer weights can be defined as $\mathbf{W}_L = \mathbf{W}_{N-L+1}^T$ (with $L = N / 2 + 1, \dots, N$).

To tie weights between layers using Keras, let's define a custom layer:

```
class DenseTranspose(tf.keras.layers.Layer):
    def __init__(self, dense, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.dense = dense
        self.activation = tf.keras.activations.get(activation)

    def build(self, batch_input_shape):
        self.biases = self.add_weight(name="bias",
                                      shape=self.dense.input_shape[-1],
                                      initializer="zeros")
        super().build(batch_input_shape)

    def call(self, inputs):
        Z = tf.matmul(inputs, self.dense.weights[0], transpose_b=True)
        return self.activation(Z + self.biases)
```

This custom layer acts like a regular Dense layer, but it uses another Dense layer's weights, transposed (setting transpose_b=True is equivalent to transposing the second argument, but it's more efficient as it performs the transposition on the fly within the matmul() operation). However, it uses its own bias vector. Now we can build a new stacked autoencoder, much like the previous one but with the decoder's Dense layers tied to the encoder's Dense layers:

```
dense_1 = tf.keras.layers.Dense(100, activation="relu")
dense_2 = tf.keras.layers.Dense(30, activation="relu")

tied_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    dense_1,
    dense_2
])

tied_decoder = tf.keras.Sequential([
    DenseTranspose(dense_2, activation="relu"),
    DenseTranspose(dense_1),
    tf.keras.layers.Reshape([28, 28])
])

tied_ae = tf.keras.Sequential([tied_encoder, tied_decoder])
```

This model achieves roughly the same reconstruction error as the previous model, using almost half the number of parameters.

Training One Autoencoder at a Time

Rather than training the whole stacked autoencoder in one go like we just did, it is possible to train one shallow autoencoder at a time, then stack all of them into a single stacked autoencoder (hence the name), as shown in [Figure 17-7](#). This technique is not used so much these days, but you may still run into papers that talk about “greedy layerwise training”, so it’s good to know what it means.

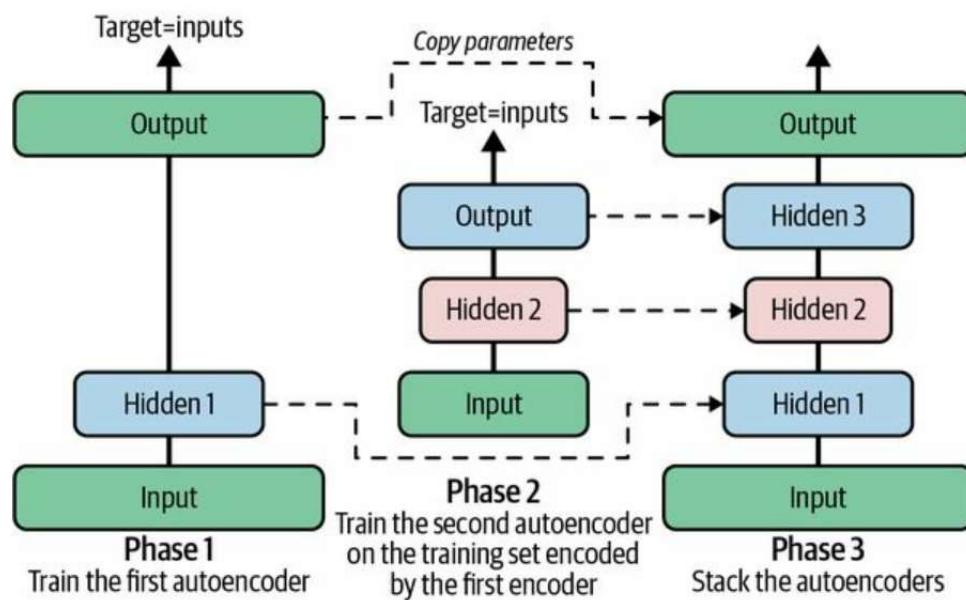


Figure 17-7. Training one autoencoder at a time

During the first phase of training, the first autoencoder learns to reconstruct the inputs. Then we encode the whole training set using this first autoencoder, and this gives us a new (compressed) training set. We then train a second autoencoder on this new dataset. This is the second phase of training. Finally, we build a big sandwich using all these autoencoders, as shown in [Figure 17-7](#) (i.e., we first stack the hidden layers of each autoencoder, then the output layers in reverse order). This gives us the final stacked autoencoder (see the “Training One Autoencoder at a Time” section in the chapter’s notebook for an implementation). We could easily train more

autoencoders this way, building a very deep stacked autoencoder.

As I mentioned earlier, one of the triggers of the deep learning tsunami was the discovery in 2006 by **Geoffrey Hinton et al.** that deep neural networks can be pretrained in an unsupervised fashion, using this greedy layerwise approach. They used restricted Boltzmann machines (RBMs; see <https://homl.info/extr-anns>) for this purpose, but in 2007 **Yoshua Bengio et al.**² showed that autoencoders worked just as well. For several years this was the only efficient way to train deep nets, until many of the techniques introduced in [Chapter 11](#) made it possible to just train a deep net in one shot.

Autoencoders are not limited to dense networks: you can also build convolutional autoencoders. Let's look at these now.

Convolutional Autoencoders

If you are dealing with images, then the autoencoders we have seen so far will not work well (unless the images are very small): as you saw in [Chapter 14](#), convolutional neural networks are far better suited than dense networks to working with images. So if you want to build an autoencoder for images (e.g., for unsupervised pretraining or dimensionality reduction), you will need to build a *convolutional autoencoder*.³ The encoder is a regular CNN composed of convolutional layers and pooling layers. It typically reduces the spatial dimensionality of the inputs (i.e., height and width) while increasing the depth (i.e., the number of feature maps). The decoder must do the reverse (upscale the image and reduce its depth back to the original dimensions), and for this you can use transpose convolutional layers (alternatively, you could combine upsampling layers with convolutional layers). Here is a basic convolutional autoencoder for Fashion MNIST:

```
conv_encoder = tf.keras.Sequential([
    tf.keras.layers.Reshape([28, 28, 1]),
    tf.keras.layers.Conv2D(16, 3, padding="same", activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2), # output: 14 x 14 x 16
    tf.keras.layers.Conv2D(32, 3, padding="same", activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2), # output: 7 x 7 x 32
    tf.keras.layers.Conv2D(64, 3, padding="same", activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2), # output: 3 x 3 x 64
    tf.keras.layers.Conv2D(30, 3, padding="same", activation="relu"),
    tf.keras.layers.GlobalAvgPool2D() # output: 30
])
conv_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(3 * 3 * 16),
    tf.keras.layers.Reshape((3, 3, 16)),
    tf.keras.layers.Conv2DTranspose(32, 3, strides=2, activation="relu"),
    tf.keras.layers.Conv2DTranspose(16, 3, strides=2, padding="same",
                                  activation="relu"),
    tf.keras.layers.Conv2DTranspose(1, 3, strides=2, padding="same"),
    tf.keras.layers.Reshape([28, 28])
])
conv_ae = tf.keras.Sequential([conv_encoder, conv_decoder])
```

It's also possible to create autoencoders with other architecture types, such as

RNNs (see the notebook for an example).

OK, let's step back for a second. So far we have looked at various kinds of autoencoders (basic, stacked, and convolutional), and how to train them (either in one shot or layer by layer). We also looked at a couple of applications: data visualization and unsupervised pretraining.

Up to now, in order to force the autoencoder to learn interesting features, we have limited the size of the coding layer, making it undercomplete. There are actually many other kinds of constraints that can be used, including ones that allow the coding layer to be just as large as the inputs, or even larger, resulting in an *overcomplete autoencoder*. Then, in the following sections we'll look at a few more kinds of autoencoders: denoising autoencoders, sparse autoencoders, and variational autoencoders.

Denoising Autoencoders

Another way to force the autoencoder to learn useful features is to add noise to its inputs, training it to recover the original, noise-free inputs. This idea has been around since the 1980s (e.g., it is mentioned in Yann LeCun's 1987 master's thesis). In a [2008 paper](#), ⁴ Pascal Vincent et al. showed that autoencoders could also be used for feature extraction. In a [2010 paper](#), ⁵ Vincent et al. introduced *stacked denoising autoencoders*.

The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched-off inputs, just like in dropout (introduced in [Chapter 11](#)). [Figure 17-8](#) shows both options.

The implementation is straightforward: it is a regular stacked autoencoder with an additional Dropout layer applied to the encoder's inputs (or you could use a GaussianNoise layer instead). Recall that the Dropout layer is only active during training (and so is the GaussianNoise layer):

```
dropout_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(30, activation="relu")
])
dropout_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
])
dropout_ae = tf.keras.Sequential([dropout_encoder, dropout_decoder])
```

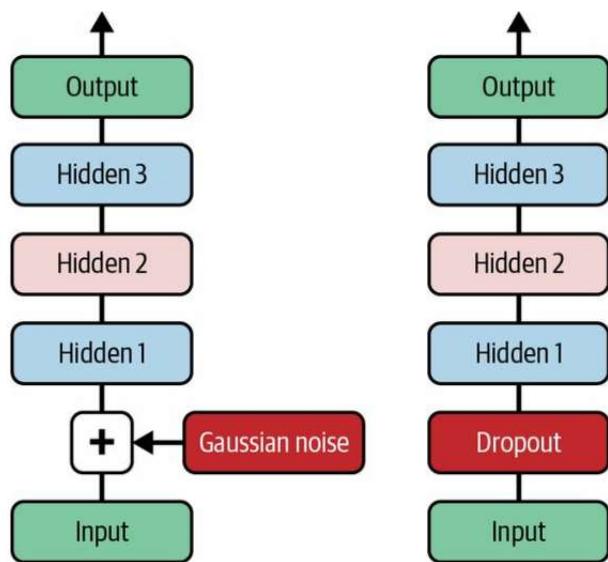


Figure 17-8. Denoising autoencoders, with Gaussian noise (left) or dropout (right)

Figure 17-9 shows a few noisy images (with half the pixels turned off), and the images reconstructed by the dropout-based denoising autoencoder. Notice how the autoencoder guesses details that are actually not in the input, such as the top of the white shirt (bottom row, fourth image). As you can see, not only can denoising autoencoders be used for data visualization or unsupervised pretraining, like the other autoencoders we've discussed so far, but they can also be used quite simply and efficiently to remove noise from images.



Figure 17-9. Noisy images (top) and their reconstructions (bottom)

Sparse Autoencoders

Another kind of constraint that often leads to good feature extraction is *sparsity*: by adding an appropriate term to the cost function, the autoencoder is pushed to reduce the number of active neurons in the coding layer. For example, it may be pushed to have on average only 5% significantly active neurons in the coding layer. This forces the autoencoder to represent each input as a combination of a small number of activations. As a result, each neuron in the coding layer typically ends up representing a useful feature (if you could speak only a few words per month, you would probably try to make them worth listening to).

A simple approach is to use the sigmoid activation function in the coding layer (to constrain the codings to values between 0 and 1), use a large coding layer (e.g., with 300 units), and add some ℓ_1 regularization to the coding layer's activations. The decoder is just a regular decoder:

```
sparse_l1_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(300, activation="sigmoid"),
    tf.keras.layers.ActivityRegularization(l1=1e-4)
])
sparse_l1_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
])
sparse_l1_ae = tf.keras.Sequential([sparse_l1_encoder, sparse_l1_decoder])
```

This ActivityRegularization layer just returns its inputs, but as a side effect it adds a training loss equal to the sum of the absolute values of its inputs. This only affects training. Equivalently, you could remove the ActivityRegularization layer and set

`activity_regularizer=tf.keras.regularizers.l1(1e-4)` in the previous layer. This penalty will encourage the neural network to produce codings close to 0, but since it will also be penalized if it does not reconstruct the inputs correctly, it

will have to output at least a few nonzero values. Using the ℓ_1 norm rather than the ℓ_2 norm will push the neural network to preserve the most important codings while eliminating the ones that are not needed for the input image (rather than just reducing all codings).

Another approach, which often yields better results, is to measure the actual sparsity of the coding layer at each training iteration, and penalize the model when the measured sparsity differs from a target sparsity. We do so by computing the average activation of each neuron in the coding layer, over the whole training batch. The batch size must not be too small, or else the mean will not be accurate.

Once we have the mean activation per neuron, we want to penalize the neurons that are too active, or not active enough, by adding a *sparsity loss* to the cost function. For example, if we measure that a neuron has an average activation of 0.3, but the target sparsity is 0.1, it must be penalized to activate less. One approach could be simply adding the squared error $(0.3 - 0.1)^2$ to the cost function, but in practice a better approach is to use the Kullback–Leibler (KL) divergence (briefly discussed in [Chapter 4](#)), which has much stronger gradients than the mean squared error, as you can see in [Figure 17-10](#).

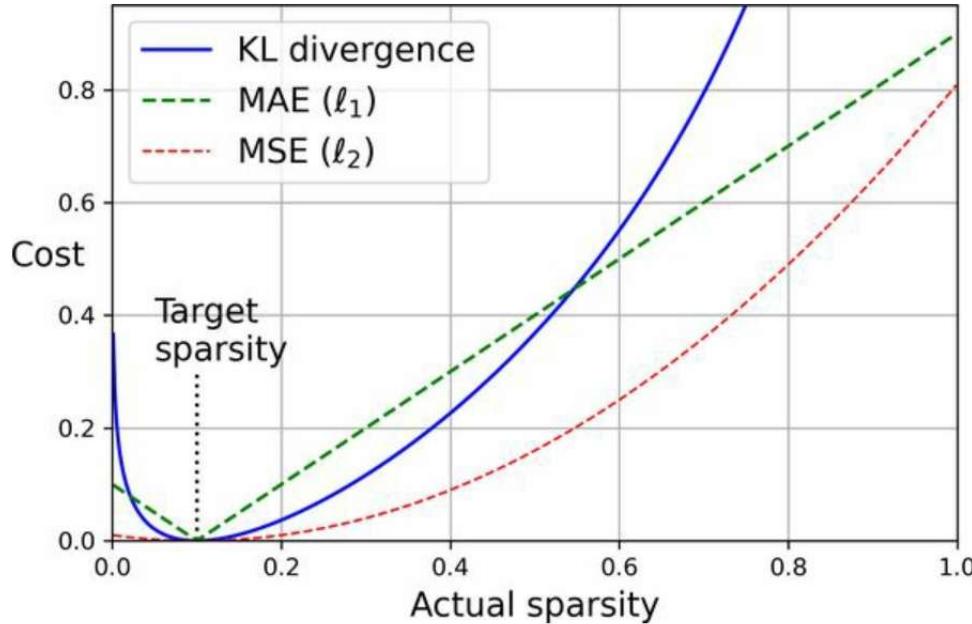


Figure 17-10. Sparsity loss

Given two discrete probability distributions P and Q , the KL divergence between these distributions, noted $D_{\text{KL}}(P \parallel Q)$, can be computed using [Equation 17-1](#).

[Equation 17-1. Kullback–Leibler divergence](#)

$$D_{\text{KL}}(P \parallel Q) = \sum_i P(i) \log P(i) Q(i)$$

In our case, we want to measure the divergence between the target probability p that a neuron in the coding layer will activate and the actual probability q , estimated by measuring the mean activation over the training batch. So, the KL divergence simplifies to [Equation 17-2](#).

[Equation 17-2. KL divergence between the target sparsity \$p\$ and the actual sparsity \$q\$](#)

$$D_{\text{KL}}(p \parallel q) = p \log p/q + (1-p) \log (1-p)/(1-q)$$

Once we have computed the sparsity loss for each neuron in the coding layer, we sum up these losses and add the result to the cost function. In order to control the relative importance of the sparsity loss and the reconstruction

loss, we can multiply the sparsity loss by a sparsity weight hyperparameter. If this weight is too high, the model will stick closely to the target sparsity, but it may not reconstruct the inputs properly, making the model useless. Conversely, if it is too low, the model will mostly ignore the sparsity objective and will not learn any interesting features.

We now have all we need to implement a sparse autoencoder based on the KL divergence. First, let's create a custom regularizer to apply KL divergence regularization:

```
kl_divergence = tf.keras.losses.kullback_leibler_divergence

class KLDivergenceRegularizer(tf.keras.regularizers.Regularizer):
    def __init__(self, weight, target):
        self.weight = weight
        self.target = target

    def __call__(self, inputs):
        mean_activities = tf.reduce_mean(inputs, axis=0)
        return self.weight * (
            kl_divergence(self.target, mean_activities) +
            kl_divergence(1. - self.target, 1. - mean_activities))
```

Now we can build the sparse autoencoder, using the KLDivergenceRegularizer for the coding layer's activations:

```
kld_reg = KLDivergenceRegularizer(weight=5e-3, target=0.1)
sparse_kl_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(300, activation="sigmoid",
                         activity_regularizer=kld_reg)
])
sparse_kl_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
])
sparse_kl_ae = tf.keras.Sequential([sparse_kl_encoder, sparse_kl_decoder])
```

After training this sparse autoencoder on Fashion MNIST, the coding layer will have roughly 10% sparsity.

Variational Autoencoders

An important category of autoencoders was introduced in 2013 by [Diederik Kingma and Max Welling](#)⁶ and quickly became one of the most popular variants: *variational autoencoders* (VAEs).

VAEs are quite different from all the autoencoders we have discussed so far, in these particular ways:

- They are *probabilistic autoencoders*, meaning that their outputs are partly determined by chance, even after training (as opposed to denoising autoencoders, which use randomness only during training).
- Most importantly, they are *generative autoencoders*, meaning that they can generate new instances that look like they were sampled from the training set.

Both these properties make VAEs rather similar to RBMs, but they are easier to train, and the sampling process is much faster (with RBMs you need to wait for the network to stabilize into a “thermal equilibrium” before you can sample a new instance). As their name suggests, variational autoencoders perform variational Bayesian inference, which is an efficient way of carrying out approximate Bayesian inference. Recall that Bayesian inference means updating a probability distribution based on new data, using equations derived from Bayes’ theorem. The original distribution is called the *prior*, while the updated distribution is called the *posterior*. In our case, we want to find a good approximation of the data distribution. Once we have that, we can sample from it.

Let’s take a look at how VAEs work. [Figure 17-11](#) (left) shows a variational autoencoder. You can recognize the basic structure of all autoencoders, with an encoder followed by a decoder (in this example, they both have two hidden layers), but there is a twist: instead of directly producing a coding for a given input, the encoder produces a *mean coding* μ and a standard deviation σ . The actual coding is then sampled randomly from a Gaussian distribution

with mean μ and standard deviation σ . After that the decoder decodes the sampled coding normally. The right part of the diagram shows a training instance going through this autoencoder. First, the encoder produces μ and σ , then a coding is sampled randomly (notice that it is not exactly located at μ), and finally this coding is decoded; the final output resembles the training instance.

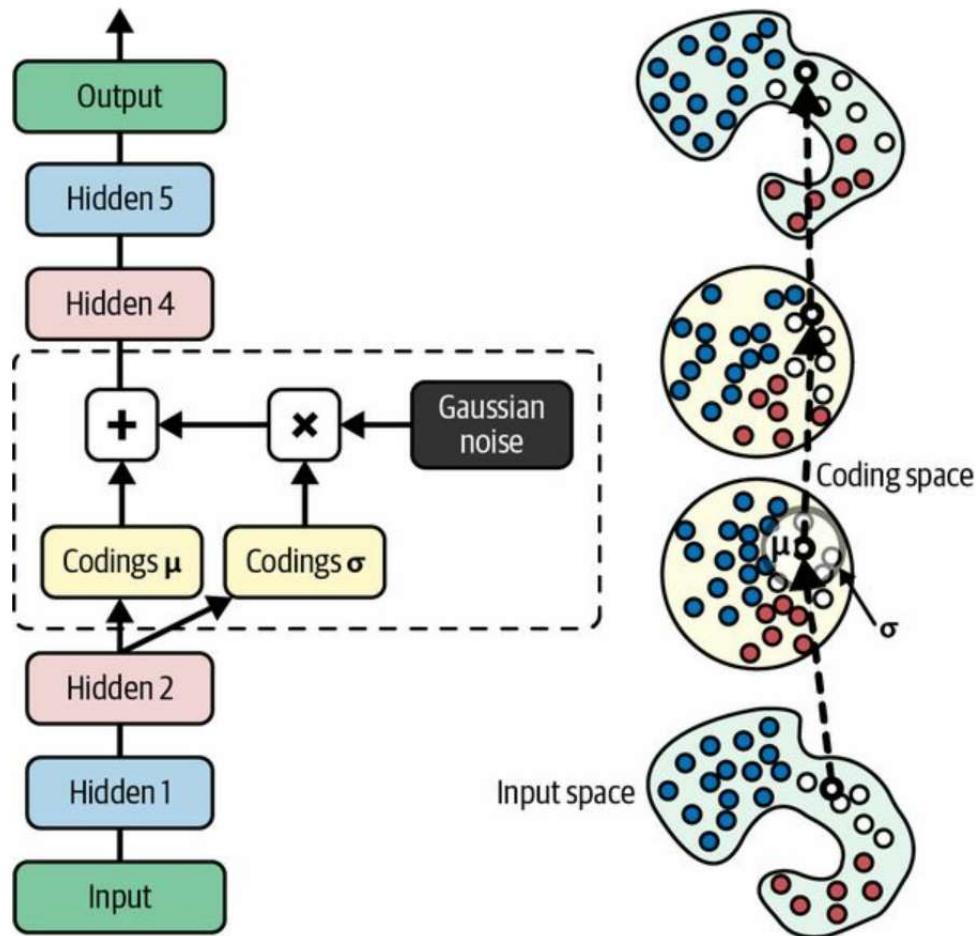


Figure 17-11. A variational autoencoder (left) and an instance going through it (right)

As you can see in the diagram, although the inputs may have a very convoluted distribution, a variational autoencoder tends to produce codings that look as though they were sampled from a simple Gaussian

distribution: ⁷ during training, the cost function (discussed next) pushes the codings to gradually migrate within the coding space (also called the *latent space*) to end up looking like a cloud of Gaussian points. One great consequence is that after training a variational autoencoder, you can very easily generate a new instance: just sample a random coding from the Gaussian distribution, decode it, and voilà!

Now, let's look at the cost function. It is composed of two parts. The first is the usual reconstruction loss that pushes the autoencoder to reproduce its inputs. We can use the MSE for this, as we did earlier. The second is the *latent loss* that pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution: it is the KL divergence between the target distribution (i.e., the Gaussian distribution) and the actual distribution of the codings. The math is a bit more complex than with the sparse autoencoder, in particular because of the Gaussian noise, which limits the amount of information that can be transmitted to the coding layer. This pushes the autoencoder to learn useful features. Luckily, the equations simplify, so the latent loss can be computed using [Equation 17-3.](#) ⁸

Equation 17-3. Variational autoencoder's latent loss

$$L = -\frac{1}{2} \sum_{i=1}^n (\ln(\sigma_i^2) - \sigma_i^2 - \mu_i^2)$$

In this equation, L is the latent loss, n is the codings' dimensionality, and μ_i and σ_i are the mean and standard deviation of the i^{th} component of the codings. The vectors μ and σ (which contain all the μ_i and σ_i) are output by the encoder, as shown in [Figure 17-11](#) (left).

A common tweak to the variational autoencoder's architecture is to make the encoder output $y = \log(\sigma^2)$ rather than σ . The latent loss can then be computed as shown in [Equation 17-4](#). This approach is more numerically stable and speeds up training.

Equation 17-4. Variational autoencoder's latent loss, rewritten using $y = \log(\sigma^2)$

$$L = -\frac{1}{2} \sum_{i=1}^n (\ln(y_i) - y_i - \mu_i^2)$$

Let's start building a variational autoencoder for Fashion MNIST (as shown in [Figure 17-11](#), but using the γ tweak). First, we will need a custom layer to sample the codings, given μ and γ :

```
class Sampling(tf.keras.layers.Layer):
    def call(self, inputs):
        mean, log_var = inputs
        return tf.random.normal(tf.shape(log_var)) * tf.exp(log_var / 2) + mean
```

This Sampling layer takes two inputs: mean (μ) and log_var (γ). It uses the function `tf.random.normal()` to sample a random vector (of the same shape as γ) from the Gaussian distribution, with mean 0 and standard deviation 1. Then it multiplies it by $\exp(\gamma / 2)$ (which is equal to σ , as you can verify mathematically), and finally it adds μ and returns the result. This samples a codings vector from the Gaussian distribution with mean μ and standard deviation σ .

Next, we can create the encoder, using the functional API because the model is not entirely sequential:

```
codings_size = 10

inputs = tf.keras.layers.Input(shape=[28, 28])
Z = tf.keras.layers.Flatten()(inputs)
Z = tf.keras.layers.Dense(150, activation="relu")(Z)
Z = tf.keras.layers.Dense(100, activation="relu")(Z)
codings_mean = tf.keras.layers.Dense(codings_size)(Z) # μ
codings_log_var = tf.keras.layers.Dense(codings_size)(Z) # γ
codings = Sampling()([codings_mean, codings_log_var])
variational_encoder = tf.keras.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])
```

Note that the Dense layers that output `codings_mean` (μ) and `codings_log_var` (γ) have the same inputs (i.e., the outputs of the second Dense layer). We then pass both `codings_mean` and `codings_log_var` to the Sampling layer. Finally, the `variational_encoder` model has three outputs. Only the codings are required, but we add `codings_mean` and `codings_log_var` as well, in case we want to inspect their values. Now let's build the decoder:

```

decoder_inputs = tf.keras.layers.Input(shape=[codings_size])
x = tf.keras.layers.Dense(100, activation="relu")(decoder_inputs)
x = tf.keras.layers.Dense(150, activation="relu")(x)
x = tf.keras.layers.Dense(28 * 28)(x)
outputs = tf.keras.layers.Reshape([28, 28])(x)
variational_decoder = tf.keras.Model(inputs=[decoder_inputs], outputs=[outputs])

```

For this decoder, we could have used the sequential API instead of the functional API, since it is really just a simple stack of layers, virtually identical to many of the decoders we have built so far. Finally, let's build the variational autoencoder model:

```

_, codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
variational_ae = tf.keras.Model(inputs=[inputs], outputs=[reconstructions])

```

We ignore the first two outputs of the encoder (we only want to feed the codings to the decoder). Lastly, we must add the latent loss and the reconstruction loss:

```

latent_loss = -0.5 * tf.reduce_sum(
    1 + codings_log_var - tf.exp(codings_log_var) - tf.square(codings_mean),
    axis=-1)
variational_ae.add_loss(tf.reduce_mean(latent_loss) / 784.)

```

We first apply [Equation 17-4](#) to compute the latent loss for each instance in the batch, summing over the last axis. Then we compute the mean loss over all the instances in the batch, and we divide the result by 784 to ensure it has the appropriate scale compared to the reconstruction loss. Indeed, the variational autoencoder's reconstruction loss is supposed to be the sum of the pixel reconstruction errors, but when Keras computes the "mse" loss it computes the mean over all 784 pixels, rather than the sum. So, the reconstruction loss is 784 times smaller than we need it to be. We could define a custom loss to compute the sum rather than the mean, but it is simpler to divide the latent loss by 784 (the final loss will be 784 times smaller than it should be, but this just means that we should use a larger learning rate).

And finally, we can compile and fit the autoencoder!

Generating Fashion MNIST Images

Now let's use this variational autoencoder to generate images that look like fashion items. All we need to do is sample random codings from a Gaussian distribution and decode them:

```
codings = tf.random.normal(shape=[3 * 7, codings_size])
images = variational_decoder(codings).numpy()
```

Figure 17-12 shows the 12 generated images.

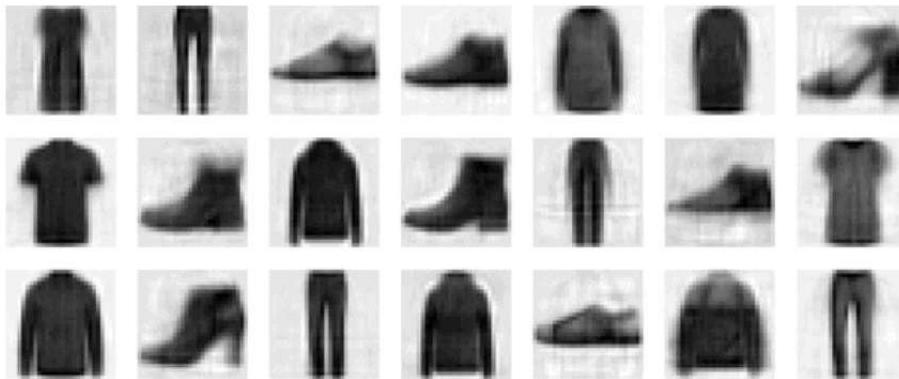


Figure 17-12. Fashion MNIST images generated by the variational autoencoder

The majority of these images look fairly convincing, if a bit too fuzzy. The rest are not great, but don't be too harsh on the autoencoder—it only had a few minutes to learn!

Variational autoencoders make it possible to perform *semantic interpolation*: instead of interpolating between two images at the pixel level, which would look as if the two images were just overlaid, we can interpolate at the codings level. For example, let's take a few codings along an arbitrary line in latent space and decode them. We get a sequence of images that gradually go from pants to sweaters (see Figure 17-13):

```
codings = np.zeros([7, codings_size])
codings[:, 3] = np.linspace(-0.8, 0.8, 7) # axis 3 looks best in this case
```

```
images = variational_decoder(codings).numpy()
```

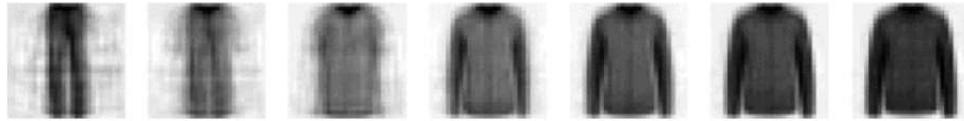


Figure 17-13. Semantic interpolation

Let's now turn our attention to GANs: they are harder to train, but when you manage to get them to work, they produce pretty amazing images.

Generative Adversarial Networks

Generative adversarial networks were proposed in a [2014 paper⁹](#) by Ian Goodfellow et al., and although the idea got researchers excited almost instantly, it took a few years to overcome some of the difficulties of training GANs. Like many great ideas, it seems simple in hindsight: make neural networks compete against each other in the hope that this competition will push them to excel. As shown in [Figure 17-14](#), a GAN is composed of two neural networks:

Generator

Takes a random distribution as input (typically Gaussian) and outputs some data—typically, an image. You can think of the random inputs as the latent representations (i.e., codings) of the image to be generated. So, as you can see, the generator offers the same functionality as a decoder in a variational autoencoder, and it can be used in the same way to generate new images: just feed it some Gaussian noise, and it outputs a brand-new image. However, it is trained very differently, as you will soon see.

Discriminator

Takes either a fake image from the generator or a real image from the training set as input, and must guess whether the input image is fake or real.

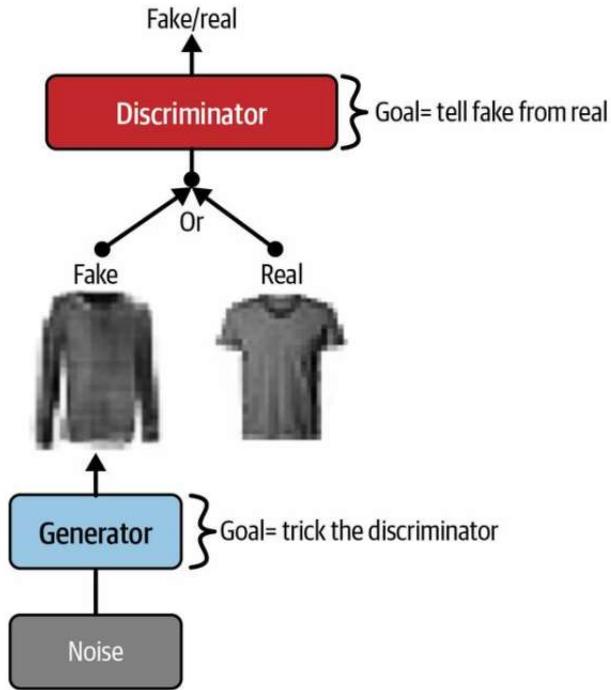


Figure 17-14. A generative adversarial network

During training, the generator and the discriminator have opposite goals: the discriminator tries to tell fake images from real images, while the generator tries to produce images that look real enough to trick the discriminator.

Because the GAN is composed of two networks with different objectives, it cannot be trained like a regular neural network. Each training iteration is divided into two phases:

- In the first phase, we train the discriminator. A batch of real images is sampled from the training set and is completed with an equal number of fake images produced by the generator. The labels are set to 0 for fake images and 1 for real images, and the discriminator is trained on this labeled batch for one step, using the binary cross-entropy loss. Importantly, backpropagation only optimizes the weights of the discriminator during this phase.
- In the second phase, we train the generator. We first use it to produce

another batch of fake images, and once again the discriminator is used to tell whether the images are fake or real. This time we do not add real images in the batch, and all the labels are set to 1 (real): in other words, we want the generator to produce images that the discriminator will (wrongly) believe to be real! Crucially, the weights of the discriminator are frozen during this step, so backpropagation only affects the weights of the generator.

NOTE

The generator never actually sees any real images, yet it gradually learns to produce convincing fake images! All it gets is the gradients flowing back through the discriminator. Fortunately, the better the discriminator gets, the more information about the real images is contained in these secondhand gradients, so the generator can make significant progress.

Let's go ahead and build a simple GAN for Fashion MNIST.

First, we need to build the generator and the discriminator. The generator is similar to an autoencoder's decoder, and the discriminator is a regular binary classifier: it takes an image as input and ends with a Dense layer containing a single unit and using the sigmoid activation function. For the second phase of each training iteration, we also need the full GAN model containing the generator followed by the discriminator:

```
codings_size = 30

Dense = tf.keras.layers.Dense
generator = tf.keras.Sequential([
    Dense(100, activation="relu", kernel_initializer="he_normal"),
    Dense(150, activation="relu", kernel_initializer="he_normal"),
    Dense(28 * 28, activation="sigmoid"),
    tf.keras.layers.Reshape([28, 28])
])
discriminator = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    Dense(150, activation="relu", kernel_initializer="he_normal"),
    Dense(100, activation="relu", kernel_initializer="he_normal"),
    Dense(1, activation="sigmoid")
```

```
])  
gan = tf.keras.Sequential([generator, discriminator])
```

Next, we need to compile these models. As the discriminator is a binary classifier, we can naturally use the binary cross-entropy loss. The gan model is also a binary classifier, so it can use the binary cross-entropy loss as well. However, the generator will only be trained through the gan model, so we do not need to compile it at all. Importantly, the discriminator should not be trained during the second phase, so we make it non-trainable before compiling the gan model:

```
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")  
discriminator.trainable = False  
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

NOTE

The trainable attribute is taken into account by Keras only when compiling a model, so after running this code, the discriminator *is* trainable if we call its fit() method or its train_on_batch() method (which we will be using), while it is *not* trainable when we call these methods on the gan model.

Since the training loop is unusual, we cannot use the regular fit() method. Instead, we will write a custom training loop. For this, we first need to create a Dataset to iterate through the images:

```
batch_size = 32  
dataset = tf.data.Dataset.from_tensor_slices(X_train).shuffle(buffer_size=1000)  
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
```

We are now ready to write the training loop. Let's wrap it in a train_gan() function:

```
def train_gan(gan, dataset, batch_size, codings_size, n_epochs):  
    generator, discriminator = gan.layers  
    for epoch in range(n_epochs):  
        for X_batch in dataset:  
            # phase 1 - training the discriminator
```

```

noise = tf.random.normal(shape=[batch_size, codings_size])
generated_images = generator(noise)
X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)
y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
discriminator.train_on_batch(X_fake_and_real, y1)
# phase 2 - training the generator
noise = tf.random.normal(shape=[batch_size, codings_size])
y2 = tf.constant([[1.]] * batch_size)
gan.train_on_batch(noise, y2)

train_gan(gan, dataset, batch_size, codings_size, n_epochs=50)

```

As discussed earlier, you can see the two phases at each iteration:

- In phase one we feed Gaussian noise to the generator to produce fake images, and we complete this batch by concatenating an equal number of real images. The targets y_1 are set to 0 for fake images and 1 for real images. Then we train the discriminator on this batch. Remember that the discriminator is trainable in this phase, but we are not touching the generator.
- In phase two, we feed the GAN some Gaussian noise. Its generator will start by producing fake images, then the discriminator will try to guess whether these images are fake or real. In this phase, we are trying to improve the generator, which means that we want the discriminator to fail: this is why the targets y_2 are all set to 1, although the images are fake. In this phase, the discriminator is *not* trainable, so the only part of the gan model that will improve is the generator.

That's it! After training, you can randomly sample some codings from a Gaussian distribution, and feed them to the generator to produce new images:

```

codings = tf.random.normal(shape=[batch_size, codings_size])
generated_images = generator.predict(codings)

```

If you display the generated images (see [Figure 17-15](#)), you will see that at the end of the first epoch, they already start to look like (very noisy) Fashion MNIST images.



Figure 17-15. Images generated by the GAN after one epoch of training

Unfortunately, the images never really get much better than that, and you may even find epochs where the GAN seems to be forgetting what it learned. Why is that? Well, it turns out that training a GAN can be challenging. Let's see why.

The Difficulties of Training GANs

During training, the generator and the discriminator constantly try to outsmart each other, in a zero-sum game. As training advances, the game may end up in a state that game theorists call a *Nash equilibrium*, named after the mathematician John Nash: this is when no player would be better off changing their own strategy, assuming the other players do not change theirs. For example, a Nash equilibrium is reached when everyone drives on the left side of the road: no driver would be better off being the only one to switch sides. Of course, there is a second possible Nash equilibrium: when everyone drives on the *right* side of the road. Different initial states and dynamics may lead to one equilibrium or the other. In this example, there is a single optimal strategy once an equilibrium is reached (i.e., driving on the same side as everyone else), but a Nash equilibrium can involve multiple competing strategies (e.g., a predator chases its prey, the prey tries to escape, and neither would be better off changing their strategy).

So how does this apply to GANs? Well, the authors of the GAN paper demonstrated that a GAN can only reach a single Nash equilibrium: that's when the generator produces perfectly realistic images, and the discriminator is forced to guess (50% real, 50% fake). This fact is very encouraging: it would seem that you just need to train the GAN for long enough, and it will eventually reach this equilibrium, giving you a perfect generator. Unfortunately, it's not that simple: nothing guarantees that the equilibrium will ever be reached.

The biggest difficulty is called *mode collapse*: this is when the generator's outputs gradually become less diverse. How can this happen? Suppose that the generator gets better at producing convincing shoes than any other class. It will fool the discriminator a bit more with shoes, and this will encourage it to produce even more images of shoes. Gradually, it will forget how to produce anything else. Meanwhile, the only fake images that the discriminator will see will be shoes, so it will also forget how to discriminate fake images of other classes. Eventually, when the discriminator manages to discriminate the fake shoes from the real ones, the generator will be forced to

move to another class. It may then become good at shirts, forgetting about shoes, and the discriminator will follow. The GAN may gradually cycle across a few classes, never really becoming very good at any of them.

Moreover, because the generator and the discriminator are constantly pushing against each other, their parameters may end up oscillating and becoming unstable. Training may begin properly, then suddenly diverge for no apparent reason, due to these instabilities. And since many factors affect these complex dynamics, GANs are very sensitive to the hyperparameters: you may have to spend a lot of effort fine-tuning them. In fact, that's why I used RMSProp rather than Nadam when compiling the models: when using Nadam, I ran into a severe mode collapse.

These problems have kept researchers very busy since 2014: many papers have been published on this topic, some proposing new cost functions ¹⁰ (though a [2018 paper](#)¹¹ by Google researchers questions their efficiency) or techniques to stabilize training or to avoid the mode collapse issue. For example, a popular technique called *experience replay* consists of storing the images produced by the generator at each iteration in a replay buffer (gradually dropping older generated images) and training the discriminator using real images plus fake images drawn from this buffer (rather than just fake images produced by the current generator). This reduces the chances that the discriminator will overfit the latest generator's outputs. Another common technique is called *mini-batch discrimination*: it measures how similar images are across the batch and provides this statistic to the discriminator, so it can easily reject a whole batch of fake images that lack diversity. This encourages the generator to produce a greater variety of images, reducing the chance of mode collapse. Other papers simply propose specific architectures that happen to perform well.

In short, this is still a very active field of research, and the dynamics of GANs are still not perfectly understood. But the good news is that great progress has been made, and some of the results are truly astounding! So let's look at some of the most successful architectures, starting with deep convolutional GANs, which were the state of the art just a few years ago. Then we will look at two more recent (and more complex) architectures.

Deep Convolutional GANs

The authors of the original GAN paper experimented with convolutional layers, but only tried to generate small images. Soon after, many researchers tried to build GANs based on deeper convolutional nets for larger images. This proved to be tricky, as training was very unstable, but Alec Radford et al. finally succeeded in late 2015, after experimenting with many different architectures and hyperparameters. They called their architecture *deep convolutional GANs* (DCGANs). ¹² Here are the main guidelines they proposed for building stable convolutional GANs:

- Replace any pooling layers with strided convolutions (in the discriminator) and transposed convolutions (in the generator).
- Use batch normalization in both the generator and the discriminator, except in the generator's output layer and the discriminator's input layer.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in the generator for all layers except the output layer, which should use tanh.
- Use leaky ReLU activation in the discriminator for all layers.

These guidelines will work in many cases, but not always, so you may still need to experiment with different hyperparameters. In fact, just changing the random seed and training the exact same model again will sometimes work. Here is a small DCGAN that works reasonably well with Fashion MNIST:

```
codings_size = 100

generator = tf.keras.Sequential([
    tf.keras.layers.Dense(7 * 7 * 128),
    tf.keras.layers.Reshape([7, 7, 128]),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2DTranspose(64, kernel_size=5, strides=2,
        padding="same", activation="relu"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2,
```

```

        padding="same", activation="tanh"),
])
discriminator = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="same",
        activation=tf.keras.layers.LeakyReLU(0.2)),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="same",
        activation=tf.keras.layers.LeakyReLU(0.2)),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
gan = tf.keras.Sequential([generator, discriminator])

```

The generator takes codings of size 100, projects them to 6,272 dimensions ($7 \times 7 \times 128$), and reshapes the result to get a $7 \times 7 \times 128$ tensor. This tensor is batch normalized and fed to a transposed convolutional layer with a stride of 2, which upsamples it from 7×7 to 14×14 and reduces its depth from 128 to 64. The result is batch normalized again and fed to another transposed convolutional layer with a stride of 2, which upsamples it from 14×14 to 28×28 and reduces the depth from 64 to 1. This layer uses the tanh activation function, so the outputs will range from -1 to 1. For this reason, before training the GAN, we need to rescale the training set to that same range. We also need to reshape it to add the channel dimension:

```
X_train_dcgan = X_train.reshape(-1, 28, 28, 1) * 2. - 1. # reshape and rescale
```

The discriminator looks much like a regular CNN for binary classification, except instead of using max pooling layers to downsample the image, we use strided convolutions (strides=2). Note that we use the leaky ReLU activation function. Overall, we respected the DCGAN guidelines, except we replaced the BatchNormalization layers in the discriminator with Dropout layers; otherwise, training was unstable in this case. Feel free to tweak this architecture: you will see how sensitive it is to the hyperparameters, especially the relative learning rates of the two networks.

Lastly, to build the dataset and then compile and train this model, we can use the same code as earlier. After 50 epochs of training, the generator produces images like those shown in [Figure 17-16](#). It's still not perfect, but many of

these images are pretty convincing.



Figure 17-16. Images generated by the DCGAN after 50 epochs of training

If you scale up this architecture and train it on a large dataset of faces, you can get fairly realistic images. In fact, DCGANs can learn quite meaningful latent representations, as you can see in [Figure 17-17](#): many images were generated, and nine of them were picked manually (top left), including three representing men with glasses, three men without glasses, and three women without glasses. For each of these categories, the codings that were used to generate the images were averaged, and an image was generated based on the resulting mean codings (lower left). In short, each of the three lower-left images represents the mean of the three images located above it. But this is not a simple mean computed at the pixel level (this would result in three overlapping faces), it is a mean computed in the latent space, so the images still look like normal faces. Amazingly, if you compute men with glasses, minus men without glasses, plus women without glasses—where each term corresponds to one of the mean codings—and you generate the image that corresponds to this coding, you get the image at the center of the 3×3 grid of faces on the right: a woman with glasses! The eight other images around it were generated based on the same vector plus a bit of noise, to illustrate the semantic interpolation capabilities of DCGANs. Being able to do arithmetic on faces feels like science fiction!

DCGANs aren't perfect, though. For example, when you try to generate very large images using DCGANs, you often end up with locally convincing features but overall inconsistencies, such as shirts with one sleeve much longer than the other, different earrings, or eyes looking in opposite directions. How can you fix this?

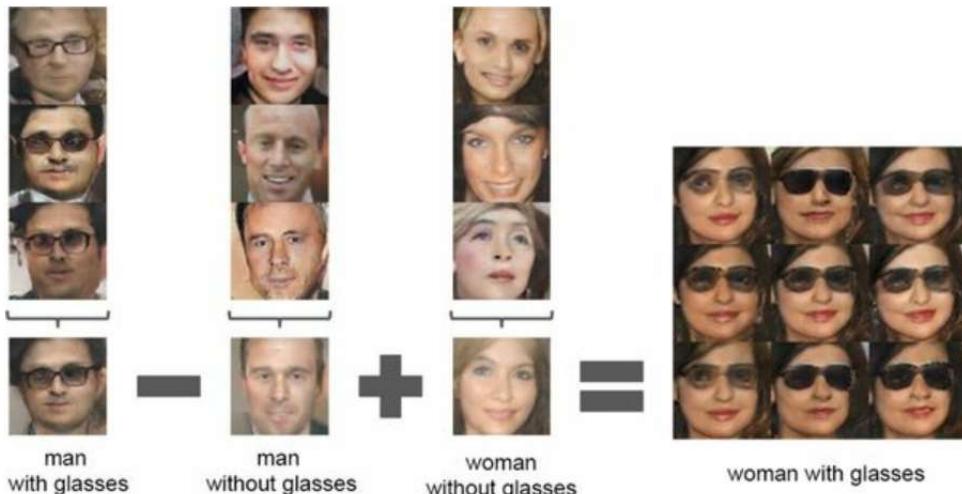


Figure 17-17. Vector arithmetic for visual concepts (part of figure 7 from the DCGAN paper) ¹³

TIP

If you add each image's class as an extra input to both the generator and the discriminator, they will both learn what each class looks like, and thus you will be able to control the class of each image produced by the generator. This is called a *conditional GAN*(CGAN). ¹⁴

Progressive Growing of GANs

In a [2018 paper](#), ¹⁵ Nvidia researchers Tero Keras et al. proposed an important technique: they suggested generating small images at the beginning of training, then gradually adding convolutional layers to both the generator and the discriminator to produce larger and larger images (4×4 , 8×8 , 16×16 , ..., 512×512 , $1,024 \times 1,024$). This approach resembles greedy layer-wise training of stacked autoencoders. The extra layers get added at the end of the generator and at the beginning of the discriminator, and previously trained layers remain trainable.

For example, when growing the generator's outputs from 4×4 to 8×8 (see [Figure 17-18](#)), an upsampling layer (using nearest neighbor filtering) is added to the existing convolutional layer ("Conv 1") to produce 8×8 feature maps. These are fed to the new convolutional layer ("Conv 2"), which in turn feeds into a new output convolutional layer. To avoid breaking the trained weights of Conv 1, we gradually fade in the two new convolutional layers (represented with dashed lines in [Figure 17-18](#)) and fade out the original output layer. The final outputs are a weighted sum of the new outputs (with weight α) and the original outputs (with weight $1 - \alpha$), slowly increasing α from 0 to 1. A similar fade-in/fade-out technique is used when a new convolutional layer is added to the discriminator (followed by an average pooling layer for downsampling). Note that all convolutional layers use "same" padding and strides of 1, so they preserve the height and width of their inputs. This includes the original convolutional layer, so it now produces 8×8 outputs (since its inputs are now 8×8). Lastly, the output layers use kernel size 1. They just project their inputs down to the desired number of color channels (typically 3).

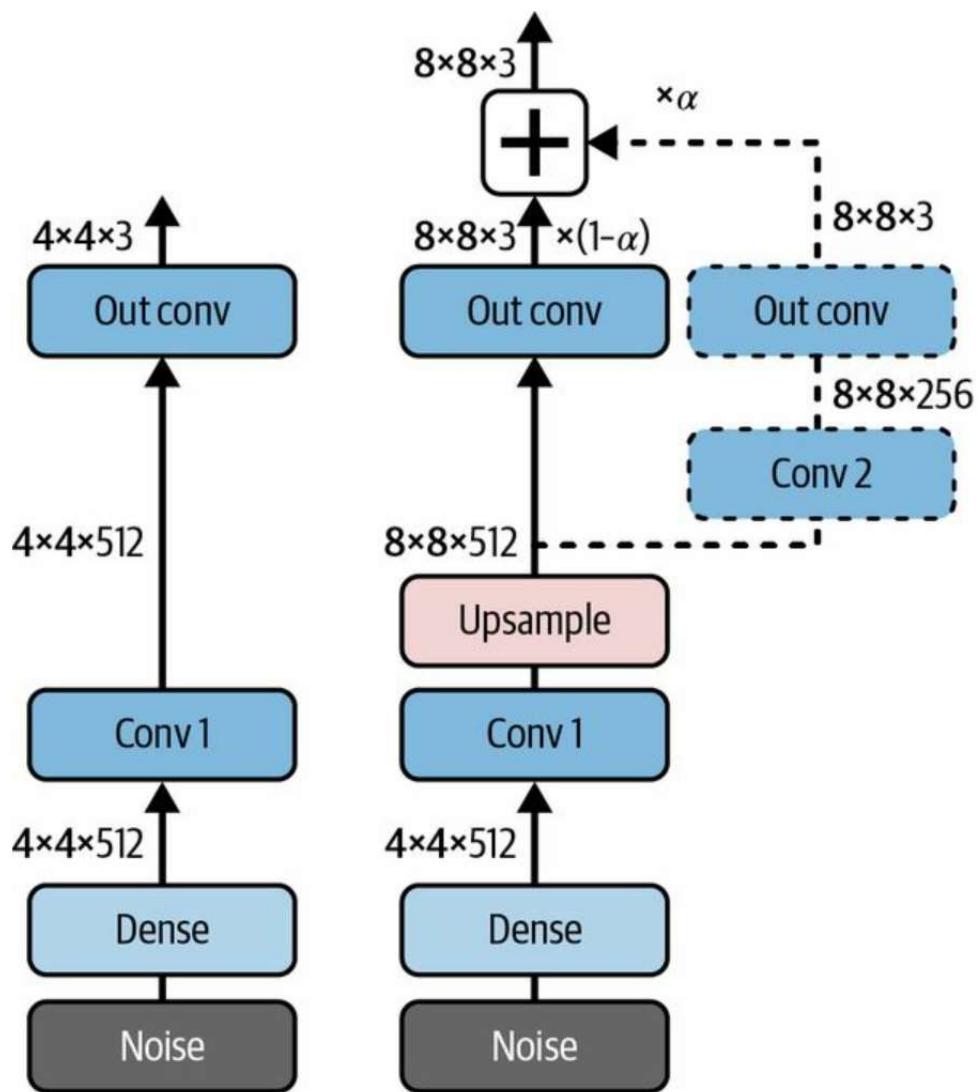


Figure 17-18. A progressively growing GAN: a GAN generator outputs 4×4 color images (left); we extend it to output 8×8 images (right)

The paper also introduced several other techniques aimed at increasing the diversity of the outputs (to avoid mode collapse) and making training more stable:

Mini-batch standard deviation layer

Added near the end of the discriminator. For each position in the inputs, it computes the standard deviation across all channels and all instances in the batch ($S = \text{tf.math.reduce_std(inputs, axis=[0, -1])}$). These standard deviations are then averaged across all points to get a single value ($v = \text{tf.reduce_mean}(S)$). Finally, an extra feature map is added to each instance in the batch and filled with the computed value ($(\text{tf.concat}([\text{inputs}, \text{tf.fill}([\text{batch_size}, \text{height}, \text{width}, 1], v)], \text{axis}=-1))$). How does this help? Well, if the generator produces images with little variety, then there will be a small standard deviation across feature maps in the discriminator. Thanks to this layer, the discriminator will have easy access to this statistic, making it less likely to be fooled by a generator that produces too little diversity. This will encourage the generator to produce more diverse outputs, reducing the risk of mode collapse.

Equalized learning rate

Initializes all weights using a Gaussian distribution with mean 0 and standard deviation 1 rather than using He initialization. However, the weights are scaled down at runtime (i.e., every time the layer is executed) by the same factor as in He initialization: they are divided by $2n_{\text{inputs}}$, where n_{inputs} is the number of inputs to the layer. The paper demonstrated that this technique significantly improved the GAN's performance when using RMSProp, Adam, or other adaptive gradient optimizers. Indeed, these optimizers normalize the gradient updates by their estimated standard deviation (see [Chapter 11](#)), so parameters that have a larger dynamic range ¹⁶ will take longer to train, while parameters with a small dynamic range may be updated too quickly, leading to instabilities. By rescaling the weights as part of the model itself rather than just rescaling them upon initialization, this approach ensures that the dynamic range is the same for all parameters throughout training, so they all learn at the same speed. This both speeds up and stabilizes training.

Pixelwise normalization layer

Added after each convolutional layer in the generator. It normalizes each activation based on all the activations in the same image and at the same

location, but across all channels (dividing by the square root of the mean squared activation). In TensorFlow code, this is inputs /
`tf.sqrt(tf.reduce_mean(tf.square(X), axis=-1, keepdims=True) + 1e-8)` (the smoothing term `1e-8` is needed to avoid division by zero). This technique avoids explosions in the activations due to excessive competition between the generator and the discriminator.

The combination of all these techniques allowed the authors to generate **extremely convincing high-definition images of faces**. But what exactly do we call “convincing”? Evaluation is one of the big challenges when working with GANs: although it is possible to automatically evaluate the diversity of the generated images, judging their quality is a much trickier and subjective task. One technique is to use human raters, but this is costly and time-consuming. So, the authors proposed to measure the similarity between the local image structure of the generated images and the training images, considering every scale. This idea led them to another groundbreaking innovation: StyleGANs.

StyleGANs

The state of the art in high-resolution image generation was advanced once again by the same Nvidia team in a [2018 paper¹⁷](#) that introduced the popular StyleGAN architecture. The authors used *style transfer* techniques in the generator to ensure that the generated images have the same local structure as the training images, at every scale, greatly improving the quality of the generated images. The discriminator and the loss function were not modified, only the generator. A StyleGAN generator is composed of two networks (see [Figure 17-19](#)):

Mapping network

An eight-layer MLP that maps the latent representations \mathbf{z} (i.e., the codings) to a vector \mathbf{w} . This vector is then sent through multiple *affine transformations* (i.e., Dense layers with no activation functions, represented by the “A” boxes in [Figure 17-19](#)), which produces multiple vectors. These vectors control the style of the generated image at different levels, from fine-grained texture (e.g., hair color) to high-level features (e.g., adult or child). In short, the mapping network maps the codings to multiple style vectors.

Synthesis network

Responsible for generating the images. It has a constant learned input (to be clear, this input will be constant *after* training, but *during* training it keeps getting tweaked by backpropagation). It processes this input through multiple convolutional and upsampling layers, as earlier, but there are two twists. First, some noise is added to the input and to all the outputs of the convolutional layers (before the activation function). Second, each noise layer is followed by an *adaptive instance normalization* (AdaIN) layer: it standardizes each feature map independently (by subtracting the feature map’s mean and dividing by its standard deviation), then it uses the style vector to determine the scale and offset of each feature map (the style vector contains one scale and

one bias term for each feature map).

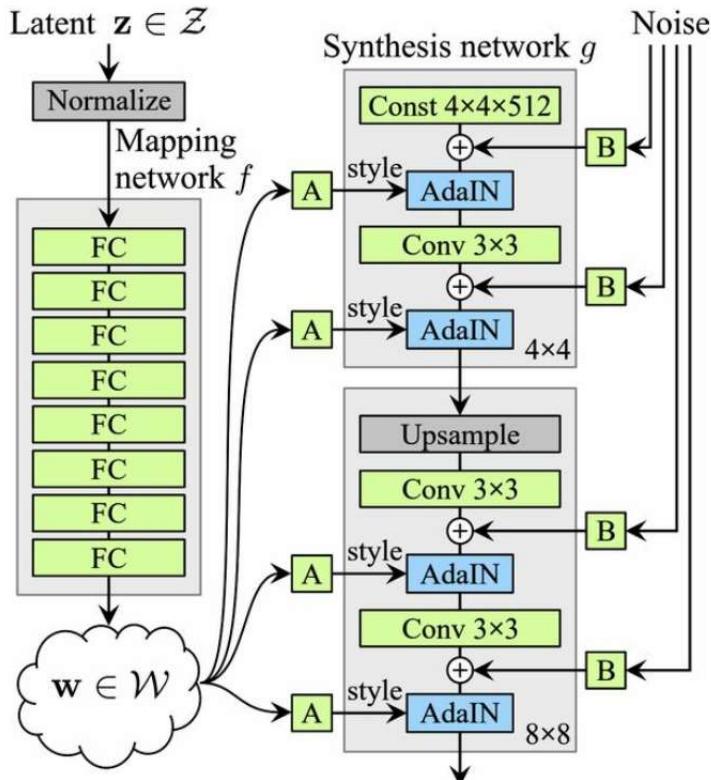


Figure 17-19. StyleGAN’s generator architecture (part of Figure 1 from the StyleGAN paper) ¹⁸

The idea of adding noise independently from the codings is very important. Some parts of an image are quite random, such as the exact position of each freckle or hair. In earlier GANs, this randomness had to either come from the codings or be some pseudorandom noise produced by the generator itself. If it came from the codings, it meant that the generator had to dedicate a significant portion of the codings’ representational power to storing noise, which this is quite wasteful. Moreover, the noise had to be able to flow through the network and reach the final layers of the generator: this seems like an unnecessary constraint that probably slowed down training. And finally, some visual artifacts may appear because the same noise was used at different levels. If instead the generator tried to produce its own

pseudorandom noise, this noise might not look very convincing, leading to more visual artifacts. Plus, part of the generator's weights would be dedicated to generating pseudorandom noise, which again seems wasteful. By adding extra noise inputs, all these issues are avoided; the GAN is able to use the provided noise to add the right amount of stochasticity to each part of the image.

The added noise is different for each level. Each noise input consists of a single feature map full of Gaussian noise, which is broadcast to all feature maps (of the given level) and scaled using learned per-feature scaling factors (this is represented by the “B” boxes in [Figure 17-19](#)) before it is added.

Finally, StyleGAN uses a technique called *mixing regularization* (or *style mixing*), where a percentage of the generated images are produced using two different codings. Specifically, the codings \mathbf{c}_1 and \mathbf{c}_2 are sent through the mapping network, giving two style vectors \mathbf{w}_1 and \mathbf{w}_2 . Then the synthesis network generates an image based on the styles \mathbf{w}_1 for the first levels and the styles \mathbf{w}_2 for the remaining levels. The cutoff level is picked randomly. This prevents the network from assuming that styles at adjacent levels are correlated, which in turn encourages locality in the GAN, meaning that each style vector only affects a limited number of traits in the generated image.

There is such a wide variety of GANs out there that it would require a whole book to cover them all. Hopefully this introduction has given you the main ideas, and most importantly the desire to learn more. Go ahead and implement your own GAN, and do not get discouraged if it has trouble learning at first: unfortunately, this is normal, and it will require quite a bit of patience to get it working, but the result is worth it. If you're struggling with an implementation detail, there are plenty of Keras or TensorFlow implementations that you can look at. In fact, if all you want is to get some amazing results quickly, then you can just use a pretrained model (e.g., there are pretrained StyleGAN models available for Keras).

Now that we've examined autoencoders and GANs, let's look at one last type of architecture: diffusion models.

Diffusion Models

The ideas behind diffusion models have been around for many years, but they were first formalized in their modern form in a [2015 paper¹⁹](#) by Jascha Sohl-Dickstein et al. from Stanford University and UC Berkeley. The authors applied tools from thermodynamics to model a diffusion process, similar to a drop of milk diffusing in a cup of tea. The core idea is to train a model to learn the reverse process: start from the completely mixed state, and gradually “unmix” the milk from the tea. Using this idea, they obtained promising results in image generation, but since GANs produced more convincing images back then, diffusion models did not get as much attention.

Then, in 2020, [Jonathan Ho et al.](#), also from UC Berkeley, managed to build a diffusion model capable of generating highly realistic images, which they called a *denoising diffusion probabilistic model* (DDPM). ²⁰ A few months later, a [2021 paper²¹](#) by OpenAI researchers Alex Nichol and Prafulla Dhariwal analyzed the DDPM architecture and proposed several improvements that allowed DDPMs to finally beat GANs: not only are DDPMs much easier to train than GANs, but the generated images are more diverse and of even higher quality. The main downside of DDPMs, as you will see, is that they take a very long time to generate images, compared to GANs or VAEs.

So how exactly does a DDPM work? Well, suppose you start with a picture of a cat (like the one you’ll see in [Figure 17-20](#)), noted \mathbf{x}_0 , and at each time step t you add a little bit of Gaussian noise to the image, with mean 0 and variance β_t . This noise is independent for each pixel: we call it *isotropic*. You first obtain the image \mathbf{x}_1 , then \mathbf{x}_2 , and so on, until the cat is completely hidden by the noise, impossible to see. The last time step is noted T . In the original DDPM paper, the authors used $T = 1,000$, and they scheduled the variance β_t in such a way that the cat signal fades linearly between time steps 0 and T . In the improved DDPM paper, T was bumped up to 4,000, and the variance schedule was tweaked to change more slowly at the beginning and at the end. In short, we’re gradually drowning the cat in noise: this is called the *forward*

process.

As we add more and more Gaussian noise in the forward process, the distribution of pixel values becomes more and more Gaussian. One important detail I left out is that the pixel values get rescaled slightly at each step, by a factor of $1 - \beta_t$. This ensures that the mean of the pixel values gradually approaches 0, since the scaling factor is a bit smaller than 1 (imagine repeatedly multiplying a number by 0.99). It also ensures that the variance will gradually converge to 1. This is because the standard deviation of the pixel values also gets scaled by $1 - \beta_t$, so the variance gets scaled by $1 - \beta_t^2$ (i.e., the square of the scaling factor). But the variance cannot shrink to 0 since we're adding Gaussian noise with variance β_t at each step. And since variances add up when you sum Gaussian distributions, you can see that the variance can only converge to $1 - \beta_t^2 + \beta_t^2 = 1$.

The forward diffusion process is summarized in [Equation 17-5](#). This equation won't teach you anything new about the forward process, but it's useful to understand this type of mathematical notation, as it's often used in ML papers. This equation defines the probability distribution q of \mathbf{x}_t given \mathbf{x}_{t-1} as a Gaussian distribution with mean \mathbf{x}_{t-1} times the scaling factor, and with a covariance matrix equal to $\beta_t \mathbf{I}$. This is the identity matrix \mathbf{I} multiplied by β_t , which means that the noise is isotropic with variance β_t .

Equation 17-5. Probability distribution q of the forward diffusion process

$$q(\mathbf{x}_t | \mathbf{x}_{t-1}) = N(\mathbf{x}_{t-1}, \beta_t \mathbf{I})$$

Interestingly, there's a shortcut for the forward process: it's possible to sample an image \mathbf{x}_t given \mathbf{x}_0 without having to first compute $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{t-1}$. Indeed, since the sum of multiple Gaussian distributions is also a Gaussian distribution, all the noise can be added in just one shot using [Equation 17-6](#). This is the equation we will be using, as it is much faster.

Equation 17-6. Shortcut for the forward diffusion process

$$q(\mathbf{x}_t | \mathbf{x}_0) = N(\mathbf{x}_0, (1 - \alpha_t^{-2}) \mathbf{I})$$

Our goal, of course, is not to drown cats in noise. On the contrary, we want to

create many new cats! We can do so by training a model that can perform the *reverse process*: going from \mathbf{x}_t to \mathbf{x}_{t-1} . We can then use it to remove a tiny bit of noise from an image, and repeat the operation many times until all the noise is gone. If we train the model on a dataset containing many cat images, then we can give it a picture entirely full of Gaussian noise, and the model will gradually make a brand new cat appear (see [Figure 17-20](#)).

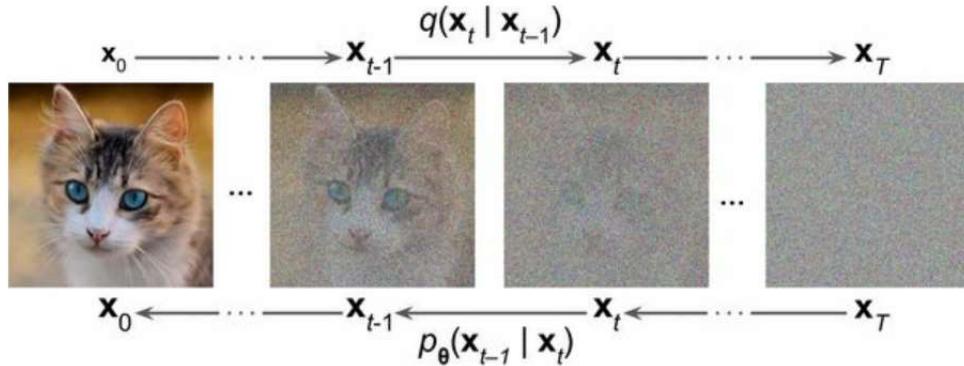


Figure 17-20. The forward process q and reverse process p

OK, so let's start coding! The first thing we need to do is to code the forward process. For this, we will first need to implement the variance schedule. How can we control how fast the cat disappears? Initially, 100% of the variance comes from the original cat image. Then at each time step t , the variance gets multiplied by $1 - \beta_t$, as explained earlier, and noise gets added. So, the part of the variance that comes from the initial distribution shrinks by a factor of $1 - \beta_t$ at each step. If we define $\alpha_t = 1 - \beta_t$, then after t time steps, the cat signal will have been multiplied by a factor of $\bar{\alpha}_t = \alpha_1 \times \alpha_2 \times \dots \times \alpha_t = \bar{\alpha}^{-t} = \prod_{i=1}^t \alpha_i$. It's this "cat signal" factor $\bar{\alpha}_t$ that we want to schedule so it shrinks down from 1 to 0 gradually between time steps 0 and T . In the improved DDPM paper, the authors schedule $\bar{\alpha}_t$ according to [Equation 17-7](#). This schedule is represented in [Figure 17-21](#).

Equation 17-7. Variance schedule equations for the forward diffusion process

$$\beta_t = 1 - \alpha_t \bar{\alpha}_t^{-1}, \text{ with } \bar{\alpha}_t = f(t)f(0) \text{ and } f(t) = \cos(t/T + s_1 + s_2 \cdot \pi/2)^2$$

In these equations:

- s is a tiny value which prevents β_t from being too small near $t = 0$. In the paper, the authors used $s = 0.008$.
- β_t is clipped to be no larger than 0.999, to avoid instabilities near $t = T$.

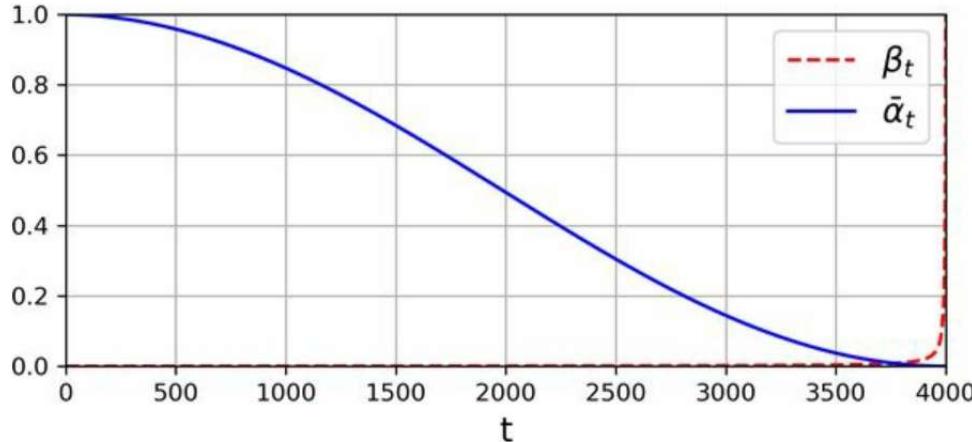


Figure 17-21. Noise variance schedule β_t , and the remaining signal variance $\bar{\alpha}_t$

Let's create a small function to compute α_t , β_t , and $\bar{\alpha}_t$, and call it with $T = 4,000$:

```
def variance_schedule(T, s=0.008, max_beta=0.999):
    t = np.arange(T + 1)
    f = np.cos((t / T + s) / (1 + s) * np.pi / 2) ** 2
    alpha = np.clip(f[1:] / f[:-1], 1 - max_beta, 1)
    alpha = np.append(1, alpha).astype(np.float32) # add alpha_0 = 1
    beta = 1 - alpha
    alpha_cumprod = np.cumprod(alpha)
    return alpha, alpha_cumprod, beta # alpha_t, alpha_t-bar, beta_t for t = 0 to T

T = 4000
alpha, alpha_cumprod, beta = variance_schedule(T)
```

To train our model to reverse the diffusion process, we will need noisy images from different time steps of the forward process. For this, let's create a `prepare_batch()` function that will take a batch of clean images from the dataset and prepare them:

```
def prepare_batch(X):
```

```

X = tf.cast(X[..., tf.newaxis], tf.float32) * 2 - 1 # scale from -1 to +1
X_shape = tf.shape(X)
t = tf.random.uniform([X_shape[0]], minval=1, maxval=T + 1, dtype=tf.int32)
alpha_cm = tf.gather(alpha_cumprod, t)
alpha_cm = tf.reshape(alpha_cm, [X_shape[0]] + [1] * (len(X_shape) - 1))
noise = tf.random.normal(X_shape)
return {
    "X_noisy": alpha_cm ** 0.5 * X + (1 - alpha_cm) ** 0.5 * noise,
    "time": t,
}, noise

```

Let's go through this code:

- For simplicity we will use Fashion MNIST, so the function must first add a channel axis. It will also help to scale the pixel values from -1 to 1 , so it's closer to the final Gaussian distribution with mean 0 and variance 1 .
- Next, the function creates t , a vector containing a random time step for each image in the batch, between 1 and T .
- Then it uses `tf.gather()` to get the value of $\alpha_{cumprod}$ for each of the time steps in the vector t . This gives us the vector α_cm , containing one value of $\bar{\alpha}_t$ for each image.
- The next line reshapes the α_cm from $[batch\ size]$ to $[batch\ size, 1, 1, 1]$. This is needed to ensure α_cm can be broadcasted with the batch X .
- Then we generate some Gaussian noise with mean 0 and variance 1 .
- Lastly, we use [Equation 17-6](#) to apply the diffusion process to the images. Note that $x ** 0.5$ is equal to the square root of x . The function returns a tuple containing the inputs and the targets. The inputs are represented as a Python dict containing the noisy images and the time steps used to generate them. The targets are the Gaussian noise used to generate each image.

NOTE

With this setup, the model will predict the noise that should be subtracted from the input image to get the original image. Why not predict the original image directly? Well, the authors tried: it simply doesn't work as well.

Next, we'll create a training dataset and a validation set that will apply the `prepare_batch()` function to every batch. As earlier, `X_train` and `X_valid` contain the Fashion MNIST images with pixel values ranging from 0 to 1:

```
def prepare_dataset(X, batch_size=32, shuffle=False):
    ds = tf.data.Dataset.from_tensor_slices(X)
    if shuffle:
        ds = ds.shuffle(buffer_size=10_000)
    return ds.batch(batch_size).map(prepare_batch).prefetch(1)

train_set = prepare_dataset(X_train, batch_size=32, shuffle=True)
valid_set = prepare_dataset(X_valid, batch_size=32)
```

Now we're ready to build the actual diffusion model itself. It can be any model you want, as long as it takes the noisy images and time steps as inputs, and predicts the noise to subtract from the input images:

```
def build_diffusion_model():
    X_noisy = tf.keras.layers.Input(shape=[28, 28, 1], name="X_noisy")
    time_input = tf.keras.layers.Input(shape=[], dtype=tf.int32, name="time")
    [...] # build the model based on the noisy images and the time steps
    outputs = [...] # predict the noise (same shape as the input images)
    return tf.keras.Model(inputs=[X_noisy, time_input], outputs=[outputs])
```

The DDPM authors used a modified [U-Net architecture](#),²² which has many similarities with the FCN architecture we discussed in [Chapter 14](#) for semantic segmentation: it's a convolutional neural network that gradually downsamples the input images, then gradually upsamples them again, with skip connections crossing over from each level of the downsampling part to the corresponding level in the upsampling part. To take into account the time steps, they encoded them using the same technique as the positional encodings in the transformer architecture (see [Chapter 16](#)). At every level in the U-Net architecture, they passed these time encodings through Dense layers and fed them to the U-Net. Lastly, they also used multi-head attention

layers at various levels. See this chapter's notebook for a basic implementation, or <https://homl.info/ddpmcode> for the official implementation: it's based on TF 1.x, which is deprecated, but it's quite readable.

We can now train the model normally. The authors noted that using the MAE loss worked better than the MSE. You can also use the Huber loss:

```
model = build_diffusion_model()
model.compile(loss=tf.keras.losses.Huber(), optimizer="adam")
history = model.fit(train_set, validation_data=valid_set, epochs=100)
```

Once the model is trained, you can use it to generate new images. Unfortunately, there's no shortcut in the reverse diffusion process, so you have to sample \mathbf{x}_T randomly from a Gaussian distribution with mean 0 and variance 1, then pass it to the model to predict the noise; subtract it from the image using [Equation 17-8](#), and you get \mathbf{x}_{T-1} . Repeat the process 3,999 more times until you get \mathbf{x}_0 : if all went well, it should look like a regular Fashion MNIST image!

Equation 17-8. Going one step in reverse in the diffusion process

$$\mathbf{x}_{t-1} = \mathbf{x}_t - \alpha^{-t} \epsilon_{\theta}(\mathbf{x}_t, t) + \beta \mathbf{z}$$

In this equation, $\epsilon_{\theta}(\mathbf{x}_t, t)$ represents the noise predicted by the model given the input image \mathbf{x}_t and the time step t . The θ represents the model parameters. Moreover, \mathbf{z} is Gaussian noise with mean 0 and variance 1. This makes the reverse process stochastic: if you run it multiple times, you will get different images.

Let's write a function that implements this reverse process, and call it to generate a few images:

```
def generate(model, batch_size=32):
    X = tf.random.normal([batch_size, 28, 28, 1])
    for t in range(T, 0, -1):
        noise = (tf.random.normal if t > 1 else tf.zeros)(tf.shape(X))
        X_noisy = model({"X_noisy": X, "time": tf.constant([t] * batch_size)})
        X = (
            1 / alpha[t] ** 0.5
```

```

        * (X - beta[t] / (1 - alpha_cumprod[t]) ** 0.5 * X_noise)
        + (1 - alpha[t]) ** 0.5 * noise
    )
return X

X_gen = generate(model) # generated images

```

This may take a minute or two. That's the main drawback of diffusion models: generating images is slow since the model needs to be called many times. It's possible to make this faster by using a smaller T value, or by using the same model prediction for several steps at a time, but the resulting images may not look as nice. That said, despite this speed limitation, diffusion models do produce high-quality and diverse images, as you can see in Figure 17-22.



Figure 17-22. Images generated by the DDPM

Diffusion models have made tremendous progress recently. In particular, a paper published in December 2021 by [Robin Rombach, Andreas Blattmann, et al.](#),²³ introduced *latent diffusion models*, where the diffusion process takes place in latent space, rather than in pixel space. To achieve this, a powerful autoencoder is used to compress each training image into a much smaller latent space, where the diffusion process takes place, then the autoencoder is used to decompress the final latent representation, generating the output image. This considerably speeds up image generation, and reduces

training time and cost dramatically. Importantly, the quality of the generated images is outstanding.

Moreover, the researchers also adapted various conditioning techniques to guide the diffusion process using text prompts, images, or any other inputs. This makes it possible to quickly produce a beautiful, high-resolution image of a salamander reading a book, or anything else you might fancy. You can also condition the image generation process using an input image. This enables many applications, such as outpainting—where an input image is extended beyond its borders—or inpainting—where holes in an image are filled in.

Lastly, a powerful pretrained latent diffusion model named *Stable Diffusion* was open sourced in August 2022 by a collaboration between LMU Munich and a few companies, including StabilityAI, and Runway, with support from EleutherAI and LAION. In September 2022, it was ported to TensorFlow and included in [KerasCV](#), a computer vision library built by the Keras team. Now anyone can generate mindblowing images in seconds, for free, even on a regular laptop (see the last exercise in this chapter). The possibilities are endless!

In the next chapter we will move on to an entirely different branch of deep learning: deep reinforcement learning.

Exercises

1. What are the main tasks that autoencoders are used for?
2. Suppose you want to train a classifier, and you have plenty of unlabeled training data but only a few thousand labeled instances. How can autoencoders help? How would you proceed?
3. If an autoencoder perfectly reconstructs the inputs, is it necessarily a good autoencoder? How can you evaluate the performance of an autoencoder?
4. What are undercomplete and overcomplete autoencoders? What is the main risk of an excessively undercomplete autoencoder? What about the main risk of an overcomplete autoencoder?
5. How do you tie weights in a stacked autoencoder? What is the point of doing so?
6. What is a generative model? Can you name a type of generative autoencoder?
7. What is a GAN? Can you name a few tasks where GANs can shine?
8. What are the main difficulties when training GANs?
9. What are diffusion models good at? What is their main limitation?
10. Try using a denoising autoencoder to pretrain an image classifier. You can use MNIST (the simplest option), or a more complex image dataset such as [CIFAR10](#) if you want a bigger challenge. Regardless of the dataset you're using, follow these steps:
 - a. Split the dataset into a training set and a test set. Train a deep denoising autoencoder on the full training set.
 - b. Check that the images are fairly well reconstructed. Visualize the images that most activate each neuron in the coding layer.

- c. Build a classification DNN, reusing the lower layers of the autoencoder. Train it using only 500 images from the training set. Does it perform better with or without pretraining?
11. Train a variational autoencoder on the image dataset of your choice, and use it to generate images. Alternatively, you can try to find an unlabeled dataset that you are interested in and see if you can generate new samples.
 12. Train a DCGAN to tackle the image dataset of your choice, and use it to generate images. Add experience replay and see if this helps. Turn it into a conditional GAN where you can control the generated class.
 13. Go through KerasCV's excellent [Stable Diffusion tutorial](#), and generate a beautiful drawing of a salamander reading a book. If you post your best drawing on Twitter, please tag me at @aureliengeron. I'd love to see your creations!

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

¹ William G. Chase and Herbert A. Simon, "Perception in Chess", *Cognitive Psychology* 4, no. 1 (1973): 55–81.

² Yoshua Bengio et al., "Greedy Layer-Wise Training of Deep Networks", *Proceedings of the 19th International Conference on Neural Information Processing Systems* (2006): 153–160.

³ Jonathan Masci et al., "Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction", *Proceedings of the 21st International Conference on Artificial Neural Networks* 1 (2011): 52–59.

⁴ Pascal Vincent et al., "Extracting and Composing Robust Features with Denoising Autoencoders", *Proceedings of the 25th International Conference on Machine Learning* (2008): 1096–1103.

⁵ Pascal Vincent et al., "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion", *Journal of Machine Learning Research* 11 (2010): 3371–3408.

⁶ Diederik Kingma and Max Welling, "Auto-Encoding Variational Bayes", arXiv preprint arXiv:1312.6114 (2013).

⁷ Variational autoencoders are actually more general; the codings are not limited to Gaussian

distributions.

- 8 For more mathematical details, check out the original paper on variational autoencoders, or Carl Doersch's [great tutorial](#) (2016).
- 9 Ian Goodfellow et al., "Generative Adversarial Nets", *Proceedings of the 27th International Conference on Neural Information Processing Systems* 2 (2014): 2672–2680.
- 10 For a nice comparison of the main GAN losses, check out this great [GitHub project by Hwalsuk Lee](#).
- 11 Mario Lucic et al., "Are GANs Created Equal? A Large-Scale Study", *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (2018): 698–707.
- 12 Alec Radford et al., "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks", arXiv preprint arXiv:1511.06434 (2015).
- 13 Reproduced with the kind authorization of the authors.
- 14 Mehdi Mirza and Simon Osindero, "Conditional Generative Adversarial Nets", arXiv preprint arXiv:1411.1784 (2014).
- 15 Tero Karras et al., "Progressive Growing of GANs for Improved Quality, Stability, and Variation", *Proceedings of the International Conference on Learning Representations* (2018).
- 16 The dynamic range of a variable is the ratio between the highest and the lowest value it may take.
- 17 Tero Karras et al., "A Style-Based Generator Architecture for Generative Adversarial Networks", arXiv preprint arXiv:1812.04948 (2018).
- 18 Reproduced with the kind authorization of the authors.
- 19 Jascha Sohl-Dickstein et al., "Deep Unsupervised Learning using Nonequilibrium Thermodynamics", arXiv preprint arXiv:1503.03585 (2015).
- 20 Jonathan Ho et al., "Denoising Diffusion Probabilistic Models" (2020).
- 21 Alex Nichol and Prafulla Dhariwal, "Improved Denoising Diffusion Probabilistic Models" (2021).
- 22 Olaf Ronneberger et al., "U-Net: Convolutional Networks for Biomedical Image Segmentation", arXiv preprint arXiv:1505.04597 (2015).
- 23 Robin Rombach, Andreas Blattmann, et al., "High-Resolution Image Synthesis with Latent Diffusion Models", arXiv preprint arXiv:2112.10752 (2021).

Chapter 18. Reinforcement Learning

Reinforcement learning (RL) is one of the most exciting fields of machine learning today, and also one of the oldest. It has been around since the 1950s, producing many interesting applications over the years,¹ particularly in games (e.g., *TD-Gammon*, a Backgammon-playing program) and in machine control, but seldom making the headline news. However, a revolution took place in 2013, when researchers from a British startup called DeepMind demonstrated a system that could learn to play just about any Atari game from scratch,² eventually outperforming humans³ in most of them, using only raw pixels as inputs and without any prior knowledge of the rules of the games.⁴ This was the first of a series of amazing feats, culminating with the victory of their system AlphaGo against Lee Sedol, a legendary professional player of the game of Go, in March 2016 and against Ke Jie, the world champion, in May 2017. No program had ever come close to beating a master of this game, let alone the world champion. Today the whole field of RL is boiling with new ideas, with a wide range of applications.

So how did DeepMind (bought by Google for over \$500 million in 2014) achieve all this? With hindsight it seems rather simple: they applied the power of deep learning to the field of reinforcement learning, and it worked beyond their wildest dreams. In this chapter I will first explain what reinforcement learning is and what it's good at, then present two of the most important techniques in deep reinforcement learning: policy gradients and deep Q-networks, including a discussion of Markov decision processes. Let's get started!

Learning to Optimize Rewards

In reinforcement learning, a software *agent* makes *observations* and takes *actions* within an *environment*, and in return it receives *rewards* from the environment. Its objective is to learn to act in a way that will maximize its expected rewards over time. If you don't mind a bit of anthropomorphism, you can think of positive rewards as pleasure, and negative rewards as pain (the term "reward" is a bit misleading in this case). In short, the agent acts in the environment and learns by trial and error to maximize its pleasure and minimize its pain.

This is quite a broad setting, which can apply to a wide variety of tasks. Here are a few examples (see [Figure 18-1](#)):

1. The agent can be the program controlling a robot. In this case, the environment is the real world, the agent observes the environment through a set of *sensors* such as cameras and touch sensors, and its actions consist of sending signals to activate motors. It may be programmed to get positive rewards whenever it approaches the target destination, and negative rewards whenever it wastes time or goes in the wrong direction.
2. The agent can be the program controlling *Ms. Pac-Man*. In this case, the environment is a simulation of the Atari game, the actions are the nine possible joystick positions (upper left, down, center, and so on), the observations are screenshots, and the rewards are just the game points.
3. Similarly, the agent can be the program playing a board game such as Go. It only gets a reward if it wins.
4. The agent does not have to control a physically (or virtually) moving thing. For example, it can be a smart thermostat, getting positive rewards whenever it is close to the target temperature and saves energy, and negative rewards when humans need to tweak the temperature, so the agent must learn to anticipate human needs.

- The agent can observe stock market prices and decide how much to buy or sell every second. Rewards are obviously the monetary gains and losses.

Note that there may not be any positive rewards at all; for example, the agent may move around in a maze, getting a negative reward at every time step, so it had better find the exit as quickly as possible! There are many other examples of tasks to which reinforcement learning is well suited, such as self-driving cars, recommender systems, placing ads on a web page, or controlling where an image classification system should focus its attention.

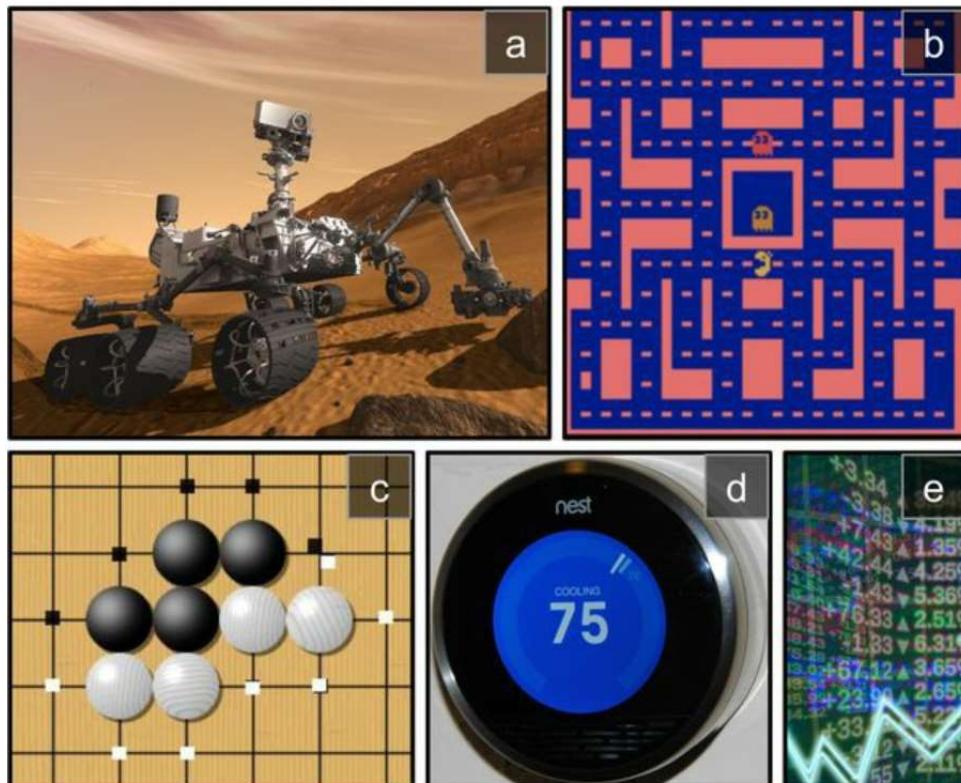


Figure 18-1. Reinforcement learning examples: (a) robotics, (b) Ms. Pac-Man, (c) Go player, (d) thermostat, (e) automatic trader ⁵

Policy Search

The algorithm a software agent uses to determine its actions is called its *policy*. The policy could be a neural network taking observations as inputs and outputting the action to take (see [Figure 18-2](#)).

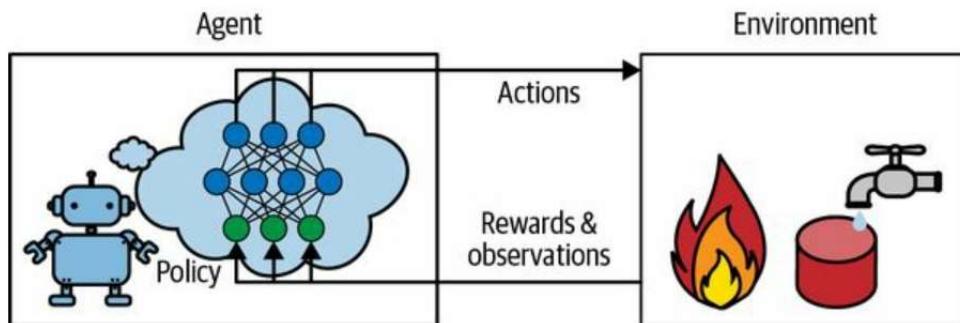


Figure 18-2. Reinforcement learning using a neural network policy

The policy can be any algorithm you can think of, and it does not have to be deterministic. In fact, in some cases it does not even have to observe the environment! For example, consider a robotic vacuum cleaner whose reward is the amount of dust it picks up in 30 minutes. Its policy could be to move forward with some probability p every second, or randomly rotate left or right with probability $1 - p$. The rotation angle would be a random angle between $-r$ and $+r$. Since this policy involves some randomness, it is called a *stochastic policy*. The robot will have an erratic trajectory, which guarantees that it will eventually get to any place it can reach and pick up all the dust. The question is, how much dust will it pick up in 30 minutes?

How would you train such a robot? There are just two *policy parameters* you can tweak: the probability p and the angle range r . One possible learning algorithm could be to try out many different values for these parameters, and pick the combination that performs best (see [Figure 18-3](#)). This is an example of *policy search*, in this case using a brute-force approach. When the *policy space* is too large (which is generally the case), finding a good set of parameters this way is like searching for a needle in a gigantic haystack.

Another way to explore the policy space is to use *genetic algorithms*. For example, you could randomly create a first generation of 100 policies and try them out, then “kill” the 80 worst policies ⁶ and make the 20 survivors produce 4 offspring each. An offspring is a copy of its parent ⁷ plus some random variation. The surviving policies plus their offspring together constitute the second generation. You can continue to iterate through generations this way until you find a good policy. ⁸

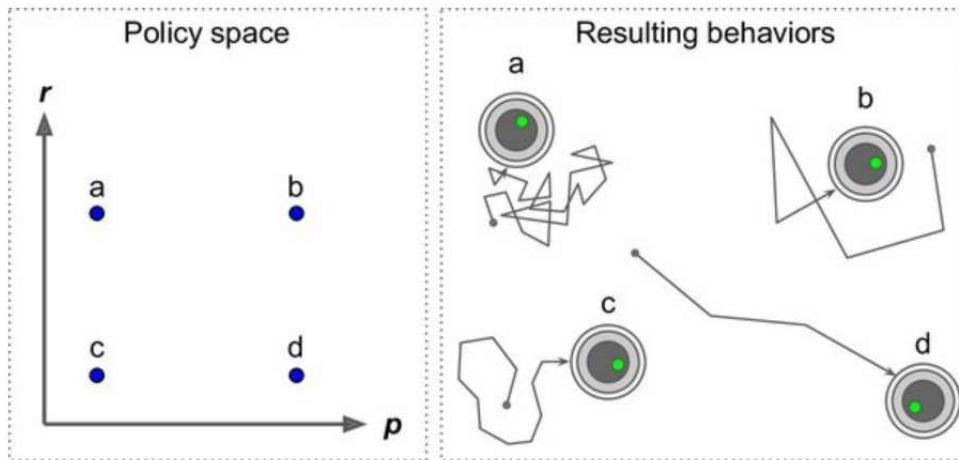


Figure 18-3. Four points in the policy space (left) and the agent's corresponding behavior (right)

Yet another approach is to use optimization techniques, by evaluating the gradients of the rewards with regard to the policy parameters, then tweaking these parameters by following the gradients toward higher rewards. ⁹ We will discuss this approach, called *policy gradients* (PG), in more detail later in this chapter. Going back to the vacuum cleaner robot, you could slightly increase p and evaluate whether doing so increases the amount of dust picked up by the robot in 30 minutes; if it does, then increase p some more, or else reduce p . We will implement a popular PG algorithm using TensorFlow, but before we do, we need to create an environment for the agent to live in —so it’s time to introduce OpenAI Gym.

Introduction to OpenAI Gym

One of the challenges of reinforcement learning is that in order to train an agent, you first need to have a working environment. If you want to program an agent that will learn to play an Atari game, you will need an Atari game simulator. If you want to program a walking robot, then the environment is the real world, and you can directly train your robot in that environment. However, this has its limits: if the robot falls off a cliff, you can't just click Undo. You can't speed up time either—adding more computing power won't make the robot move any faster—and it's generally too expensive to train 1,000 robots in parallel. In short, training is hard and slow in the real world, so you generally need a *simulated environment* at least for bootstrap training. For example, you might use a library like **PyBullet** or **MuJoCo** for 3D physics simulation.

OpenAI Gym¹⁰ is a toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, and so on), that you can use to train agents, compare them, or develop new RL algorithms.

OpenAI Gym is preinstalled on Colab, but it's an older version, so you'll need to replace it with the latest one. You also need to install a few of its dependencies. If you are coding on your own machine instead of Colab, and you followed the installation instructions at <https://homl.info/install>, then you can skip this step; otherwise, enter these commands:

```
# Only run these commands on Colab or Kaggle!
%pip install -q -U gym
%pip install -q -U gym[classic_control,box2d,atari,accept-rom-license]
```

The first %pip command upgrades Gym to the latest version. The -q option stands for *quiet*: it makes the output less verbose. The -U option stands for *upgrade*. The second %pip command installs the libraries required to run various kinds of environments. This includes classic environments from *control theory*—the science of controlling dynamical systems—such as

balancing a pole on a cart. It also includes environments based on the Box2D library—a 2D physics engine for games. Lastly, it includes environments based on the Arcade Learning Environment (ALE), which is an emulator for Atari 2600 games. Several Atari game ROMs are downloaded automatically, and by running this code you agree with Atari’s ROM licenses.

With that, you’re ready to use OpenAI Gym. Let’s import it and make an environment:

```
import gym  
env = gym.make("CartPole-v1", render_mode="rgb_array")
```

Here, we’ve created a CartPole environment. This is a 2D simulation in which a cart can be accelerated left or right in order to balance a pole placed on top of it (see [Figure 18-4](#)). This is a classic control task.

TIP

The `gym.envs.registry` dictionary contains the names and specifications of all the available environments.

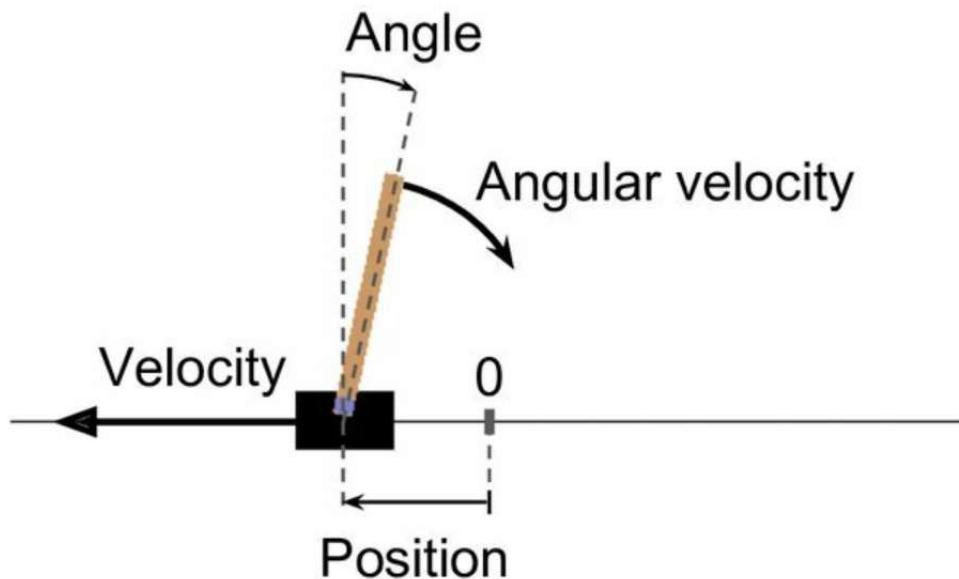


Figure 18-4. The CartPole environment

After the environment is created, you must initialize it using the `reset()` method, optionally specifying a random seed. This returns the first observation. Observations depend on the type of environment. For the CartPole environment, each observation is a 1D NumPy array containing four floats representing the cart's horizontal position (0.0 = center), its velocity (positive means right), the angle of the pole (0.0 = vertical), and its angular velocity (positive means clockwise). The `reset()` method also returns a dictionary that may contain extra environment-specific information. This can be useful for debugging or for training. For example, in many Atari environments, it contains the number of lives left. However, in the CartPole environment, this dictionary is empty.

```
>>> obs, info = env.reset(seed=42)
>>> obs
array([ 0.0273956, -0.00611216,  0.03585979,  0.0197368], dtype=float32)
>>> info
{}
```

Let's call the `render()` method to render this environment as an image. Since

we set `render_mode="rgb_array"` when creating the environment, the image will be returned as a NumPy array:

```
>>> img = env.render()
>>> img.shape # height, width, channels (3 = Red, Green, Blue)
(400, 600, 3)
```

You can then use Matplotlib's `imshow()` function to display this image, as usual.

Now let's ask the environment what actions are possible:

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` means that the possible actions are integers 0 and 1, which represent accelerating left or right. Other environments may have additional discrete actions, or other kinds of actions (e.g., continuous). Since the pole is leaning toward the right (`obs[2] > 0`), let's accelerate the cart toward the right:

```
>>> action = 1 # accelerate right
>>> obs, reward, done, truncated, info = env.step(action)
>>> obs
array([ 0.02727336,  0.18847767,  0.03625453, -0.26141977], dtype=float32)
>>> reward
1.0
>>> done
False
>>> truncated
False
>>> info
{}
```

The `step()` method executes the desired action and returns five values:

obs

This is the new observation. The cart is now moving toward the right (`obs[1] > 0`). The pole is still tilted toward the right (`obs[2] > 0`), but its angular velocity is now negative (`obs[3] < 0`), so it will likely be tilted

toward the left after the next step.

reward

In this environment, you get a reward of 1.0 at every step, no matter what you do, so the goal is to keep the episode running for as long as possible.

done

This value will be True when the episode is over. This will happen when the pole tilts too much, or goes off the screen, or after 200 steps (in this last case, you have won). After that, the environment must be reset before it can be used again.

truncated

This value will be True when an episode is interrupted early, for example by an environment wrapper that imposes a maximum number of steps per episode (see Gym's documentation for more details on environment wrappers). Some RL algorithms treat truncated episodes differently from episodes finished normally (i.e., when done is True), but in this chapter we will treat them identically.

info

This environment-specific dictionary may provide extra information, just like the one returned by the reset() method.

TIP

Once you have finished using an environment, you should call its close() method to free resources.

Let's hardcode a simple policy that accelerates left when the pole is leaning toward the left and accelerates right when the pole is leaning toward the right. We will run this policy to see the average rewards it gets over 500 episodes:

```

def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs, info = env.reset(seed=episode)
    for step in range(200):
        action = basic_policy(obs)
        obs, reward, done, truncated, info = env.step(action)
        episode_rewards += reward
        if done or truncated:
            break
    totals.append(episode_rewards)

```

This code is self-explanatory. Let's look at the result:

```

>>> import numpy as np
>>> np.mean(totals), np.std(totals), min(totals), max(totals)
(41.698, 8.389445512070509, 24.0, 63.0)

```

Even with 500 tries, this policy never managed to keep the pole upright for more than 63 consecutive steps. Not great. If you look at the simulation in this chapter's notebook, you will see that the cart oscillates left and right more and more strongly until the pole tilts too much. Let's see if a neural network can come up with a better policy.

Neural Network Policies

Let's create a neural network policy. This neural network will take an observation as input, and it will output the action to be executed, just like the policy we hardcoded earlier. More precisely, it will estimate a probability for each action, and then we will select an action randomly, according to the estimated probabilities (see [Figure 18-5](#)). In the case of the CartPole environment, there are just two possible actions (left or right), so we only need one output neuron. It will output the probability p of action 0 (left), and of course the probability of action 1 (right) will be $1 - p$. For example, if it outputs 0.7, then we will pick action 0 with 70% probability, or action 1 with 30% probability.

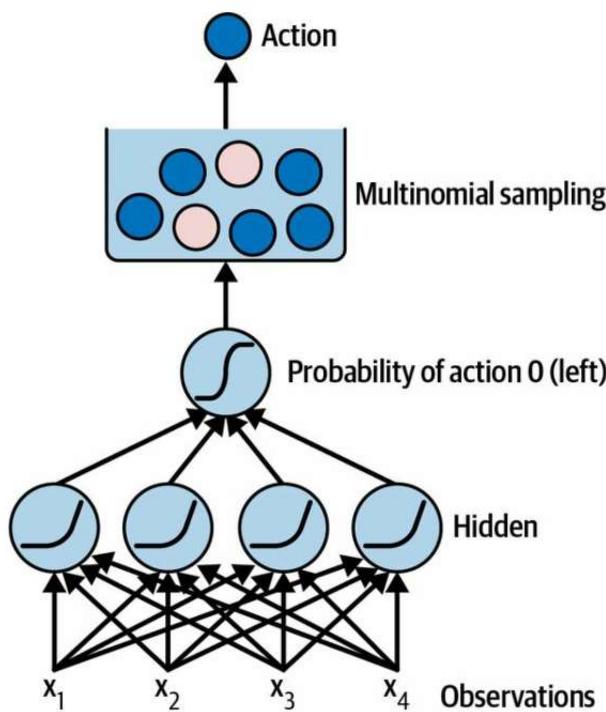


Figure 18-5. Neural network policy

You may wonder why we are picking a random action based on the

probabilities given by the neural network, rather than just picking the action with the highest score. This approach lets the agent find the right balance between *exploring* new actions and *exploiting* the actions that are known to work well. Here's an analogy: suppose you go to a restaurant for the first time, and all the dishes look equally appealing, so you randomly pick one. If it turns out to be good, you can increase the probability that you'll order it next time, but you shouldn't increase that probability up to 100%, or else you will never try out the other dishes, some of which may be even better than the one you tried. This *exploration/exploitation dilemma* is central in reinforcement learning.

Also note that in this particular environment, the past actions and observations can safely be ignored, since each observation contains the environment's full state. If there were some hidden state, then you might need to consider past actions and observations as well. For example, if the environment only revealed the position of the cart but not its velocity, you would have to consider not only the current observation but also the previous observation in order to estimate the current velocity. Another example is when the observations are noisy; in that case, you generally want to use the past few observations to estimate the most likely current state. The CartPole problem is thus as simple as can be; the observations are noise-free, and they contain the environment's full state.

Here is the code to build a basic neural network policy using Keras:

```
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(5, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid"),
])
```

We use a Sequential model to define the policy network. The number of inputs is the size of the observation space—which in the case of CartPole is 4—and we have just five hidden units because it's a fairly simple task. Finally, we want to output a single probability—the probability of going left—so we have a single output neuron using the sigmoid activation function. If there

were more than two possible actions, there would be one output neuron per action, and we would use the softmax activation function instead.

OK, we now have a neural network policy that will take observations and output action probabilities. But how do we train it?

Evaluating Actions: The Credit Assignment Problem

If we knew what the best action was at each step, we could train the neural network as usual, by minimizing the cross entropy between the estimated probability distribution and the target probability distribution. It would just be regular supervised learning. However, in reinforcement learning the only guidance the agent gets is through rewards, and rewards are typically sparse and delayed. For example, if the agent manages to balance the pole for 100 steps, how can it know which of the 100 actions it took were good, and which of them were bad? All it knows is that the pole fell after the last action, but surely this last action is not entirely responsible. This is called the *credit assignment problem*: when the agent gets a reward, it is hard for it to know which actions should get credited (or blamed) for it. Think of a dog that gets rewarded hours after it behaved well; will it understand what it is being rewarded for?

To tackle this problem, a common strategy is to evaluate an action based on the sum of all the rewards that come after it, usually applying a *discount factor*, γ (gamma), at each step. This sum of discounted rewards is called the action's *return*. Consider the example in [Figure 18-6](#). If an agent decides to go right three times in a row and gets +10 reward after the first step, 0 after the second step, and finally -50 after the third step, then assuming we use a discount factor $\gamma = 0.8$, the first action will have a return of $10 + \gamma \times 0 + \gamma^2 \times (-50) = -22$. If the discount factor is close to 0, then future rewards won't count for much compared to immediate rewards. Conversely, if the discount factor is close to 1, then rewards far into the future will count almost as much as immediate rewards. Typical discount factors vary from 0.9 to 0.99. With a discount factor of 0.95, rewards 13 steps into the future count roughly for half as much as immediate rewards (since $0.95^{13} \approx 0.5$), while with a discount factor of 0.99, rewards 69 steps into the future count for half as much as immediate rewards. In the CartPole environment, actions have fairly short-term effects, so choosing a discount factor of 0.95 seems reasonable.

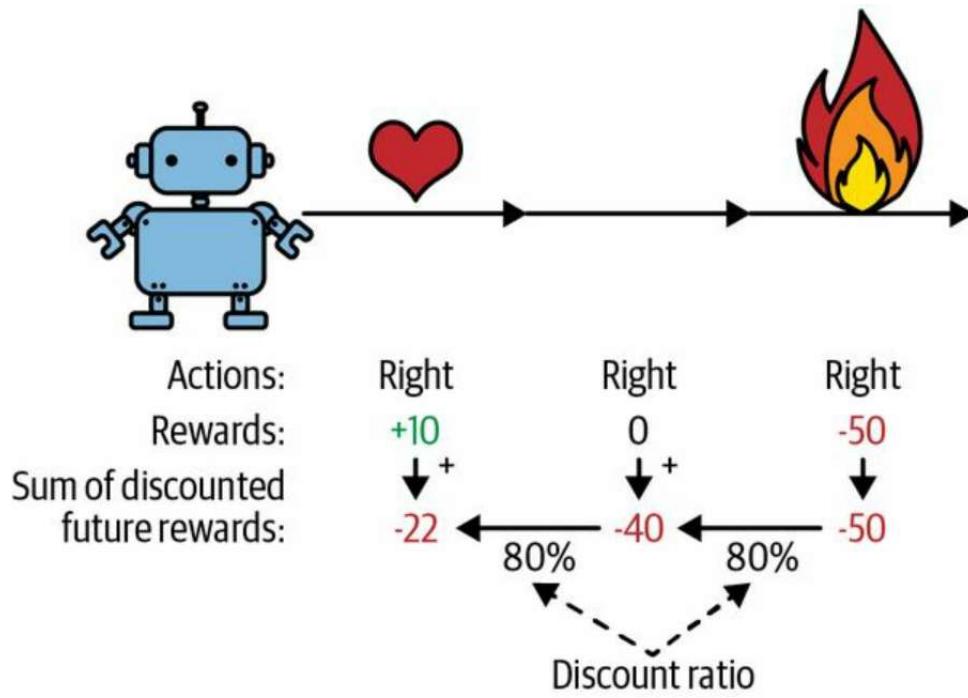


Figure 18-6. Computing an action's return: the sum of discounted future rewards

Of course, a good action may be followed by several bad actions that cause the pole to fall quickly, resulting in the good action getting a low return. Similarly, a good actor may sometimes star in a terrible movie. However, if we play the game enough times, on average good actions will get a higher return than bad ones. We want to estimate how much better or worse an action is, compared to the other possible actions, on average. This is called the *action advantage*. For this, we must run many episodes and normalize all the action returns, by subtracting the mean and dividing by the standard deviation. After that, we can reasonably assume that actions with a negative advantage were bad while actions with a positive advantage were good. OK, now that we have a way to evaluate each action, we are ready to train our first agent using policy gradients. Let's see how.

Policy Gradients

As discussed earlier, PG algorithms optimize the parameters of a policy by following the gradients toward higher rewards. One popular class of PG algorithms, called *REINFORCE algorithms*, was introduced back in 1992¹¹ by Ronald Williams. Here is one common variant:

1. First, let the neural network policy play the game several times, and at each step, compute the gradients that would make the chosen action even more likely—but don’t apply these gradients yet.
2. Once you have run several episodes, compute each action’s advantage, using the method described in the previous section.
3. If an action’s advantage is positive, it means that the action was probably good, and you want to apply the gradients computed earlier to make the action even more likely to be chosen in the future. However, if the action’s advantage is negative, it means the action was probably bad, and you want to apply the opposite gradients to make this action slightly *less* likely in the future. The solution is to multiply each gradient vector by the corresponding action’s advantage.
4. Finally, compute the mean of all the resulting gradient vectors, and use it to perform a gradient descent step.

Let’s use Keras to implement this algorithm. We will train the neural network policy we built earlier so that it learns to balance the pole on the cart. First, we need a function that will play one step. We will pretend for now that whatever action it takes is the right one so that we can compute the loss and its gradients. These gradients will just be saved for a while, and we will modify them later depending on how good or bad the action turned out to be:

```
def play_one_step(env, obs, model, loss_fn):
    with tf.GradientTape() as tape:
        left_proba = model(obs[np.newaxis])
        action = (tf.random.uniform([1, 1]) > left_proba)
```

```

y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)
loss = tf.reduce_mean(loss_fn(y_target, left_proba))

grads = tape.gradient(loss, model.trainable_variables)
obs, reward, done, truncated, info = env.step(int(action))
return obs, reward, done, truncated, grads

```

Let's walk through this function:

- Within the GradientTape block (see [Chapter 12](#)), we start by calling the model, giving it a single observation. We reshape the observation so it becomes a batch containing a single instance, as the model expects a batch. This outputs the probability of going left.
- Next, we sample a random float between 0 and 1, and we check whether it is greater than left_proba. The action will be False with probability left_proba, or True with probability $1 - \text{left_proba}$. Once we cast this Boolean to an integer, the action will be 0 (left) or 1 (right) with the appropriate probabilities.
- We now define the target probability of going left: it is 1 minus the action (cast to a float). If the action is 0 (left), then the target probability of going left will be 1. If the action is 1 (right), then the target probability will be 0.
- Then we compute the loss using the given loss function, and we use the tape to compute the gradient of the loss with regard to the model's trainable variables. Again, these gradients will be tweaked later, before we apply them, depending on how good or bad the action turned out to be.
- Finally, we play the selected action, and we return the new observation, the reward, whether the episode is ended or not, whether it is truncated or not, and of course the gradients that we just computed.

Now let's create another function that will rely on the `play_one_step()` function to play multiple episodes, returning all the rewards and gradients for each episode and each step:

```

def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):
    all_rewards = []
    all_grads = []
    for episode in range(n_episodes):
        current_rewards = []
        current_grads = []
        obs, info = env.reset()
        for step in range(n_max_steps):
            obs, reward, done, truncated, grads = play_one_step(
                env, obs, model, loss_fn)
            current_rewards.append(reward)
            current_grads.append(grads)
            if done or truncated:
                break
        all_rewards.append(current_rewards)
        all_grads.append(current_grads)

    return all_rewards, all_grads

```

This code returns a list of reward lists: one reward list per episode, containing one reward per step. It also returns a list of gradient lists: one gradient list per episode, each containing one tuple of gradients per step and each tuple containing one gradient tensor per trainable variable.

The algorithm will use the `play_multiple_episodes()` function to play the game several times (e.g., 10 times), then it will go back and look at all the rewards, discount them, and normalize them. To do that, we need a couple more functions; the first will compute the sum of future discounted rewards at each step, and the second will normalize all these discounted rewards (i.e., the returns) across many episodes by subtracting the mean and dividing by the standard deviation:

```

def discount_rewards(rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_factor
    return discounted

def discount_and_normalize_rewards(all_rewards, discount_factor):
    all_discounted_rewards = [discount_rewards(rewards, discount_factor)
                             for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)

```

```
reward_mean = flat_rewards.mean()
reward_std = flat_rewards.std()
return [(discounted_rewards - reward_mean) / reward_std
        for discounted_rewards in all_discounted_rewards]
```

Let's check that this works:

```
>>> discount_rewards([10, 0, -50], discount_factor=0.8)
array([-22, -40, -50])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]],
...                                discount_factor=0.8)
...
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([1.26665318, 1.07277777])]
```

The call to `discount_rewards()` returns exactly what we expect (see [Figure 18-6](#)). You can verify that the function `discount_and_normalize_rewards()` does indeed return the normalized action advantages for each action in both episodes. Notice that the first episode was much worse than the second, so its normalized advantages are all negative; all actions from the first episode would be considered bad, and conversely all actions from the second episode would be considered good.

We are almost ready to run the algorithm! Now let's define the hyperparameters. We will run 150 training iterations, playing 10 episodes per iteration, and each episode will last at most 200 steps. We will use a discount factor of 0.95:

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95
```

We also need an optimizer and the loss function. A regular Nadam optimizer with learning rate 0.01 will do just fine, and we will use the binary cross-entropy loss function because we are training a binary classifier (there are two possible actions—left or right):

```
optimizer = tf.keras.optimizers.Nadam(learning_rate=0.01)
```

```
loss_fn = tf.keras.losses.binary_crossentropy
```

We are now ready to build and run the training loop!

```
for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)
    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_factor)

    all_mean_grads = []
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean([
            final_reward * all_grads[episode_index][step][var_index]
            for episode_index, final_rewards in enumerate(all_final_rewards)
            for step, final_reward in enumerate(final_rewards)], axis=0)
        all_mean_grads.append(mean_grads)

optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))
```

Let's walk through this code:

- At each training iteration, this loop calls the `play_multiple_episodes()` function, which plays 10 episodes and returns the rewards and gradients for each step in each episode.
- Then we call the `discount_and_normalize_rewards()` function to compute each action's normalized advantage, called the `final_reward` in this code. This provides a measure of how good or bad each action actually was, in hindsight.
- Next, we go through each trainable variable, and for each of them we compute the weighted mean of the gradients for that variable over all episodes and all steps, weighted by the `final_reward`.
- Finally, we apply these mean gradients using the optimizer: the model's trainable variables will be tweaked, and hopefully the policy will be a bit better.

And we're done! This code will train the neural network policy, and it will successfully learn to balance the pole on the cart. The mean reward per episode will get very close to 200. By default, that's the maximum for this

environment. Success!

The simple policy gradients algorithm we just trained solved the CartPole task, but it would not scale well to larger and more complex tasks. Indeed, it is highly *sample inefficient*, meaning it needs to explore the game for a very long time before it can make significant progress. This is due to the fact that it must run multiple episodes to estimate the advantage of each action, as we have seen. However, it is the foundation of more powerful algorithms, such as *actor-critic* algorithms (which we will discuss briefly at the end of this chapter).

TIP

Researchers try to find algorithms that work well even when the agent initially knows nothing about the environment. However, unless you are writing a paper, you should not hesitate to inject prior knowledge into the agent, as it will speed up training dramatically. For example, since you know that the pole should be as vertical as possible, you could add negative rewards proportional to the pole's angle. This will make the rewards much less sparse and speed up training. Also, if you already have a reasonably good policy (e.g., hardcoded), you may want to train the neural network to imitate it before using policy gradients to improve it.

We will now look at another popular family of algorithms. Whereas PG algorithms directly try to optimize the policy to increase rewards, the algorithms we will explore now are less direct: the agent learns to estimate the expected return for each state, or for each action in each state, then it uses this knowledge to decide how to act. To understand these algorithms, we must first consider *Markov decision processes* (MDPs).

Markov Decision Processes

In the early 20th century, the mathematician Andrey Markov studied stochastic processes with no memory, called *Markov chains*. Such a process has a fixed number of states, and it randomly evolves from one state to another at each step. The probability for it to evolve from a state s to a state s' is fixed, and it depends only on the pair (s, s') , not on past states. This is why we say that the system has no memory.

Figure 18-7 shows an example of a Markov chain with four states.

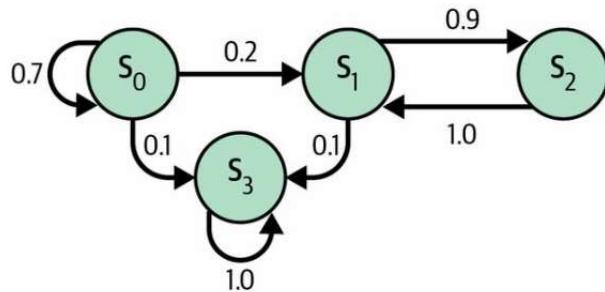


Figure 18-7. Example of a Markov chain

Suppose that the process starts in state s_0 , and there is a 70% chance that it will remain in that state at the next step. Eventually it is bound to leave that state and never come back, because no other state points back to s_0 . If it goes to state s_1 , it will then most likely go to state s_2 (90% probability), then immediately back to state s_1 (with 100% probability). It may alternate a number of times between these two states, but eventually it will fall into state s_3 and remain there forever, since there's no way out: this is called a *terminal state*. Markov chains can have very different dynamics, and they are heavily used in thermodynamics, chemistry, statistics, and much more.

Markov decision processes were first described in the 1950s by [Richard Bellman](#).¹² They resemble Markov chains, but with a twist: at each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action. Moreover, some state transitions return some reward (positive or negative), and the agent's goal is to find a

policy that will maximize reward over time.

For example, the MDP represented in [Figure 18-8](#) has three states (represented by circles) and up to three possible discrete actions at each step (represented by diamonds).

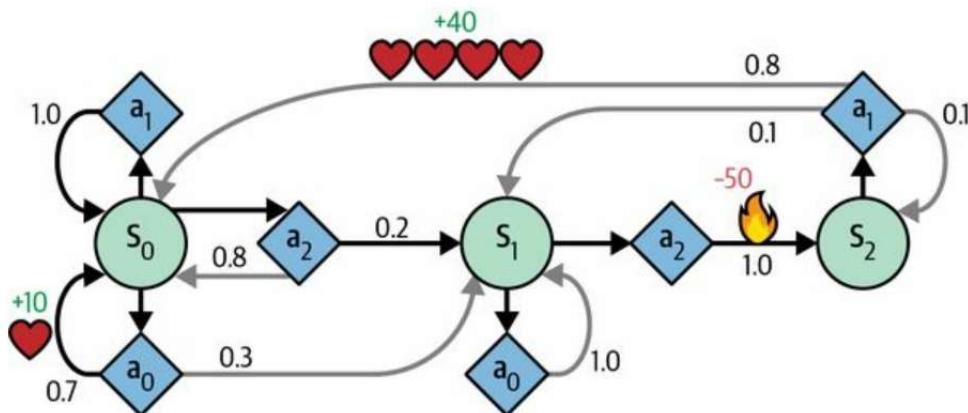


Figure 18-8. Example of a Markov decision process

If it starts in state s_0 , the agent can choose between actions a_0 , a_1 , or a_2 . If it chooses action a_1 , it just remains in state s_0 with certainty, and without any reward. It can thus decide to stay there forever if it wants to. But if it chooses action a_0 , it has a 70% probability of gaining a reward of +10 and remaining in state s_0 . It can then try again and again to gain as much reward as possible, but at one point it is going to end up instead in state s_1 . In state s_1 it has only two possible actions: a_0 or a_2 . It can choose to stay put by repeatedly choosing action a_0 , or it can choose to move on to state s_2 and get a negative reward of -50 (ouch). In state s_2 it has no choice but to take action a_1 , which will most likely lead it back to state s_0 , gaining a reward of +40 on the way. You get the picture. By looking at this MDP, can you guess which strategy will gain the most reward over time? In state s_0 it is clear that action a_0 is the best option, and in state s_2 the agent has no choice but to take action a_1 , but in state s_1 it is not obvious whether the agent should stay put (a_0) or go through the fire (a_2).

Bellman found a way to estimate the *optimal state value* of any state s , noted $V^*(s)$, which is the sum of all discounted future rewards the agent can expect

on average after it reaches the state, assuming it acts optimally. He showed that if the agent acts optimally, then the *Bellman optimality equation* applies (see [Equation 18-1](#)). This recursive equation says that if the agent acts optimally, then the optimal value of the current state is equal to the reward it will get on average after taking one optimal action, plus the expected optimal value of all possible next states that this action can lead to.

Equation 18-1. Bellman optimality equation

$$V^*(s) = \max_a \sum s' T(s, a, s') [R(s, a, s') + \gamma \cdot V^*(s')] \text{ for all } s$$

In this equation:

- $T(s, a, s')$ is the transition probability from state s to state s' , given that the agent chose action a . For example, in [Figure 18-8](#), $T(s_2, a_1, s_0) = 0.8$.
- $R(s, a, s')$ is the reward that the agent gets when it goes from state s to state s' , given that the agent chose action a . For example, in [Figure 18-8](#), $R(s_2, a_1, s_0) = +40$.
- γ is the discount factor.

This equation leads directly to an algorithm that can precisely estimate the optimal state value of every possible state: first initialize all the state value estimates to zero, and then iteratively update them using the *value iteration* algorithm (see [Equation 18-2](#)). A remarkable result is that, given enough time, these estimates are guaranteed to converge to the optimal state values, corresponding to the optimal policy.

Equation 18-2. Value iteration algorithm

$$V^{k+1}(s) \leftarrow \max_a \sum s' T(s, a, s') [R(s, a, s') + \gamma \cdot V^k(s')] \text{ for all } s$$

In this equation, $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.

NOTE

This algorithm is an example of *dynamic programming*, which breaks down a complex

problem into tractable subproblems that can be tackled iteratively.

Knowing the optimal state values can be useful, in particular to evaluate a policy, but it does not give us the optimal policy for the agent. Luckily, Bellman found a very similar algorithm to estimate the optimal *state-action values*, generally called *Q-values* (quality values). The optimal Q-value of the state-action pair (s, a) , noted $Q^*(s, a)$, is the sum of discounted future rewards the agent can expect on average after it reaches the state s and chooses action a , but before it sees the outcome of this action, assuming it acts optimally after that action.

Let's look at how it works. Once again, you start by initializing all the Q-value estimates to zero, then you update them using the *Q-value iteration* algorithm (see [Equation 18-3](#)).

Equation 18-3. Q-value iteration algorithm

$$Q_{k+1}(s, a) \leftarrow \sum s' T(s, a, s') [R(s, a, s') + \gamma \cdot \max a' Q_k(s', a')] \\ \text{for all } (s, a)$$

Once you have the optimal Q-values, defining the optimal policy, noted $\pi^*(s)$, is trivial; when the agent is in state s , it should choose the action with the highest Q-value for that state: $\pi^*(s) = \arg\max_a Q^*(s, a)$.

Let's apply this algorithm to the MDP represented in [Figure 18-8](#). First, we need to define the MDP:

```
transition_probabilities = [ # shape=[s, a, s']  
    [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],  
    [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],  
    [None, [0.8, 0.1, 0.1], None]  
]  
rewards = [ # shape=[s, a, s']  
    [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],  
    [[0, 0, 0], [0, 0, 0], [0, 0, -50]],  
    [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]  
]  
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

For example, to know the transition probability of going from s_2 to s_0 after playing action a_1 , we will look up `transition_probabilities[2][1][0]` (which is 0.8). Similarly, to get the corresponding reward, we will look up `rewards[2][1][0]` (which is +40). And to get the list of possible actions in s_2 , we will look up `possible_actions[2]` (in this case, only action a_1 is possible). Next, we must initialize all the Q-values to zero (except for the impossible actions, for which we set the Q-values to $-\infty$):

```
Q_values = np.full((3, 3), -np.inf) # -np.inf for impossible actions
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0 # for all possible actions
```

Now let's run the Q-value iteration algorithm. It applies [Equation 18-3](#) repeatedly, to all Q-values, for every state and every possible action:

```
gamma = 0.90 # the discount factor

for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                transition_probabilities[s][a][sp]
                * (rewards[s][a][sp] + gamma * Q_prev[sp].max())
            for sp in range(3)])
```

That's it! The resulting Q-values look like this:

```
>>> Q_values
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.          , -inf, -4.87971488],
       [-inf, 50.13365013, -inf]])
```

For example, when the agent is in state s_0 and it chooses action a_1 , the expected sum of discounted future rewards is approximately 17.0.

For each state, we can find the action that has the highest Q-value:

```
>>> Q_values.argmax(axis=1) # optimal action for each state
array([0, 0, 1])
```

This gives us the optimal policy for this MDP when using a discount factor of 0.90: in state s_0 choose action a_0 , in state s_1 choose action a_0 (i.e., stay put), and in state s_2 choose action a_1 (the only possible action). Interestingly, if we increase the discount factor to 0.95, the optimal policy changes: in state s_1 the best action becomes a_2 (go through the fire!). This makes sense because the more you value future rewards, the more you are willing to put up with some pain now for the promise of future bliss.

Temporal Difference Learning

Reinforcement learning problems with discrete actions can often be modeled as Markov decision processes, but the agent initially has no idea what the transition probabilities are (it does not know $T(s, a, s')$), and it does not know what the rewards are going to be either (it does not know $R(s, a, s')$). It must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities.

The *temporal difference (TD) learning* algorithm is very similar to the Q-value iteration algorithm, but tweaked to take into account the fact that the agent has only partial knowledge of the MDP. In general we assume that the agent initially knows only the possible states and actions, and nothing more. The agent uses an *exploration policy*—for example, a purely random policy—to explore the MDP, and as it progresses, the TD learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed (see [Equation 18-4](#)).

Equation 18-4. TD learning algorithm

$$V_{k+1}(s) \leftarrow (1-\alpha) V_k(s) + \alpha r + \gamma \cdot V_k(s') \text{ or, equivalently: } V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s') \text{ with } \delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k(s)$$

In this equation:

- α is the learning rate (e.g., 0.01).
- $r + \gamma \cdot V_k(s')$ is called the *TD target*.
- $\delta_k(s, r, s')$ is called the *TD error*.

A more concise way of writing the first form of this equation is to use the notation $a \leftarrow \alpha b$, which means $a_{k+1} \leftarrow (1 - \alpha) \cdot a_k + \alpha \cdot b_k$. So, the first line of [Equation 18-4](#) can be rewritten like this: $V(s) \leftarrow \alpha r + \gamma \cdot V(s')$.

TIP

TD learning has many similarities with stochastic gradient descent, including the fact that it handles one sample at a time. Moreover, just like SGD, it can only truly converge if you gradually reduce the learning rate; otherwise, it will keep bouncing around the optimum Q-values.

For each state s , this algorithm keeps track of a running average of the immediate rewards the agent gets upon leaving that state, plus the rewards it expects to get later, assuming it acts optimally.

Q-Learning

Similarly, the Q-learning algorithm is an adaptation of the Q-value iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown (see [Equation 18-5](#)). Q-learning works by watching an agent play (e.g., randomly) and gradually improving its estimates of the Q-values. Once it has accurate Q-value estimates (or close enough), then the optimal policy is just choosing the action that has the highest Q-value (i.e., the greedy policy).

Equation 18-5. Q-learning algorithm

$$Q(s,a) \leftarrow \alpha r + \gamma \max_{a'} Q(s', a')$$

For each state-action pair (s, a) , this algorithm keeps track of a running average of the rewards r the agent gets upon leaving the state s with action a , plus the sum of discounted future rewards it expects to get. To estimate this sum, we take the maximum of the Q-value estimates for the next state s' , since we assume that the target policy will act optimally from then on.

Let's implement the Q-learning algorithm. First, we will need to make an agent explore the environment. For this, we need a step function so that the agent can execute one action and get the resulting state and reward:

```
def step(state, action):
    probas = transition_probabilities[state][action]
    next_state = np.random.choice([0, 1, 2], p=probas)
    reward = rewards[state][action][next_state]
    return next_state, reward
```

Now let's implement the agent's exploration policy. Since the state space is pretty small, a simple random policy will be sufficient. If we run the algorithm for long enough, the agent will visit every state many times, and it will also try every possible action many times:

```
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

Next, after we initialize the Q-values just like earlier, we are ready to run the Q-learning algorithm with learning rate decay (using power scheduling, introduced in [Chapter 11](#)):

```

alpha0 = 0.05 # initial learning rate
decay = 0.005 # learning rate decay
gamma = 0.90 # discount factor
state = 0 # initial state

for iteration in range(10_000):
    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = Q_values[next_state].max() # greedy policy at the next step
    alpha = alpha0 / (1 + iteration * decay)
    Q_values[state, action] *= 1 - alpha
    Q_values[state, action] += alpha * (reward + gamma * next_value)
    state = next_state

```

This algorithm will converge to the optimal Q-values, but it will take many iterations, and possibly quite a lot of hyperparameter tuning. As you can see in [Figure 18-9](#), the Q-value iteration algorithm (left) converges very quickly, in fewer than 20 iterations, while the Q-learning algorithm (right) takes about 8,000 iterations to converge. Obviously, not knowing the transition probabilities or the rewards makes finding the optimal policy significantly harder!

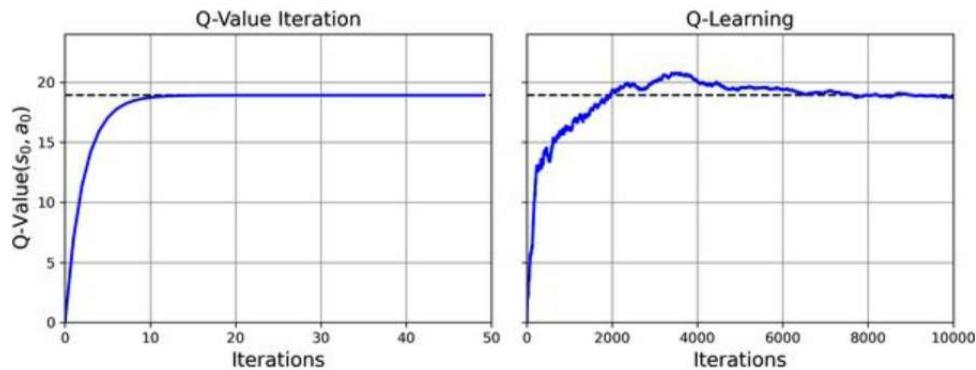


Figure 18-9. Learning curve of the Q-value iteration algorithm versus the Q-learning algorithm

The Q-learning algorithm is called an *off-policy* algorithm because the policy being trained is not necessarily the one used during training. For example, in

the code we just ran, the policy being executed (the exploration policy) was completely random, while the policy being trained was never used. After training, the optimal policy corresponds to systematically choosing the action with the highest Q-value. Conversely, the policy gradients algorithm is an *on-policy* algorithm: it explores the world using the policy being trained. It is somewhat surprising that Q-learning is capable of learning the optimal policy by just watching an agent act randomly. Imagine learning to play golf when your teacher is a blindfolded monkey. Can we do better?

Exploration Policies

Of course, Q-learning can work only if the exploration policy explores the MDP thoroughly enough. Although a purely random policy is guaranteed to eventually visit every state and every transition many times, it may take an extremely long time to do so. Therefore, a better option is to use the *ϵ -greedy policy* (ϵ is epsilon): at each step it acts randomly with probability ϵ , or greedily with probability $1-\epsilon$ (i.e., choosing the action with the highest Q-value). The advantage of the ϵ -greedy policy (compared to a completely random policy) is that it will spend more and more time exploring the interesting parts of the environment, as the Q-value estimates get better and better, while still spending some time visiting unknown regions of the MDP. It is quite common to start with a high value for ϵ (e.g., 1.0) and then gradually reduce it (e.g., down to 0.05).

Alternatively, rather than relying only on chance for exploration, another approach is to encourage the exploration policy to try actions that it has not tried much before. This can be implemented as a bonus added to the Q-value estimates, as shown in [Equation 18-6](#).

Equation 18-6. Q-learning using an exploration function

$$Q(s, a) \leftarrow \alpha r + \gamma \max_{a'} f(Q(s', a'), N(s', a'))$$

In this equation:

- $N(s', a')$ counts the number of times the action a' was chosen in state s' .
- $f(Q, N)$ is an *exploration function*, such as $f(Q, N) = Q + \kappa/(1 + N)$, where κ is a curiosity hyperparameter that measures how much the agent is attracted to the unknown.

Approximate Q-Learning and Deep Q-Learning

The main problem with Q-learning is that it does not scale well to large (or even medium) MDPs with many states and actions. For example, suppose you wanted to use Q-learning to train an agent to play *Ms. Pac-Man* (see [Figure 18-1](#)). There are about 150 pellets that Ms. Pac-Man can eat, each of which can be present or absent (i.e., already eaten). So, the number of possible states is greater than $2^{150} \approx 10^{45}$. And if you add all the possible combinations of positions for all the ghosts and Ms. Pac-Man, the number of possible states becomes larger than the number of atoms in our planet, so there's absolutely no way you can keep track of an estimate for every single Q-value.

The solution is to find a function $Q_{\theta}(s, a)$ that approximates the Q-value of any state-action pair (s, a) using a manageable number of parameters (given by the parameter vector θ). This is called *approximate Q-learning*. For years it was recommended to use linear combinations of handcrafted features extracted from the state (e.g., the distances of the closest ghosts, their directions, and so on) to estimate Q-values, but in 2013, [DeepMind](#) showed that using deep neural networks can work much better, especially for complex problems, and it does not require any feature engineering. A DNN used to estimate Q-values is called a *deep Q-network* (DQN), and using a DQN for approximate Q-learning is called *deep Q-learning*.

Now, how can we train a DQN? Well, consider the approximate Q-value computed by the DQN for a given state-action pair (s, a) . Thanks to Bellman, we know we want this approximate Q-value to be as close as possible to the reward r that we actually observe after playing action a in state s , plus the discounted value of playing optimally from then on. To estimate this sum of future discounted rewards, we can just execute the DQN on the next state s' , for all possible actions a' . We get an approximate future Q-value for each possible action. We then pick the highest (since we assume we will be playing optimally) and discount it, and this gives us an estimate of the sum of future discounted rewards. By summing the reward r and the future discounted value estimate, we get a target Q-value $y(s, a)$ for the state-action

pair (s, a) , as shown in [Equation 18-7](#).

Equation 18-7. Target Q-value

$$y(s,a) = r + \gamma \cdot \max_{a'} Q_\theta(s', a')$$

With this target Q-value, we can run a training step using any gradient descent algorithm. Specifically, we generally try to minimize the squared error between the estimated Q-value $Q_\theta(s, a)$ and the target Q-value $y(s, a)$, or the Huber loss to reduce the algorithm's sensitivity to large errors. And that's the deep Q-learning algorithm! Let's see how to implement it to solve the CartPole environment.

Implementing Deep Q-Learning

The first thing we need is a deep Q-network. In theory, we need a neural net that takes a state-action pair as input, and outputs an approximate Q-value. However, in practice it's much more efficient to use a neural net that takes only a state as input, and outputs one approximate Q-value for each possible action. To solve the CartPole environment, we do not need a very complicated neural net; a couple of hidden layers will do:

```
input_shape = [4] # == env.observation_space.shape
n_outputs = 2 # == env.action_space.n

model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    tf.keras.layers.Dense(32, activation="elu"),
    tf.keras.layers.Dense(n_outputs)
])
```

To select an action using this DQN, we pick the action with the largest predicted Q-value. To ensure that the agent explores the environment, we will use an ϵ -greedy policy (i.e., we will choose a random action with probability ϵ):

```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(n_outputs) # random action
    else:
        Q_values = model.predict(state[np.newaxis], verbose=0)[0]
        return Q_values.argmax() # optimal action according to the DQN
```

Instead of training the DQN based only on the latest experiences, we will store all experiences in a *replay buffer* (or *replay memory*), and we will sample a random training batch from it at each training iteration. This helps reduce the correlations between the experiences in a training batch, which tremendously helps training. For this, we will just use a double-ended queue (deque):

```
from collections import deque  
  
replay_buffer = deque(maxlen=2000)
```

TIP

A *deque* is a queue elements can be efficiently added to or removed from on both ends. Inserting and deleting items from the ends of the queue is very fast, but random access can be slow when the queue gets long. If you need a very large replay buffer, you should use a circular buffer instead (see the notebook for an implementation), or check out DeepMind's [Reverb library](#).

Each experience will be composed of six elements: a state s , the action a that the agent took, the resulting reward r , the next state s' it reached, a Boolean indicating whether the episode ended at that point (`done`), and finally another Boolean indicating whether the episode was truncated at that point. We will need a small function to sample a random batch of experiences from the replay buffer. It will return six NumPy arrays corresponding to the six experience elements:

```
def sample_experiences(batch_size):  
    indices = np.random.randint(len(replay_buffer), size=batch_size)  
    batch = [replay_buffer[index] for index in indices]  
    return [  
        np.array([experience[field_index] for experience in batch])  
        for field_index in range(6)  
    ] # [states, actions, rewards, next_states, dones, truncated]
```

Let's also create a function that will play a single step using the ϵ -greedy policy, then store the resulting experience in the replay buffer:

```
def play_one_step(env, state, epsilon):  
    action = epsilon_greedy_policy(state, epsilon)  
    next_state, reward, done, truncated, info = env.step(action)  
    replay_buffer.append((state, action, reward, next_state, done, truncated))  
    return next_state, reward, done, truncated, info
```

Finally, let's create one last function that will sample a batch of experiences

from the replay buffer and train the DQN by performing a single gradient descent step on this batch:

```
batch_size = 32
discount_factor = 0.95
optimizer = tf.keras.optimizers.Nadam(learning_rate=1e-2)
loss_fn = tf.keras.losses.mean_squared_error

def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones, truncateds = experiences
    next_Q_values = model.predict(next_states, verbose=0)
    max_next_Q_values = next_Q_values.max(axis=1)
    runs = 1.0 - (dones | truncateds) # episode is not done or truncated
    target_Q_values = rewards + runs * discount_factor * max_next_Q_values
    target_Q_values = target_Q_values.reshape(-1, 1)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Here's what's happening in this code:

- First we define some hyperparameters, and we create the optimizer and the loss function.
- Then we create the `training_step()` function. It starts by sampling a batch of experiences, then it uses the DQN to predict the Q-value for each possible action in each experience's next state. Since we assume that the agent will be playing optimally, we only keep the maximum Q-value for each next state. Next, we use [Equation 18-7](#) to compute the target Q-value for each experience's state-action pair.
- We want to use the DQN to compute the Q-value for each experienced state-action pair, but the DQN will also output the Q-values for the other possible actions, not just for the action that was actually chosen by the agent. So, we need to mask out all the Q-values we do not need. The

`tf.one_hot()` function makes it possible to convert an array of action indices into such a mask. For example, if the first three experiences contain actions 1, 1, 0, respectively, then the mask will start with `[[0, 1], [0, 1], [1, 0], ...]`. We can then multiply the DQN's output with this mask, and this will zero out all the Q-values we do not want. We then sum over axis 1 to get rid of all the zeros, keeping only the Q-values of the experienced state-action pairs. This gives us the `Q_values` tensor, containing one predicted Q-value for each experience in the batch.

- Next, we compute the loss: it is the mean squared error between the target and predicted Q-values for the experienced state-action pairs.
- Finally, we perform a gradient descent step to minimize the loss with regard to the model's trainable variables.

This was the hardest part. Now training the model is straightforward:

```
for episode in range(600):
    obs, info = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, truncated, info = play_one_step(env, obs, epsilon)
        if done or truncated:
            break
    if episode > 50:
        training_step(batch_size)
```

We run 600 episodes, each for a maximum of 200 steps. At each step, we first compute the epsilon value for the ϵ -greedy policy: it will go from 1 down to 0.01, linearly, in a bit under 500 episodes. Then we call the `play_one_step()` function, which will use the ϵ -greedy policy to pick an action, then execute it and record the experience in the replay buffer. If the episode is done or truncated, we exit the loop. Finally, if we are past episode 50, we call the `training_step()` function to train the model on one batch sampled from the replay buffer. The reason we play many episodes without training is to give the replay buffer some time to fill up (if we don't wait enough, then there will not be enough diversity in the replay buffer). And

that's it: we just implemented the Deep Q-learning algorithm!

Figure 18-10 shows the total rewards the agent got during each episode.

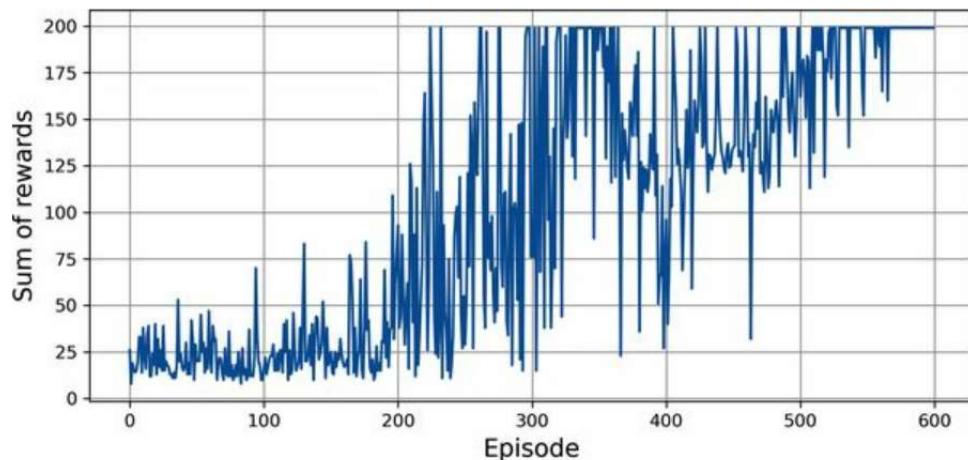


Figure 18-10. Learning curve of the deep Q-learning algorithm

As you can see, the algorithm took a while to start learning anything, in part because ϵ was very high at the beginning. Then its progress was erratic: it first reached the max reward around episode 220, but it immediately dropped, then bounced up and down a few times, and soon after it looked like it had finally stabilized near the max reward, at around episode 320, its score again dropped down dramatically. This is called *catastrophic forgetting*, and it is one of the big problems facing virtually all RL algorithms: as the agent explores the environment, it updates its policy, but what it learns in one part of the environment may break what it learned earlier in other parts of the environment. The experiences are quite correlated, and the learning environment keeps changing—this is not ideal for gradient descent! If you increase the size of the replay buffer, the algorithm will be less subject to this problem. Tuning the learning rate may also help. But the truth is, reinforcement learning is hard: training is often unstable, and you may need to try many hyperparameter values and random seeds before you find a combination that works well. For example, if you try changing the activation function from "elu" to "relu", the performance will be much lower.

NOTE

Reinforcement learning is notoriously difficult, largely because of the training instabilities and the huge sensitivity to the choice of hyperparameter values and random seeds.¹³ As the researcher Andrej Karpathy put it, “[Supervised learning] wants to work. [...] RL must be forced to work”. You will need time, patience, perseverance, and perhaps a bit of luck too. This is a major reason RL is not as widely adopted as regular deep learning (e.g., convolutional nets). But there are a few real-world applications, beyond AlphaGo and Atari games: for example, Google uses RL to optimize its datacenter costs, and it is used in some robotics applications, for hyperparameter tuning, and in recommender systems.

You might wonder why we didn’t plot the loss. It turns out that loss is a poor indicator of the model’s performance. The loss might go down, yet the agent might perform worse (e.g., this can happen when the agent gets stuck in one small region of the environment, and the DQN starts overfitting this region). Conversely, the loss could go up, yet the agent might perform better (e.g., if the DQN was underestimating the Q-values and it starts correctly increasing its predictions, the agent will likely perform better, getting more rewards, but the loss might increase because the DQN also sets the targets, which will be larger too). So, it’s preferable to plot the rewards.

The basic deep Q-learning algorithm we’ve been using so far would be too unstable to learn to play Atari games. So how did DeepMind do it? Well, they tweaked the algorithm!

Deep Q-Learning Variants

Let's look at a few variants of the deep Q-learning algorithm that can stabilize and speed up training.

Fixed Q-value Targets

In the basic deep Q-learning algorithm, the model is used both to make predictions and to set its own targets. This can lead to a situation analogous to a dog chasing its own tail. This feedback loop can make the network unstable: it can diverge, oscillate, freeze, and so on. To solve this problem, in their 2013 paper the DeepMind researchers used two DQNs instead of one: the first is the *online model*, which learns at each step and is used to move the agent around, and the other is the *target model* used only to define the targets. The target model is just a clone of the online model:

```
target = tf.keras.models.clone_model(model) # clone the model's architecture  
target.set_weights(model.get_weights()) # copy the weights
```

Then, in the `training_step()` function, we just need to change one line to use the target model instead of the online model when computing the Q-values of the next states:

```
next_Q_values = target.predict(next_states, verbose=0)
```

Finally, in the training loop, we must copy the weights of the online model to the target model, at regular intervals (e.g., every 50 episodes):

```
if episode % 50 == 0:  
    target.set_weights(model.get_weights())
```

Since the target model is updated much less often than the online model, the Q-value targets are more stable, the feedback loop we discussed earlier is dampened, and its effects are less severe. This approach was one of the DeepMind researchers' main contributions in their 2013 paper, allowing agents to learn to play Atari games from raw pixels. To stabilize training, they used a tiny learning rate of 0.00025, they updated the target model only every 10,000 steps (instead of 50), and they used a very large replay buffer of 1 million experiences. They decreased epsilon very slowly, from 1 to 0.1 in 1 million steps, and they let the algorithm run for 50 million steps. Moreover,

their DQN was a deep convolutional net.

Now let's take a look at another DQN variant that managed to beat the state of the art once more.

Double DQN

In a [2015 paper](#), ¹⁴ DeepMind researchers tweaked their DQN algorithm, increasing its performance and somewhat stabilizing training. They called this variant *double DQN*. The update was based on the observation that the target network is prone to overestimating Q-values. Indeed, suppose all actions are equally good: the Q-values estimated by the target model should be identical, but since they are approximations, some may be slightly greater than others, by pure chance. The target model will always select the largest Q-value, which will be slightly greater than the mean Q-value, most likely overestimating the true Q-value (a bit like counting the height of the tallest random wave when measuring the depth of a pool). To fix this, the researchers proposed using the online model instead of the target model when selecting the best actions for the next states, and using the target model only to estimate the Q-values for these best actions. Here is the updated `training_step()` function:

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones, truncateds = experiences
    next_Q_values = model.predict(next_states, verbose=0) # ≠ target.predict()
    best_next_actions = next_Q_values.argmax(axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    max_next_Q_values = (target.predict(next_states, verbose=0) * next_mask
        ).sum(axis=1)
    [...] # the rest is the same as earlier
```

Just a few months later, another improvement to the DQN algorithm was proposed; we'll look at that next.

Prioritized Experience Replay

Instead of sampling experiences *uniformly* from the replay buffer, why not sample important experiences more frequently? This idea is called *importance sampling* (IS) or *prioritized experience replay* (PER), and it was introduced in a [2015 paper¹⁵](#) by DeepMind researchers (once again!).

More specifically, experiences are considered “important” if they are likely to lead to fast learning progress. But how can we estimate this? One reasonable approach is to measure the magnitude of the TD error $\delta = r + \gamma \cdot V(s') - V(s)$. A large TD error indicates that a transition (s, a, s') is very surprising, and thus probably worth learning from. ¹⁶ When an experience is recorded in the replay buffer, its priority is set to a very large value, to ensure that it gets sampled at least once. However, once it is sampled (and every time it is sampled), the TD error δ is computed, and this experience’s priority is set to $p = |\delta|$ (plus a small constant to ensure that every experience has a nonzero probability of being sampled). The probability P of sampling an experience with priority p is proportional to p^ζ , where ζ is a hyperparameter that controls how greedy we want importance sampling to be: when $\zeta = 0$, we just get uniform sampling, and when $\zeta = 1$, we get full-blown importance sampling. In the paper, the authors used $\zeta = 0.6$, but the optimal value will depend on the task.

There’s one catch, though: since the samples will be biased toward important experiences, we must compensate for this bias during training by downweighting the experiences according to their importance, or else the model will just overfit the important experiences. To be clear, we want important experiences to be sampled more often, but this also means we must give them a lower weight during training. To do this, we define each experience’s training weight as $w = (n P)^{-\beta}$, where n is the number of experiences in the replay buffer, and β is a hyperparameter that controls how much we want to compensate for the importance sampling bias (0 means not at all, while 1 means entirely). In the paper, the authors used $\beta = 0.4$ at the beginning of training and linearly increased it to $\beta = 1$ by the end of training. Again, the optimal value will depend on the task, but if you increase one, you

will usually want to increase the other as well.

Now let's look at one last important variant of the DQN algorithm.

Dueling DQN

The *dueling DQN* algorithm (DDQN, not to be confused with double DQN, although both techniques can easily be combined) was introduced in yet another 2015 paper¹⁷ by DeepMind researchers. To understand how it works, we must first note that the Q-value of a state-action pair (s, a) can be expressed as $Q(s, a) = V(s) + A(s, a)$, where $V(s)$ is the value of state s and $A(s, a)$ is the *advantage* of taking the action a in state s , compared to all other possible actions in that state. Moreover, the value of a state is equal to the Q-value of the best action a^* for that state (since we assume the optimal policy will pick the best action), so $V(s) = Q(s, a^*)$, which implies that $A(s, a^*) = 0$. In a dueling DQN, the model estimates both the value of the state and the advantage of each possible action. Since the best action should have an advantage of 0, the model subtracts the maximum predicted advantage from all predicted advantages. Here is a simple DDQN model, implemented using the functional API:

```
input_states = tf.keras.layers.Input(shape=[4])
hidden1 = tf.keras.layers.Dense(32, activation="elu")(input_states)
hidden2 = tf.keras.layers.Dense(32, activation="elu")(hidden1)
state_values = tf.keras.layers.Dense(1)(hidden2)
raw_advantages = tf.keras.layers.Dense(n_outputs)(hidden2)
advantages = raw_advantages - tf.reduce_max(raw_advantages, axis=1,
                                             keepdims=True)
Q_values = state_values + advantages
model = tf.keras.Model(inputs=[input_states], outputs=[Q_values])
```

The rest of the algorithm is just the same as earlier. In fact, you can build a double dueling DQN and combine it with prioritized experience replay! More generally, many RL techniques can be combined, as DeepMind demonstrated in a 2017 paper:¹⁸ the paper's authors combined six different techniques into an agent called *Rainbow*, which largely outperformed the state of the art.

As you can see, deep reinforcement learning is a fast-growing field and there's much more to discover!

Overview of Some Popular RL Algorithms

Before we close this chapter, let's take a brief look at a few other popular algorithms:

*AlphaGo*¹⁹

AlphaGo uses a variant of *Monte Carlo tree search* (MCTS) based on deep neural networks to beat human champions at the game of Go. MCTS was invented in 1949 by Nicholas Metropolis and Stanislaw Ulam. It selects the best move after running many simulations, repeatedly exploring the search tree starting from the current position, and spending more time on the most promising branches. When it reaches a node that it hasn't visited before, it plays randomly until the game ends, and updates its estimates for each visited node (excluding the random moves), increasing or decreasing each estimate depending on the final outcome. AlphaGo is based on the same principle, but it uses a policy network to select moves, rather than playing randomly. This policy net is trained using policy gradients. The original algorithm involved three more neural networks, and was more complicated, but it was simplified in the [AlphaGo Zero paper](#),²⁰ which uses a single neural network to both select moves and evaluate game states. The [AlphaZero paper](#)²¹ generalized this algorithm, making it capable of tackling not only the game of Go, but also chess and shogi (Japanese chess). Lastly, the [MuZero paper](#)²² continued to improve upon this algorithm, outperforming the previous iterations even though the agent starts out without even knowing the rules of the game!

Actor-critic algorithms

Actor-critics are a family of RL algorithms that combine policy gradients with deep Q-networks. An actor-critic agent contains two neural networks: a policy net and a DQN. The DQN is trained normally, by learning from the agent's experiences. The policy net learns differently (and much faster) than in regular PG: instead of estimating the value of

each action by going through multiple episodes, then summing the future discounted rewards for each action, and finally normalizing them, the agent (actor) relies on the action values estimated by the DQN (critic). It's a bit like an athlete (the agent) learning with the help of a coach (the DQN).

Asynchronous advantage actor-critic (A3C)²³

This is an important actor-critic variant introduced by DeepMind researchers in 2016 where multiple agents learn in parallel, exploring different copies of the environment. At regular intervals, but asynchronously (hence the name), each agent pushes some weight updates to a master network, then it pulls the latest weights from that network. Each agent thus contributes to improving the master network and benefits from what the other agents have learned. Moreover, instead of estimating the Q-values, the DQN estimates the advantage of each action (hence the second A in the name), which stabilizes training.

Advantage actor-critic (A2C)

A2C is a variant of the A3C algorithm that removes the asynchronicity. All model updates are synchronous, so gradient updates are performed over larger batches, which allows the model to better utilize the power of the GPU.

Soft actor-critic (SAC)²⁴

SAC is an actor-critic variant proposed in 2018 by Tuomas Haarnoja and other UC Berkeley researchers. It learns not only rewards, but also to maximize the entropy of its actions. In other words, it tries to be as unpredictable as possible while still getting as many rewards as possible. This encourages the agent to explore the environment, which speeds up training, and makes it less likely to repeatedly execute the same action when the DQN produces imperfect estimates. This algorithm has demonstrated an amazing sample efficiency (contrary to all the previous algorithms, which learn very slowly).

Proximal policy optimization (PPO)²⁵

This algorithm by John Schulman and other OpenAI researchers is based on A2C, but it clips the loss function to avoid excessively large weight updates (which often lead to training instabilities). PPO is a simplification of the previous *trust region policy optimization²⁶* (TRPO) algorithm, also by OpenAI. OpenAI made the news in April 2019 with its AI called OpenAI Five, based on the PPO algorithm, which defeated the world champions at the multiplayer game *Dota 2*.

Curiosity-based exploration²⁷

A recurring problem in RL is the sparsity of the rewards, which makes learning very slow and inefficient. Deepak Pathak and other UC Berkeley researchers have proposed an exciting way to tackle this issue: why not ignore the rewards, and just make the agent extremely curious to explore the environment? The rewards thus become intrinsic to the agent, rather than coming from the environment. Similarly, stimulating curiosity in a child is more likely to give good results than purely rewarding the child for getting good grades. How does this work? The agent continuously tries to predict the outcome of its actions, and it seeks situations where the outcome does not match its predictions. In other words, it wants to be surprised. If the outcome is predictable (boring), it goes elsewhere. However, if the outcome is unpredictable but the agent notices that it has no control over it, it also gets bored after a while. With only curiosity, the authors succeeded in training an agent at many video games: even though the agent gets no penalty for losing, the game starts over, which is boring so it learns to avoid it.

Open-ended learning (OEL)

The objective of OEL is to train agents capable of endlessly learning new and interesting tasks, typically generated procedurally. We're not there yet, but there has been some amazing progress over the last few years. For example, a [2019 paper²⁸](#) by a team of researchers from Uber AI introduced the *POET algorithm*, which generates multiple simulated 2D

environments with bumps and holes and trains one agent per environment: the agent's goal is to walk as fast as possible while avoiding the obstacles. The algorithm starts out with simple environments, but they gradually get harder over time: this is called *curriculum learning*.

Moreover, although each agent is only trained within one environment, it must regularly compete against other agents, across all environments. In each environment, the winner is copied over and it replaces the agent that was there before. This way, knowledge is regularly transferred across environments, and the most adaptable agents are selected. In the end, the agents are much better walkers than agents trained on a single task, and they can tackle much harder environments. Of course, this principle can be applied to other environments and tasks as well. If you're interested in OEL, make sure to check out the [Enhanced POET paper](#),²⁹ as well as DeepMind's [2021 paper](#)³⁰ on this topic.

TIP

If you'd like to learn more about reinforcement learning, check out the book [*Reinforcement Learning*](#) by Phil Winder (O'Reilly).

We covered many topics in this chapter: policy gradients, Markov chains, Markov decision processes, Q-learning, approximate Q-learning, and deep Q-learning and its main variants (fixed Q-value targets, double DQN, dueling DQN, and prioritized experience replay), and finally we took a quick look at a few other popular algorithms. Reinforcement learning is a huge and exciting field, with new ideas and algorithms popping out every day, so I hope this chapter sparked your curiosity: there is a whole world to explore!

Exercises

1. How would you define reinforcement learning? How is it different from regular supervised or unsupervised learning?
2. Can you think of three possible applications of RL that were not mentioned in this chapter? For each of them, what is the environment? What is the agent? What are some possible actions? What are the rewards?
3. What is the discount factor? Can the optimal policy change if you modify the discount factor?
4. How do you measure the performance of a reinforcement learning agent?
5. What is the credit assignment problem? When does it occur? How can you alleviate it?
6. What is the point of using a replay buffer?
7. What is an off-policy RL algorithm?
8. Use policy gradients to solve OpenAI Gym's LunarLander-v2 environment.
9. Use a double dueling DQN to train an agent that can achieve a superhuman level at the famous Atari *Breakout* game ("ALE/Breakout-v5"). The observations are images. To simplify the task, you should convert them to grayscale (i.e., average over the channels axis) then crop them and downsample them, so they're just large enough to play, but not more. An individual image does not tell you which way the ball and the paddles are going, so you should merge two or three consecutive images to form each state. Lastly, the DQN should be composed mostly of convolutional layers.
10. If you have about \$100 to spare, you can purchase a Raspberry Pi 3 plus

some cheap robotics components, install TensorFlow on the Pi, and go wild! For an example, check out this [fun post](#) by Lukas Biewald, or take a look at GoPiGo or BrickPi. Start with simple goals, like making the robot turn around to find the brightest angle (if it has a light sensor) or the closest object (if it has a sonar sensor), and move in that direction. Then you can start using deep learning: for example, if the robot has a camera, you can try to implement an object detection algorithm so it detects people and moves toward them. You can also try to use RL to make the agent learn on its own how to use the motors to achieve that goal. Have fun!

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

-
- 1 For more details, be sure to check out Richard Sutton and Andrew Barto's book on RL, *Reinforcement Learning: An Introduction* (MIT Press).
 - 2 Volodymyr Mnih et al., "Playing Atari with Deep Reinforcement Learning", arXiv preprint arXiv:1312.5602 (2013).
 - 3 Volodymyr Mnih et al., "Human-Level Control Through Deep Reinforcement Learning", *Nature* 518 (2015): 529–533.
 - 4 Check out the videos of DeepMind's system learning to play *Space Invaders*, *Breakout*, and other video games at <https://homl.info/dqn3>.
 - 5 Images (a), (d), and (e) are in the public domain. Image (b) is a screenshot from the *Ms. Pac-Man* game, copyright Atari (fair use in this chapter). Image (c) is reproduced from Wikipedia; it was created by user Stevertigo and released under [Creative Commons BY-SA 2.0](#).
 - 6 It is often better to give the poor performers a slight chance of survival, to preserve some diversity in the "gene pool".
 - 7 If there is a single parent, this is called *asexual reproduction*. With two (or more) parents, it is called *sexual reproduction*. An offspring's genome (in this case a set of policy parameters) is randomly composed of parts of its parents' genomes.
 - 8 One interesting example of a genetic algorithm used for reinforcement learning is the *NeuroEvolution of Augmenting Topologies* (NEAT) algorithm.
 - 9 This is called *gradient ascent*. It's just like gradient descent, but in the opposite direction: maximizing instead of minimizing.
 - 10 OpenAI is an artificial intelligence research company, funded in part by Elon Musk. Its stated goal is to promote and develop friendly AIs that will benefit humanity (rather than exterminate

it).

- 11 Ronald J. Williams, “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning”, *Machine Learning* 8 (1992) : 229–256.
- 12 Richard Bellman, “A Markovian Decision Process”, *Journal of Mathematics and Mechanics* 6, no. 5 (1957): 679–684.
- 13 A great 2018 post by Alex Irpan nicely lays out RL’s biggest difficulties and limitations.
- 14 Hado van Hasselt et al., “Deep Reinforcement Learning with Double Q-Learning”, *Proceedings of the 30th AAAI Conference on Artificial Intelligence* (2015): 2094–2100.
- 15 Tom Schaul et al., “Prioritized Experience Replay”, arXiv preprint arXiv:1511.05952 (2015).
- 16 It could also just be that the rewards are noisy, in which case there are better methods for estimating an experience’s importance (see the paper for some examples).
- 17 Ziyu Wang et al., “Dueling Network Architectures for Deep Reinforcement Learning”, arXiv preprint arXiv:1511.06581 (2015).
- 18 Matteo Hessel et al., “Rainbow: Combining Improvements in Deep Reinforcement Learning”, arXiv preprint arXiv:1710.02298 (2017): 3215–3222.
- 19 David Silver et al., “Mastering the Game of Go with Deep Neural Networks and Tree Search”, *Nature* 529 (2016): 484–489.
- 20 David Silver et al., “Mastering the Game of Go Without Human Knowledge”, *Nature* 550 (2017): 354–359.
- 21 David Silver et al., “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”, arXiv preprint arXiv:1712.01815.
- 22 Julian Schrittwieser et al., “Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model”, arXiv preprint arXiv:1911.08265 (2019).
- 23 Volodymyr Mnih et al., “Asynchronous Methods for Deep Reinforcement Learning”, *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1928–1937.
- 24 Tuomas Haarnoja et al., “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”, *Proceedings of the 35th International Conference on Machine Learning* (2018): 1856–1865.
- 25 John Schulman et al., “Proximal Policy Optimization Algorithms”, arXiv preprint arXiv:1707.06347 (2017).
- 26 John Schulman et al., “Trust Region Policy Optimization”, *Proceedings of the 32nd International Conference on Machine Learning* (2015): 1889–1897.
- 27 Deepak Pathak et al., “Curiosity-Driven Exploration by Self-Supervised Prediction”, *Proceedings of the 34th International Conference on Machine Learning* (2017): 2778–2787.
- 28 Rui Wang et al., “Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions”, arXiv preprint arXiv:1901.01753 (2019).

29 Rui Wang et al., “Enhanced POET: Open-Ended Reinforcement Learning Through Unbounded Invention of Learning Challenges and Their Solutions”, arXiv preprint arXiv:2003.08536 (2020).

30 Open-Ended Learning Team et al., “Open-Ended Learning Leads to Generally Capable Agents”, arXiv preprint arXiv:2107.12808 (2021).

Chapter 19. Training and Deploying TensorFlow Models at Scale

Once you have a beautiful model that makes amazing predictions, what do you do with it? Well, you need to put it in production! This could be as simple as running the model on a batch of data, and perhaps writing a script that runs this model every night. However, it is often much more involved. Various parts of your infrastructure may need to use this model on live data, in which case you will probably want to wrap your model in a web service: this way, any part of your infrastructure can query the model at any time using a simple REST API (or some other protocol), as we discussed in [Chapter 2](#). But as time passes, you'll need to regularly retrain your model on fresh data and push the updated version to production. You must handle model versioning, gracefully transition from one model to the next, possibly roll back to the previous model in case of problems, and perhaps run multiple different models in parallel to perform *A/B experiments*. ¹ If your product becomes successful, your service may start to get a large number of queries per second (QPS), and it must scale up to support the load. A great solution to scale up your service, as you will see in this chapter, is to use TF Serving, either on your own hardware infrastructure or via a cloud service such as Google Vertex AI. ² It will take care of efficiently serving your model, handle graceful model transitions, and more. If you use the cloud platform you will also get many extra features, such as powerful monitoring tools.

Moreover, if you have a lot of training data and compute-intensive models, then training time may be prohibitively long. If your product needs to adapt to changes quickly, then a long training time can be a showstopper (e.g., think of a news recommendation system promoting news from last week). Perhaps even more importantly, a long training time will prevent you from experimenting with new ideas. In machine learning (as in many other fields),

it is hard to know in advance which ideas will work, so you should try out as many as possible, as fast as possible. One way to speed up training is to use hardware accelerators such as GPUs or TPUs. To go even faster, you can train a model across multiple machines, each equipped with multiple hardware accelerators. TensorFlow's simple yet powerful distribution strategies API makes this easy, as you will see.

In this chapter we will look at how to deploy models, first using TF Serving, then using Vertex AI. We will also take a quick look at deploying models to mobile apps, embedded devices, and web apps. Then we will discuss how to speed up computations using GPUs and how to train models across multiple devices and servers using the distribution strategies API. Lastly, we will explore how to train models and fine-tune their hyperparameters at scale using Vertex AI. That's a lot of topics to discuss, so let's dive in!

Serving a TensorFlow Model

Once you have trained a TensorFlow model, you can easily use it in any Python code: if it's a Keras model, just call its `predict()` method! But as your infrastructure grows, there comes a point where it is preferable to wrap your model in a small service whose sole role is to make predictions and have the rest of the infrastructure query it (e.g., via a REST or gRPC API). ³ This decouples your model from the rest of the infrastructure, making it possible to easily switch model versions or scale the service up as needed (independently from the rest of your infrastructure), perform A/B experiments, and ensure that all your software components rely on the same model versions. It also simplifies testing and development, and more. You could create your own microservice using any technology you want (e.g., using the Flask library), but why reinvent the wheel when you can just use TF Serving?

Using TensorFlow Serving

TF Serving is a very efficient, battle-tested model server, written in C++. It can sustain a high load, serve multiple versions of your models and watch a model repository to automatically deploy the latest versions, and more (see Figure 19-1).

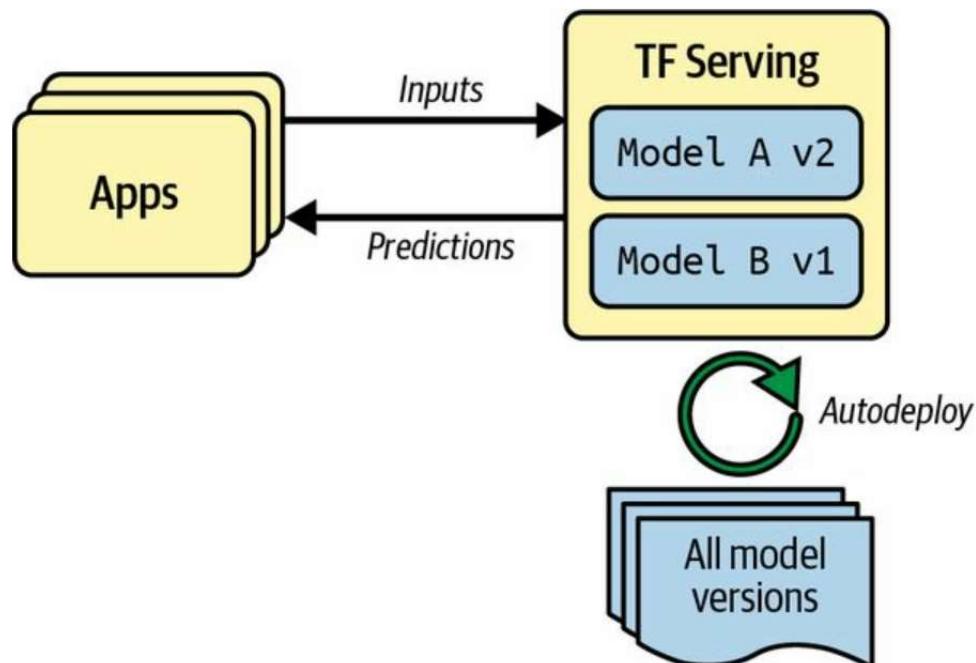


Figure 19-1. TF Serving can serve multiple models and automatically deploy the latest version of each model

So let's suppose you have trained an MNIST model using Keras, and you want to deploy it to TF Serving. The first thing you have to do is export this model to the SavedModel format, introduced in [Chapter 10](#).

Exporting SavedModels

You already know how to save the model: just call `model.save()`. Now to version the model, you just need to create a subdirectory for each model version. Easy!

```
from pathlib import Path
import tensorflow as tf

X_train, X_valid, X_test = [...] # load and split the MNIST dataset
model = [...] # build & train an MNIST model (also handles image preprocessing)

model_name = "my_mnist_model"
model_version = "0001"
model_path = Path(model_name) / model_version
model.save(model_path, save_format="tf")
```

It's usually a good idea to include all the preprocessing layers in the final model you export so that it can ingest data in its natural form once it is deployed to production. This avoids having to take care of preprocessing separately within the application that uses the model. Bundling the preprocessing steps within the model also makes it simpler to update them later on and limits the risk of mismatch between a model and the preprocessing steps it requires.

WARNING

Since a SavedModel saves the computation graph, it can only be used with models that are based exclusively on TensorFlow operations, excluding the `tf.py_function()` operation, which wraps arbitrary Python code.

TensorFlow comes with a small `saved_model_cli` command-line interface to inspect SavedModels. Let use it to inspect our exported model:

```
$ saved_model_cli show --dir my_mnist_model/0001
The given SavedModel contains the following tag-sets:
'serve'
```

What does this output mean? Well, a SavedModel contains one or more *metagraphs*. A metagraph is a computation graph plus some function signature definitions, including their input and output names, types, and shapes. Each metagraph is identified by a set of tags. For example, you may want to have a metagraph containing the full computation graph, including

the training operations: you would typically tag this one as "train". And you might have another metagraph containing a pruned computation graph with only the prediction operations, including some GPU-specific operations: this one might be tagged as "serve", "gpu". You might want to have other metagraphs as well. This can be done using TensorFlow's low-level [SavedModel API](#). However, when you save a Keras model using its `save()` method, it saves a single metagraph tagged as "serve". Let's inspect this "serve" tag set:

```
$ saved_model_cli show --dir 0001/my_mnist_model --tag_set serve
The given SavedModel MetaGraphDef contains SignatureDefs with these keys:
SignatureDef key: "__saved_model_init_op"
SignatureDef key: "serving_default"
```

This metagraph contains two signature definitions: an initialization function called "`__saved_model_init_op`", which you do not need to worry about, and a default serving function called "`serving_default`". When saving a Keras model, the default serving function is the model's `call()` method, which makes predictions, as you already know. Let's get more details about this serving function:

```
$ saved_model_cli show --dir 0001/my_mnist_model --tag_set serve \
--signature_def serving_default
The given SavedModel SignatureDef contains the following input(s):
inputs['flatten_input'] tensor_info:
  dtype: DT_UINT8
  shape: (-1, 28, 28)
  name: serving_default_flatten_input:0
The given SavedModel SignatureDef contains the following output(s):
outputs['dense_1'] tensor_info:
  dtype: DT_FLOAT
  shape: (-1, 10)
  name: StatefulPartitionedCall:0
Method name is: tensorflow/serving/predict
```

Note that the function's input is named "`flatten_input`", and the output is named "`dense_1`". These correspond to the Keras model's input and output layer names. You can also see the type and shape of the input and output data. Looks good!

Now that you have a SavedModel, the next step is to install TF Serving.

Installing and starting TensorFlow Serving

There are many ways to install TF Serving: using the system's package manager, using a Docker image,⁴ installing from source, and more. Since Colab runs on Ubuntu, we can use Ubuntu's apt package manager like this:

```
url = "https://storage.googleapis.com/tensorflow-serving-apt"
src = "stable tensorflow-model-server tensorflow-model-server-universal"
!echo 'deb {url} {src}' > /etc/apt/sources.list.d/tensorflow-serving.list
!curl '{url}/tensorflow-serving.release.pub.gpg' | apt-key add -
!apt update -q && apt-get install -y tensorflow-model-server
%pip install -q -U tensorflow-serving-api
```

This code starts by adding TensorFlow's package repository to Ubuntu's list of package sources. Then it downloads TensorFlow's public GPG key and adds it to the package manager's key list so it can verify TensorFlow's package signatures. Next, it uses apt to install the tensorflow-model-server package. Lastly, it installs the tensorflow-serving-api library, which we will need to communicate with the server.

Now we want to start the server. The command will require the absolute path of the base model directory (i.e., the path to my_mnist_model, not 0001), so let's save that to the MODEL_DIR environment variable:

```
import os
os.environ["MODEL_DIR"] = str(model_path.parent.absolute())
```

We can then start the server:

```
%%bash --bg
tensorflow_model_server \
--port=8500 \
--rest_api_port=8501 \
--model_name=my_mnist_model \
--model_base_path="${MODEL_DIR}" >my_server.log 2>&1
```

In Jupyter or Colab, the %%bash --bg magic command executes the cell as a

bash script, running it in the background. The `>my_server.log 2>&1` part redirects the standard output and standard error to the `my_server.log` file. And that's it! TF Serving is now running in the background, and its logs are saved to `my_server.log`. It loaded our MNIST model (version 1), and it is now waiting for gRPC and REST requests, respectively, on ports 8500 and 8501.

RUNNING TF SERVING IN A DOCKER CONTAINER

If you are running the notebook on your own machine and you have installed [Docker](#), you can run `docker pull tensorflow/serving` in a terminal to download the TF Serving image. The TensorFlow team highly recommends this installation method because it is simple, it will not mess with your system, and it offers high performance.⁵ To start the server inside a Docker container, you can run the following command in a terminal:

```
$ docker run -it --rm -v "/path/to/my_mnist_model:/models/my_mnist_model" \
-p 8500:8500 -p 8501:8501 -e MODEL_NAME=my_mnist_model tensorflow/serving
```

Here is what all these command-line options mean:

-it

Makes the container interactive (so you can press Ctrl-C to stop it) and displays the server's output.

--rm

Deletes the container when you stop it: no need to clutter your machine with interrupted containers. However, it does not delete the image.

-v "/path/to/my_mnist_model:/models/my_mnist_model"

Makes the host's `my_mnist_model` directory available to the container at the path `/models/mnist_model`. You must replace `/path/to/my_mnist_model` with the absolute path of this directory. On Windows, remember to use \ instead of / in the host path, but not in

the container path (since the container runs on Linux).

`-p 8500:8500`

Makes the Docker engine forward the host's TCP port 8500 to the container's TCP port 8500. By default, TF Serving uses this port to serve the gRPC API.

`-p 8501:8501`

Forwards the host's TCP port 8501 to the container's TCP port 8501. The Docker image is configured to use this port by default to serve the REST API.

`-e MODEL_NAME=my_mnist_model`

Sets the container's MODEL_NAME environment variable, so TF Serving knows which model to serve. By default, it will look for models in the `/models` directory, and it will automatically serve the latest version it finds.

`tensorflow/serving`

This is the name of the image to run.

Now that the server is up and running, let's query it, first using the REST API, then the gRPC API.

Querying TF Serving through the REST API

Let's start by creating the query. It must contain the name of the function signature you want to call, and of course the input data. Since the request must use the JSON format, we have to convert the input images from a NumPy array to a Python list:

```
import json  
  
X_new = X_test[:3] # pretend we have 3 new digit images to classify
```

```
request_json = json.dumps({
    "signature_name": "serving_default",
    "instances": X_new.tolist(),
})
```

Note that the JSON format is 100% text-based. The request string looks like this:

```
>>> request_json
'{"signature_name": "serving_default", "instances": [[[0, 0, 0, 0, ...]]]}'
```

Now let's send this request to TF Serving via an HTTP POST request. This can be done using the `requests` library (it is not part of Python's standard library, but it is preinstalled on Colab):

```
import requests

server_url = "http://localhost:8501/v1/models/my_mnist_model:predict"
response = requests.post(server_url, data=request_json)
response.raise_for_status() # raise an exception in case of error
response = response.json()
```

If all goes well, the response should be a dictionary containing a single "predictions" key. The corresponding value is the list of predictions. This list is a Python list, so let's convert it to a NumPy array and round the floats it contains to the second decimal:

```
>>> import numpy as np
>>> y_proba = np.array(response["predictions"])
>>> y_proba.round(2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.97, 0.01, 0. , 0. , 0. , 0.01, 0. , 0. ]])
```

Hurray, we have the predictions! The model is close to 100% confident that the first image is a 7, 99% confident that the second image is a 2, and 97% confident that the third image is a 1. That's correct.

The REST API is nice and simple, and it works well when the input and output data are not too large. Moreover, just about any client application can

make REST queries without additional dependencies, whereas other protocols are not always so readily available. However, it is based on JSON, which is text-based and fairly verbose. For example, we had to convert the NumPy array to a Python list, and every float ended up represented as a string. This is very inefficient, both in terms of serialization/deserialization time—we have to convert all the floats to strings and back—and in terms of payload size: many floats end up being represented using over 15 characters, which translates to over 120 bits for 32-bit floats! This will result in high latency and bandwidth usage when transferring large NumPy arrays.⁶ So, let's see how to use gRPC instead.

TIP

When transferring large amounts of data, or when latency is important, it is much better to use the gRPC API, if the client supports it, as it uses a compact binary format and an efficient communication protocol based on HTTP/2 framing.

Querying TF Serving through the gRPC API

The gRPC API expects a serialized PredictRequest protocol buffer as input, and it outputs a serialized PredictResponse protocol buffer. These protobufs are part of the tensorflow-serving-api library, which we installed earlier. First, let's create the request:

```
from tensorflow_serving.apis.predict_pb2 import PredictRequest

request = PredictRequest()
request.model_spec.name = model_name
request.model_spec.signature_name = "serving_default"
input_name = model.input_names[0] # == "flatten_input"
request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new))
```

This code creates a PredictRequest protocol buffer and fills in the required fields, including the model name (defined earlier), the signature name of the function we want to call, and finally the input data, in the form of a Tensor protocol buffer. The `tf.make_tensor_proto()` function creates a Tensor

protocol buffer based on the given tensor or NumPy array, in this case `X_new`.

Next, we'll send the request to the server and get its response. For this, we will need the `grpcio` library, which is preinstalled in Colab:

```
import grpc
from tensorflow_serving.apis import prediction_service_pb2_grpc

channel = grpc.insecure_channel('localhost:8500')
predict_service = prediction_service_pb2_grpc.PredictionServiceStub(channel)
response = predict_service.Predict(request, timeout=10.0)
```

The code is quite straightforward: after the imports, we create a gRPC communication channel to `localhost` on TCP port 8500, then we create a gRPC service over this channel and use it to send a request, with a 10-second timeout. Note that the call is synchronous: it will block until it receives the response or when the timeout period expires. In this example the channel is insecure (no encryption, no authentication), but gRPC and TF Serving also support secure channels over SSL/TLS.

Next, let's convert the `PredictResponse` protocol buffer to a tensor:

```
output_name = model.output_names[0] # == "dense_1"
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto)
```

If you run this code and print `y_proba.round(2)`, you will get the exact same estimated class probabilities as earlier. And that's all there is to it: in just a few lines of code, you can now access your TensorFlow model remotely, using either REST or gRPC.

Deploying a new model version

Now let's create a new model version and export a `SavedModel`, this time to the `my_mnist_model/0002` directory:

```
model = [...] # build and train a new MNIST model version
```

```
model_version = "0002"  
model_path = Path(model_name) / model_version  
model.save(model_path, save_format="tf")
```

At regular intervals (the delay is configurable), TF Serving checks the model directory for new model versions. If it finds one, it automatically handles the transition gracefully: by default, it answers pending requests (if any) with the previous model version, while handling new requests with the new version. As soon as every pending request has been answered, the previous model version is unloaded. You can see this at work in the TF Serving logs (in *my_server.log*):

```
[...]  
Reading SavedModel from: /models/my_mnist_model/0002  
Reading meta graph with tags { serve }  
[...]  
Successfully loaded servable version {name: my_mnist_model version: 2}  
Quiescing servable version {name: my_mnist_model version: 1}  
Done quiescing servable version {name: my_mnist_model version: 1}  
Unloading servable version {name: my_mnist_model version: 1}
```

TIP

If the SavedModel contains some example instances in the *assets.extra* directory, you can configure TF Serving to run the new model on these instances before starting to use it to serve requests. This is called *model warmup*: it will ensure that everything is properly loaded, avoiding long response times for the first requests.

This approach offers a smooth transition, but it may use too much RAM—especially GPU RAM, which is generally the most limited. In this case, you can configure TF Serving so that it handles all pending requests with the previous model version and unloads it before loading and using the new model version. This configuration will avoid having two model versions loaded at the same time, but the service will be unavailable for a short period.

As you can see, TF Serving makes it straightforward to deploy new models. Moreover, if you discover that version 2 does not work as well as you expected, then rolling back to version 1 is as simple as removing the

my_mnist_model/0002 directory.

TIP

Another great feature of TF Serving is its automatic batching capability, which you can activate using the `--enable_batching` option upon startup. When TF Serving receives multiple requests within a short period of time (the delay is configurable), it will automatically batch them together before using the model. This offers a significant performance boost by leveraging the power of the GPU. Once the model returns the predictions, TF Serving dispatches each prediction to the right client. You can trade a bit of latency for a greater throughput by increasing the batching delay (see the `--batching_parameters_file` option).

If you expect to get many queries per second, you will want to deploy TF Serving on multiple servers and load-balance the queries (see [Figure 19-2](#)). This will require deploying and managing many TF Serving containers across these servers. One way to handle that is to use a tool such as [Kubernetes](#), which is an open source system for simplifying container orchestration across many servers. If you do not want to purchase, maintain, and upgrade all the hardware infrastructure, you will want to use virtual machines on a cloud platform such as Amazon AWS, Microsoft Azure, Google Cloud Platform, IBM Cloud, Alibaba Cloud, Oracle Cloud, or some other platform as a service (PaaS) offering. Managing all the virtual machines, handling container orchestration (even with the help of Kubernetes), taking care of TF Serving configuration, tuning and monitoring—all of this can be a full-time job. Fortunately, some service providers can take care of all this for you. In this chapter we will use Vertex AI: it's the only platform with TPUs today; it supports TensorFlow 2, Scikit-Learn, and XGBoost; and it offers a nice suite of AI services. There are several other providers in this space that are capable of serving TensorFlow models as well, though, such as Amazon AWS SageMaker and Microsoft AI Platform, so make sure to check them out too.

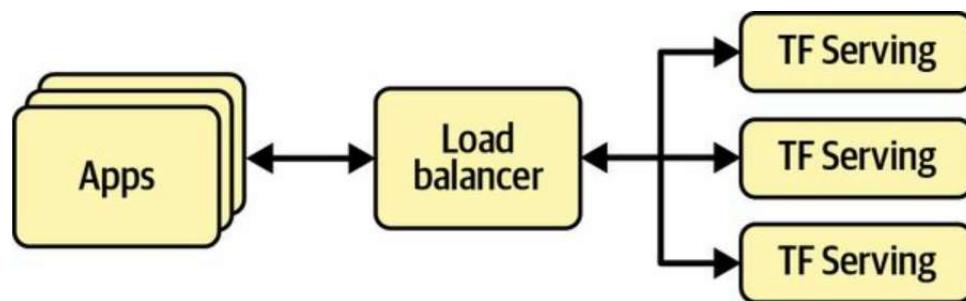


Figure 19-2. Scaling up TF Serving with load balancing

Now let's see how to serve our wonderful MNIST model on the cloud!

Creating a Prediction Service on Vertex AI

Vertex AI is a platform within Google Cloud Platform (GCP) that offers a wide range of AI-related tools and services. You can upload datasets, get humans to label them, store commonly used features in a feature store and use them for training or in production, and train models across many GPU or TPU servers with automatic hyperparameter tuning or model architecture search (AutoML). You can also manage your trained models, use them to make batch predictions on large amounts of data, schedule multiple jobs for your data workflows, serve your models via REST or gRPC at scale, and experiment with your data and models within a hosted Jupyter environment called the *Workbench*. There's even a *Matching Engine* service that lets you compare vectors very efficiently (i.e., approximate nearest neighbors). GCP also includes other AI services, such as APIs for computer vision, translation, speech-to-text, and more.

Before we start, there's a little bit of setup to take care of:

1. Log in to your Google account, and then go to the [Google Cloud Platform console](#) (see [Figure 19-3](#)). If you don't have a Google account, you'll have to create one.
2. If it's your first time using GCP, you'll have to read and accept the terms and conditions. New users are offered a free trial, including \$300 worth of GCP credit that you can use over the course of 90 days (as of May 2022). You'll only need a small portion of that to pay for the services you'll use in this chapter. Upon signing up for a free trial, you'll still need to create a payment profile and enter your credit card number: it's used for verification purposes—probably to avoid people using the free trial multiple times—but you won't be billed for the first \$300, and after that you'll only be charged if you opt in by upgrading to a paid account.

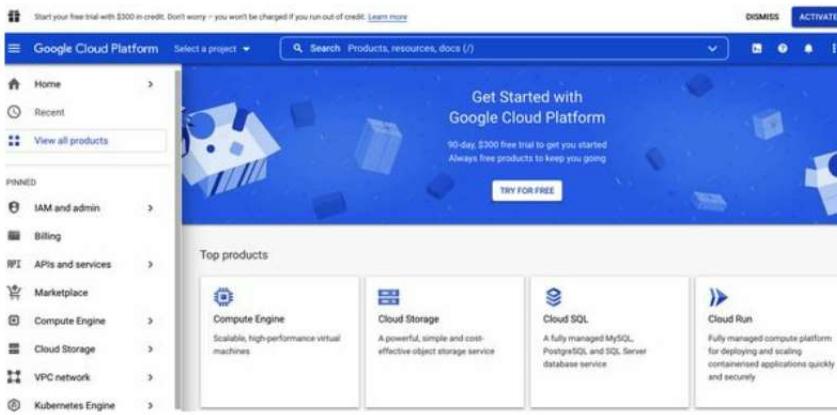


Figure 19-3. Google Cloud Platform console

3. If you have used GCP before and your free trial has expired, then the services you will use in this chapter will cost you some money. It shouldn't be too much, especially if you remember to turn off the services when you don't need them anymore. Make sure you understand and agree to the pricing conditions before you run any service. I hereby decline any responsibility if services end up costing more than you expected! Also make sure your billing account is active. To check, open the **≡** navigation menu at the top left and click Billing, then make sure you have set up a payment method and that the billing account is active.
4. Every resource in GCP belongs to a *project*. This includes all the virtual machines you may use, the files you store, and the training jobs you run. When you create an account, GCP automatically creates a project for you, called "My First Project". If you want, you can change its display name by going to the project settings: in the **≡** navigation menu, select "IAM and admin → Settings", change the project's display name, and click SAVE. Note that the project also has a unique ID and number. You can choose the project ID when you create a project, but you cannot change it later. The project number is automatically generated and cannot be changed. If you want to create a new project, click the project name at the top of the page, then click NEW PROJECT and enter the project name. You can also click EDIT to set the project ID. Make sure billing is active for this new project so that service fees can be billed (to

your free credits, if any).

WARNING

Always set an alarm to remind yourself to turn services off when you know you will only need them for a few hours, or else you might leave them running for days or months, incurring potentially significant costs.

5. Now that you have a GCP account and a project, and billing is activated, you must activate the APIs you need. In the  navigation menu, select “APIs and services”, and make sure the Cloud Storage API is enabled. If needed, click + ENABLE APIS AND SERVICES, find Cloud Storage, and enable it. Also enable the Vertex AI API.

You could continue to do everything via the GCP console, but I recommend using Python instead: this way you can write scripts to automate just about anything you want with GCP, and it’s often more convenient than clicking your way through menus and forms, especially for common tasks.

GOOGLE CLOUD CLI AND SHELL

Google Cloud’s command-line interface (CLI) includes the `gcloud` command, which lets you control almost everything in GCP, and `gsutil`, which lets you interact with Google Cloud Storage. This CLI is preinstalled in Colab: all you need to do is authenticate using `google.auth.authenticate_user()`, and you’re good to go. For example, `!gcloud config list` will display the configuration.

GCP also offers a preconfigured shell environment called the Google Cloud Shell, which you can use directly in your web browser; it runs on a free Linux VM (Debian) with the Google Cloud SDK already preinstalled and configured for you, so there’s no need to authenticate. The Cloud Shell is available anywhere in GCP: just click the Activate Cloud Shell icon at the top right of the page (see [Figure 19-4](#)).

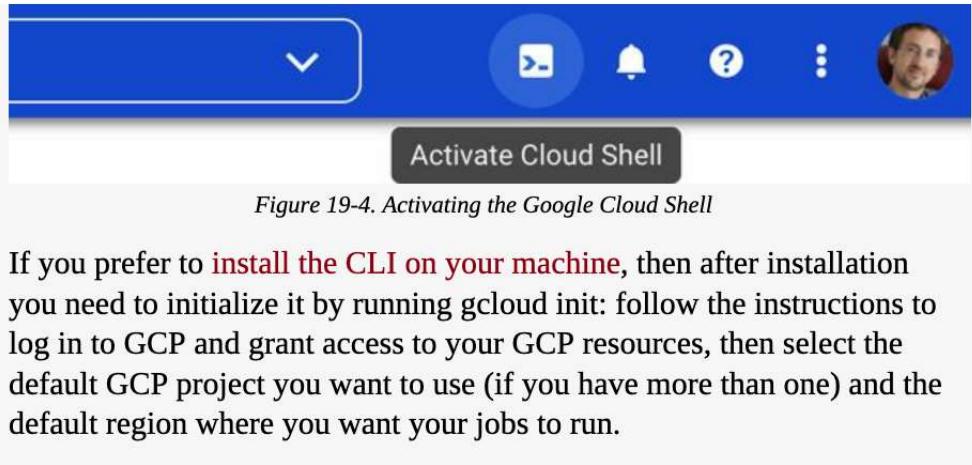


Figure 19-4. Activating the Google Cloud Shell

If you prefer to [install the CLI on your machine](#), then after installation you need to initialize it by running gcloud init: follow the instructions to log in to GCP and grant access to your GCP resources, then select the default GCP project you want to use (if you have more than one) and the default region where you want your jobs to run.

The first thing you need to do before you can use any GCP service is to authenticate. The simplest solution when using Colab is to execute the following code:

```
from google.colab import auth  
auth.authenticate_user()
```

The authentication process is based on [OAuth 2.0](#): a pop-up window will ask you to confirm that you want the Colab notebook to access your Google credentials. If you accept, you must select the same Google account you used for GCP. Then you will be asked to confirm that you agree to give Colab full access to all your data on Google Drive and in GCP. If you allow access, only the current notebook will have access, and only until the Colab runtime expires. Obviously, you should only accept this if you trust the code in the notebook.

WARNING

If you are *not* working with the official notebooks from <https://github.com/ageron/handson-ml3>, then you should be extra careful: if the notebook's author is mischievous, they could include code to do whatever they want with your data.

AUTHENTICATION AND AUTHORIZATION ON GCP

In general, using OAuth 2.0 authentication is only recommended when an application must access the user's personal data or resources from another application, on the user's behalf. For example, some applications allow the user to save data to their Google Drive, but for that the application first needs the user to authenticate with Google and allow access to Google Drive. In general, the application will only ask for the level of access it needs; it won't be an unlimited access: for example, the application will only request access to Google Drive, not Gmail or any other Google service. Moreover, the authorization usually expires after a while, and it can always be revoked.

When an application needs to access a service on GCP on its own behalf, not on behalf of the user, then it should generally use a *service account*. For example, if you build a website that needs to send prediction requests to a Vertex AI endpoint, then the website will be accessing the service on its own behalf. There's no data or resource that it needs to access in the user's Google account. In fact, many users of the website will not even have a Google account. For this scenario, you first need to create a service account. Select "IAM and admin → Service accounts" in the GCP console's  navigation menu (or use the search box), then click + CREATE SERVICE ACCOUNT, fill in the first page of the form (service account name, ID, description), and click CREATE AND CONTINUE. Next, you must give this account some access rights. Select the "Vertex AI user" role: this will allow the service account to make predictions and use other Vertex AI services, but nothing else. Click CONTINUE. You can now optionally grant some users access to the service account: this is useful when your GCP user account is part of an organization and you wish to authorize other users in the organization to deploy applications that will be based on this service account, or to manage the service account itself. Next, click DONE.

Once you have created a service account, your application must authenticate as that service account. There are several ways to do that. If

your application is hosted on GCP—for example, if you are coding a website hosted on Google Compute Engine—then the simplest and safest solution is to attach the service account to the GCP resource that hosts your website, such as a VM instance or a Google App Engine service. This can be done when creating the GCP resource, by selecting the service account in the “Identity and API access” section. Some resources, such as VM instances, also let you attach the service account after the VM instance is created: you must stop it and edit its settings. In any case, once a service account is attached to a VM instance, or any other GCP resource running your code, GCP’s client libraries (discussed shortly) will automatically authenticate as the chosen service account, with no extra step needed.

If your application is hosted using Kubernetes, then you should use Google’s Workload Identity service to map the right service account to each Kubernetes service account. If your application is not hosted on GCP—for example, if you are just running the Jupyter notebook on your own machine—then you can either use the Workload Identity Federation service (that’s the safest but hardest option), or just generate an access key for your service account, save it to a JSON file, and point the `GOOGLE_APPLICATION_CREDENTIALS` environment variable to it so your client application can access it. You can manage access keys by clicking the service account you just created, and then opening the KEYS tab. Make sure to keep the key file secret: it’s like a password for the service account.

For more details on setting up authentication and authorization so your application can access GCP services, check out the [documentation](#).

Now let’s create a Google Cloud Storage bucket to store our SavedModels (a GCS *bucket* is a container for your data). For this we will use the `google-cloud-storage` library, which is preinstalled in Colab. We first create a Client object, which will serve as the interface with GCS, then we use it to create the bucket:

```
from google.cloud import storage
```

```
project_id = "my_project" # change this to your project ID
bucket_name = "my_bucket" # change this to a unique bucket name
location = "us-central1"

storage_client = storage.Client(project=project_id)
bucket = storage_client.create_bucket(bucket_name, location=location)
```

TIP

If you want to reuse an existing bucket, replace the last line with `bucket = storage_client.bucket(bucket_name)`. Make sure location is set to the bucket's region.

GCS uses a single worldwide namespace for buckets, so simple names like “machine-learning” will most likely not be available. Make sure the bucket name conforms to DNS naming conventions, as it may be used in DNS records. Moreover, bucket names are public, so do not put anything private in the name. It is common to use your domain name, your company name, or your project ID as a prefix to ensure uniqueness, or simply use a random number as part of the name.

You can change the region if you want, but be sure to choose one that supports GPUs. Also, you may want to consider the fact that prices vary greatly between regions, some regions produce much more CO₂ than others, some regions do not support all services, and using a single-region bucket improves performance. See [Google Cloud’s list of regions](#) and [Vertex AI’s documentation on locations](#) for more details. If you are unsure, it might be best to stick with “us-central1”.

Next, let’s upload the `my_mnist_model` directory to the new bucket. Files in GCS are called *blobs* (or *objects*), and under the hood they are all just placed in the bucket without any directory structure. Blob names can be arbitrary Unicode strings, and they can even contain forward slashes (/). The GCP console and other tools use these slashes to give the illusion that there are directories. So, when we upload the `my_mnist_model` directory, we only care about the files, not the directories:

```
def upload_directory(bucket, dirpath):
    dirpath = Path(dirpath)
    for filepath in dirpath.glob("*/*"):
        if filepath.is_file():
            blob = bucket.blob(filepath.relative_to(dirpath.parent).as_posix())
            blob.upload_from_filename(filepath)

upload_directory(bucket, "my_mnist_model")
```

This function works fine now, but it would be very slow if there were many files to upload. It's not too hard to speed it up tremendously by multithreading it (see the notebook for an implementation). Alternatively, if you have the Google Cloud CLI, then you can use following command instead:

```
!gsutil -m cp -r my_mnist_model gs://{bucket_name}/
```

Next, let's tell Vertex AI about our MNIST model. To communicate with Vertex AI, we can use the `google-cloud-aiplatform` library (it still uses the old AI Platform name instead of Vertex AI). It's not preinstalled in Colab, so we need to install it. After that, we can import the library and initialize it—just to specify some default values for the project ID and the location—then we can create a new Vertex AI model: we specify a display name, the GCS path to our model (in this case the version 0001), and the URL of the Docker container we want Vertex AI to use to run this model. If you visit that URL and navigate up one level, you will find other containers you can use. This one supports TensorFlow 2.8 with a GPU:

```
from google.cloud import aiplatform

server_image = "gcr.io/cloud-aiplatform/prediction/tf2-gpu.2-8:latest"

aiplatform.init(project=project_id, location=location)
mnist_model = aiplatform.Model.upload(
    display_name="mnist",
    artifact_uri=f"gs://{bucket_name}/my_mnist_model/0001",
    serving_container_image_uri=server_image,
)
```

Now let's deploy this model so we can query it via a gRPC or REST API to make predictions. For this we first need to create an *endpoint*. This is what client applications connect to when they want to access a service. Then we need to deploy our model to this endpoint:

```
endpoint = aiplatform.Endpoint.create(display_name="mnist-endpoint")

endpoint.deploy(
    mnist_model,
    min_replica_count=1,
    max_replica_count=5,
    machine_type="n1-standard-4",
    accelerator_type="NVIDIA_TESLA_K80",
    accelerator_count=1
)
```

This code may take a few minutes to run, because Vertex AI needs to set up a virtual machine. In this example, we use a fairly basic machine of type n1-standard-4 (see <https://homl.info/machinetypes> for other types). We also use a basic GPU of type NVIDIA_TESLA_K80 (see <https://homl.info/accelerators> for other types). If you selected another region than "us-central1", then you may need to change the machine type or the accelerator type to values that are supported in that region (e.g., not all regions have Nvidia Tesla K80 GPUs).

NOTE

Google Cloud Platform enforces various GPU quotas, both worldwide and per region: you cannot create thousands of GPU nodes without prior authorization from Google. To check your quotas, open "IAM and admin → Quotas" in the GCP console. If some quotas are too low (e.g., if you need more GPUs in a particular region), you can ask for them to be increased; it often takes about 48 hours.

Vertex AI will initially spawn the minimum number of compute nodes (just one in this case), and whenever the number of queries per second becomes too high, it will spawn more nodes (up to the maximum number you defined, five in this case) and will load-balance the queries between them. If the QPS

rate goes down for a while, Vertex AI will stop the extra compute nodes automatically. The cost is therefore directly linked to the load, as well as the machine and accelerator types you selected and the amount of data you store on GCS. This pricing model is great for occasional users and for services with important usage spikes. It's also ideal for startups: the price remains low until the startup actually starts up.

Congratulations, you have deployed your first model to the cloud! Now let's query this prediction service:

```
response = endpoint.predict(instances=X_new.tolist())
```

We first need to convert the images we want to classify to a Python list, as we did earlier when we sent requests to TF Serving using the REST API. The response object contains the predictions, represented as a Python list of lists of floats. Let's round them to two decimal places and convert them to a NumPy array:

```
>>> import numpy as np
>>> np.round(response.predictions, 2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.97, 0.01, 0. , 0. , 0. , 0.01, 0. , 0. ]])
```

Yes! We get the exact same predictions as earlier. We now have a nice prediction service running on the cloud that we can query from anywhere securely, and which can automatically scale up or down depending on the number of QPS. When you are done using the endpoint, don't forget to delete it, to avoid paying for nothing:

```
endpoint.undeploy_all() # undeploy all models from the endpoint
endpoint.delete()
```

Now let's see how to run a job on Vertex AI to make predictions on a potentially very large batch of data.

Running Batch Prediction Jobs on Vertex AI

If we have a large number of predictions to make, then instead of calling our prediction service repeatedly, we can ask Vertex AI to run a prediction job for us. This does not require an endpoint, only a model. For example, let's run a prediction job on the first 100 images of the test set, using our MNIST model. For this, we first need to prepare the batch and upload it to GCS. One way to do this is to create a file containing one instance per line, each formatted as a JSON value—this format is called *JSON Lines*—then pass this file to Vertex AI. So let's create a JSON Lines file in a new directory, then upload this directory to GCS:

```
batch_path = Path("my_mnist_batch")
batch_path.mkdir(exist_ok=True)
with open(batch_path / "my_mnist_batch.jsonl", "w") as jsonl_file:
    for image in X_test[:100].tolist():
        jsonl_file.write(json.dumps(image))
        jsonl_file.write("\n")

upload_directory(bucket, batch_path)
```

Now we're ready to launch the prediction job, specifying the job's name, the type and number of machines and accelerators to use, the GCS path to the JSON Lines file we just created, and the path to the GCS directory where Vertex AI will save the model's predictions:

```
batch_prediction_job = mnist_model.batch_predict(
    job_display_name="my_batch_prediction_job",
    machine_type="n1-standard-4",
    starting_replica_count=1,
    max_replica_count=5,
    accelerator_type="NVIDIA_TESLA_K80",
    accelerator_count=1,
    gcs_source=[f"gs://{bucket_name}/{batch_path.name}/my_mnist_batch.jsonl"],
    gcs_destination_prefix=f"gs://{bucket_name}/my_mnist_predictions/",
    sync=True # set to False if you don't want to wait for completion
)
```

TIP

For large batches, you can split the inputs into multiple JSON Lines files and list them all via the gcs_source argument.

This will take a few minutes, mostly to spawn the compute nodes on Vertex AI. Once this command completes, the predictions will be available in a set of files named something like *prediction.results-00001-of-00002*. These files use the JSON Lines format by default, and each value is a dictionary containing an instance and its corresponding prediction (i.e., 10 probabilities). The instances are listed in the same order as the inputs. The job also outputs *prediction-errors** files, which can be useful for debugging if something goes wrong. We can iterate through all these output files using batch_prediction_job.iter_outputs(), so let's go through all the predictions and store them in a y_probas array:

```
y_probas = []
for blob in batch_prediction_job.iter_outputs():
    if "prediction.results" in blob.name:
        for line in blob.download_as_text().splitlines():
            y_proba = json.loads(line)["prediction"]
            y_probas.append(y_proba)
```

Now let's see how good these predictions are:

```
>>> y_pred = np.argmax(y_probas, axis=1)
>>> accuracy = np.sum(y_pred == y_test[:100]) / 100
0.98
```

Nice, 98% accuracy!

The JSON Lines format is the default, but when dealing with large instances such as images, it is too verbose. Luckily, the batch_predict() method accepts an instances_format argument that lets you choose another format if you want. It defaults to "jsonl", but you can change it to "csv", "tf-record", "tf-record-gzip", "bigquery", or "file-list". If you set it to "file-list", then the gcs_source argument should point to a text file containing one input filepath

per line; for instance, pointing to PNG image files. Vertex AI will read these files as binary, encode them using Base64, and pass the resulting byte strings to the model. This means that you must add a preprocessing layer in your model to parse the Base64 strings, using `tf.io.decode_base64()`. If the files are images, you must then parse the result using a function like `tf.io.decode_image()` or `tf.io.decode_png()`, as discussed in [Chapter 13](#).

When you’re finished using the model, you can delete it if you want, by running `mnist_model.delete()`. You can also delete the directories you created in your GCS bucket, optionally the bucket itself (if it’s empty), and the batch prediction job:

```
for prefix in ["my_mnist_model/", "my_mnist_batch/", "my_mnist_predictions/"]:
    blobs = bucket.list_blobs(prefix=prefix)
    for blob in blobs:
        blob.delete()

bucket.delete() # if the bucket is empty
batch_prediction_job.delete()
```

You now know how to deploy a model to Vertex AI, create a prediction service, and run batch prediction jobs. But what if you want to deploy your model to a mobile app instead? Or to an embedded device, such as a heating control system, a fitness tracker, or a self-driving car?

Deploying a Model to a Mobile or Embedded Device

Machine learning models are not limited to running on big centralized servers with multiple GPUs: they can run closer to the source of data (this is called *edge computing*), for example in the user's mobile device or in an embedded device. There are many benefits to decentralizing the computations and moving them toward the edge: it allows the device to be smart even when it's not connected to the internet, it reduces latency by not having to send data to a remote server and reduces the load on the servers, and it may improve privacy, since the user's data can stay on the device.

However, deploying models to the edge has its downsides too. The device's computing resources are generally tiny compared to a beefy multi-GPU server. A large model may not fit in the device, it may use too much RAM and CPU, and it may take too long to download. As a result, the application may become unresponsive, and the device may heat up and quickly run out of battery. To avoid all this, you need to make a lightweight and efficient model, without sacrificing too much of its accuracy. The **TFLite** library provides several tools ⁷ to help you deploy your models to the edge, with three main objectives:

- Reduce the model size, to shorten download time and reduce RAM usage.
- Reduce the amount of computations needed for each prediction, to reduce latency, battery usage, and heating.
- Adapt the model to device-specific constraints.

To reduce the model size, TFLite's model converter can take a SavedModel and compress it to a much lighter format based on **FlatBuffers**. This is an efficient cross-platform serialization library (a bit like protocol buffers) initially created by Google for gaming. It is designed so you can load FlatBuffers straight to RAM without any preprocessing: this reduces the loading time and memory footprint. Once the model is loaded into a mobile

or embedded device, the TFLite interpreter will execute it to make predictions. Here is how you can convert a SavedModel to a FlatBuffer and save it to a `.tflite` file:

```
converter = tf.lite.TFLiteConverter.from_saved_model(str(model_path))
tflite_model = converter.convert()
with open("my_converted_savedmodel.tflite", "wb") as f:
    f.write(tflite_model)
```

TIP

You can also save a Keras model directly to a FlatBuffer using `tf.lite.TFLiteConverter.from_keras_model(model)`.

The converter also optimizes the model, both to shrink it and to reduce its latency. It prunes all the operations that are not needed to make predictions (such as training operations), and it optimizes computations whenever possible; for example, $3 \times a + 4 \times a + 5 \times a$ will be converted to $12 \times a$. Additionally, it tries to fuse operations whenever possible. For example, if possible, batch normalization layers end up folded into the previous layer's addition and multiplication operations. To get a good idea of how much TFLite can optimize a model, download one of the [pretrained TFLite models](#), such as *Inception_V1_quant* (click `tflite&pb`), unzip the archive, then open the excellent [Netron graph visualization tool](#) and upload the `.pb` file to view the original model. It's a big, complex graph, right? Next, open the optimized `.tflite` model and marvel at its beauty!

Another way you can reduce the model size—other than simply using smaller neural network architectures—is by using smaller bit-widths: for example, if you use half-floats (16 bits) rather than regular floats (32 bits), the model size will shrink by a factor of 2, at the cost of a (generally small) accuracy drop. Moreover, training will be faster, and you will use roughly half the amount of GPU RAM.

TFLite's converter can go further than that, by quantizing the model weights down to fixed-point, 8-bit integers! This leads to a fourfold size reduction

compared to using 32-bit floats. The simplest approach is called *post-training quantization*: it just quantizes the weights after training, using a fairly basic but efficient symmetrical quantization technique. It finds the maximum absolute weight value, m , then it maps the floating-point range $-m$ to $+m$ to the fixed-point (integer) range -127 to $+127$. For example, if the weights range from -1.5 to $+0.8$, then the bytes -127 , 0 , and $+127$ will correspond to the floats -1.5 , 0.0 , and $+1.5$, respectively (see Figure 19-5). Note that 0.0 always maps to 0 when using symmetrical quantization. Also note that the byte values $+68$ to $+127$ will not be used in this example, since they map to floats greater than $+0.8$.

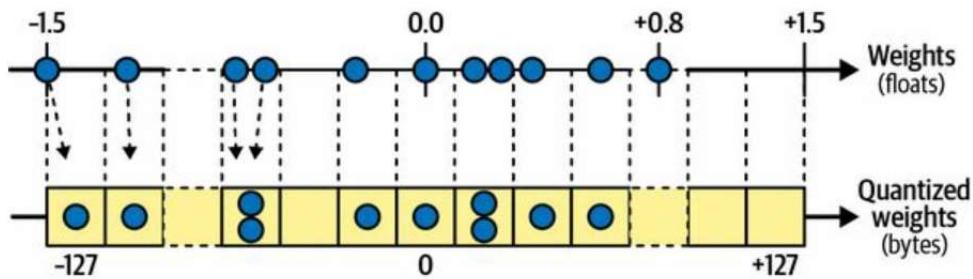


Figure 19-5. From 32-bit floats to 8-bit integers, using symmetrical quantization

To perform this post-training quantization, simply add `DEFAULT` to the list of converter optimizations before calling the `convert()` method:

```
converter.optimizations = [tf.lite.Optimize.DEFAULT]
```

This technique dramatically reduces the model's size, which makes it much faster to download, and uses less storage space. At runtime the quantized weights get converted back to floats before they are used. These recovered floats are not perfectly identical to the original floats, but they're not too far off, so the accuracy loss is usually acceptable. To avoid recomputing the float values all the time, which would severely slow down the model, TFLite caches them: unfortunately, this means that this technique does not reduce RAM usage, and it doesn't speed up the model either. It's mostly useful to reduce the application's size.

The most effective way to reduce latency and power consumption is to also

quantize the activations so that the computations can be done entirely with integers, without the need for any floating-point operations. Even when using the same bit-width (e.g., 32-bit integers instead of 32-bit floats), integer computations use less CPU cycles, consume less energy, and produce less heat. And if you also reduce the bit-width (e.g., down to 8-bit integers), you can get huge speedups. Moreover, some neural network accelerator devices—such as Google’s Edge TPU—can only process integers, so full quantization of both weights and activations is compulsory. This can be done post-training; it requires a calibration step to find the maximum absolute value of the activations, so you need to provide a representative sample of training data to TFLite (it does not need to be huge), and it will process the data through the model and measure the activation statistics required for quantization. This step is typically fast.

The main problem with quantization is that it loses a bit of accuracy: it is similar to adding noise to the weights and activations. If the accuracy drop is too severe, then you may need to use *quantization-aware training*. This means adding fake quantization operations to the model so it can learn to ignore the quantization noise during training; the final weights will then be more robust to quantization. Moreover, the calibration step can be taken care of automatically during training, which simplifies the whole process.

I have explained the core concepts of TFLite, but going all the way to coding a mobile or embedded application would require a dedicated book. Fortunately, some exist: if you want to learn more about building TensorFlow applications for mobile and embedded devices, check out the O’Reilly books *TinyML: Machine Learning with TensorFlow on Arduino and Ultra-Low Power Micro-Controllers*, by Pete Warden (former lead of the TFLite team) and Daniel Situnayake and *AI and Machine Learning for On-Device Development*, by Laurence Moroney.

Now what if you want to use your model in a website, running directly in the user’s browser?

Running a Model in a Web Page

Running your machine learning model on the client side, in the user's browser, rather than on the server side can be useful in many scenarios, such as:

- When your web application is often used in situations where the user's connectivity is intermittent or slow (e.g., a website for hikers), so running the model directly on the client side is the only way to make your website reliable.
- When you need the model's responses to be as fast as possible (e.g., for an online game). Removing the need to query the server to make predictions will definitely reduce the latency and make the website much more responsive.
- When your web service makes predictions based on some private user data, and you want to protect the user's privacy by making the predictions on the client side so that the private data never has to leave the user's machine.

For all these scenarios, you can use the [TensorFlow.js \(TFJS\) JavaScript library](#). This library can load a TFLite model and make predictions directly in the user's browser. For example, the following JavaScript module imports the TFJS library, downloads a pretrained MobileNet model, and uses this model to classify an image and log the predictions. You can play with the code at <https://homl.info/tfjscode>, using Glitch.com, a website that lets you build web apps in your browser for free; click the PREVIEW button in the lower-right corner of the page to see the code in action:

```
import "https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest";
import "https://cdn.jsdelivr.net/npm/@tensorflow-models/mobilenet@1.0.0";

const image = document.getElementById("image");

mobilenet.load().then(model => {
```

```
model.classify(image).then(predictions => {
  for (var i = 0; i < predictions.length; i++) {
    let className = predictions[i].className
    let proba = (predictions[i].probability * 100).toFixed(1)
    console.log(className + " : " + proba + "%");
  }
});
```

It's even possible to turn this website into a *progressive web app* (PWA): this is a website that respects a number of criteria ⁸ that allow it to be viewed in any browser, and even installed as a standalone app on a mobile device. For example, try visiting <https://homl.info/tfjswpa> on a mobile device: most modern browsers will ask you whether you would like to add TFJS Demo to your home screen. If you accept, you will see a new icon in your list of applications. Clicking this icon will load the TFJS Demo website inside its own window, just like a regular mobile app. A PWA can even be configured to work offline, by using a *service worker*: this is a JavaScript module that runs in its own separate thread in the browser and intercepts network requests, allowing it to cache resources so the PWA can run faster, or even entirely offline. It can also deliver push messages, run tasks in the background, and more. PWAs allow you to manage a single code base for the web and for mobile devices. They also make it easier to ensure that all users run the same version of your application. You can play with this TFJS Demo's PWA code on Glitch.com at <https://homl.info/wpacode>.

TIP

Check out many more demos of machine learning models running in your browser at <https://tensorflow.org/js/demos>.

TFJS also supports training a model directly in your web browser! And it's actually pretty fast. If your computer has a GPU card, then TFJS can generally use it, even if it's not an Nvidia card. Indeed, TFJS will use WebGL when it's available, and since modern web browsers generally support a wide range of GPU cards, TFJS actually supports more GPU cards

than regular TensorFlow (which only supports Nvidia cards).

Training a model in a user's web browser can be especially useful to guarantee that this user's data remains private. A model can be trained centrally, and then fine-tuned locally, in the browser, based on that user's data. If you're interested in this topic, check out *federated learning*.

Once again, doing justice to this topic would require a whole book. If you want to learn more about TensorFlow.js, check out the O'reilly books *Practical Deep Learning for Cloud, Mobile, and Edge*, by Anirudh Koul et al., or *Learning TensorFlow.js*, by Gant Laborde.

Now that you've seen how to deploy TensorFlow models to TF Serving, or to the cloud with Vertex AI, or to mobile and embedded devices using TFLite, or to a web browser using TFJS, let's discuss how to use GPUs to speed up computations.

Using GPUs to Speed Up Computations

In [Chapter 11](#) we looked at several techniques that can considerably speed up training: better weight initialization, sophisticated optimizers, and so on. But even with all of these techniques, training a large neural network on a single machine with a single CPU can take hours, days, or even weeks, depending on the task. Thanks to GPUs, this training time can be reduced down to minutes or hours. Not only does this save an enormous amount of time, but it also means that you can experiment with various models much more easily, and frequently retrain your models on fresh data.

In the previous chapters, we used GPU-enabled runtimes on Google Colab. All you have to do for this is select “Change runtime type” from the Runtime menu, and choose the GPU accelerator type; TensorFlow automatically detects the GPU and uses it to speed up computations, and the code is exactly the same as without a GPU. Then, in this chapter you saw how to deploy your models to Vertex AI on multiple GPU-enabled compute nodes: it’s just a matter of selecting the right GPU-enabled Docker image when creating the Vertex AI model, and selecting the desired GPU type when calling `endpoint.deploy()`. But what if you want to buy your own GPU? And what if you want to distribute the computations across the CPU and multiple GPU devices on a single machine (see [Figure 19-6](#))? This is what we will discuss now, then later in this chapter we will discuss how to distribute computations across multiple servers.

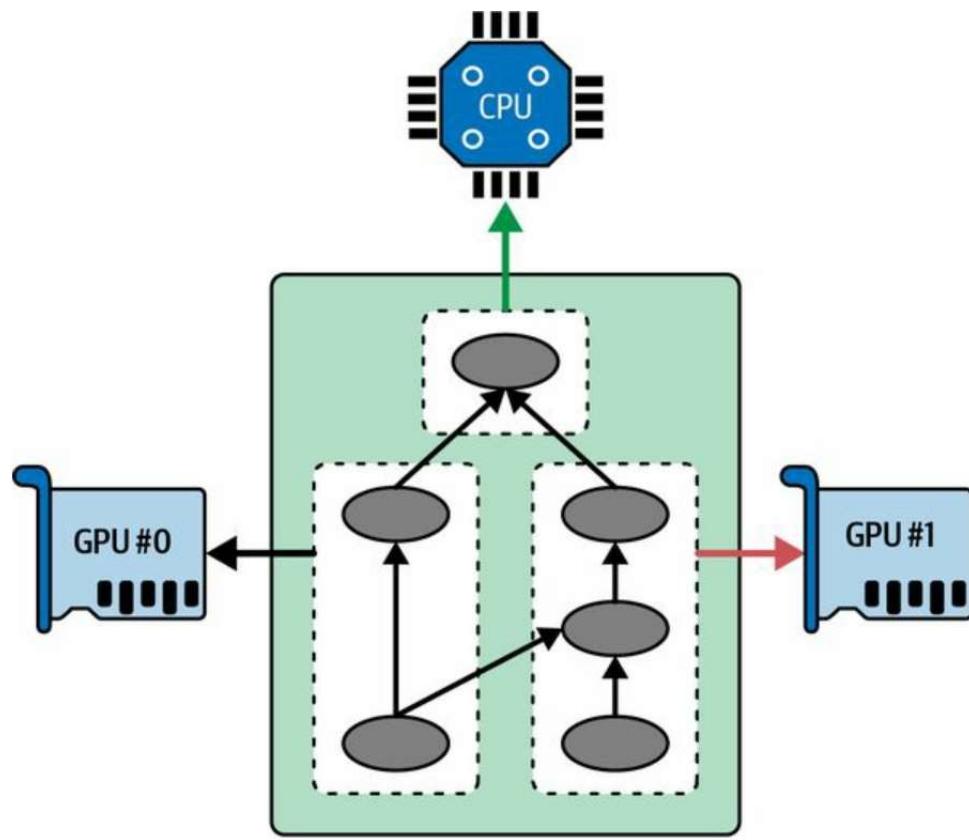


Figure 19-6. Executing a TensorFlow graph across multiple devices in parallel

Getting Your Own GPU

If you know that you'll be using a GPU heavily and for a long period of time, then buying your own can make financial sense. You may also want to train your models locally because you do not want to upload your data to the cloud. Or perhaps you just want to buy a GPU card for gaming, and you'd like to use it for deep learning as well.

If you decide to purchase a GPU card, then take some time to make the right choice. You will need to consider the amount of RAM you will need for your tasks (e.g., typically at least 10 GB for image processing or NLP), the bandwidth (i.e., how fast you can send data into and out of the GPU), the number of cores, the cooling system, etc. Tim Dettmers wrote an [excellent blog post](#) to help you choose: I encourage you to read it carefully. At the time of this writing, TensorFlow only supports [Nvidia cards with CUDA Compute Capability 3.5+](#) (as well as Google's TPUs, of course), but it may extend its support to other manufacturers, so make sure to check [TensorFlow's documentation](#) to see what devices are supported today.

If you go for an Nvidia GPU card, you will need to install the appropriate Nvidia drivers and several Nvidia libraries.⁹ These include the *Compute Unified Device Architecture* library (CUDA) Toolkit, which allows developers to use CUDA-enabled GPUs for all sorts of computations (not just graphics acceleration), and the *CUDA Deep Neural Network* library (cuDNN), a GPU-accelerated library of common DNN computations such as activation layers, normalization, forward and backward convolutions, and pooling (see [Chapter 14](#)). cuDNN is part of Nvidia's Deep Learning SDK. Note that you will need to create an Nvidia developer account in order to download it. TensorFlow uses CUDA and cuDNN to control the GPU cards and accelerate computations (see [Figure 19-7](#)).

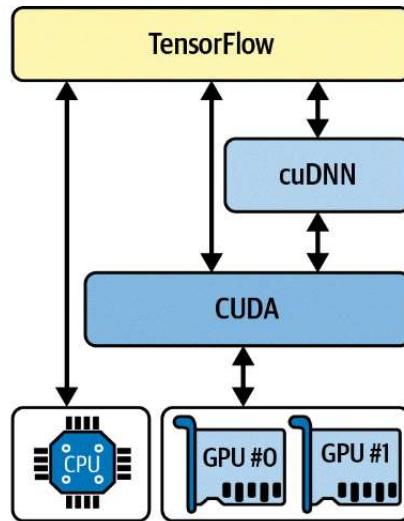


Figure 19-7. TensorFlow uses CUDA and cuDNN to control GPUs and boost DNNs

Once you have installed the GPU card(s) and all the required drivers and libraries, you can use the `nvidia-smi` command to check that everything is properly installed. This command lists the available GPU cards, as well as all the processes running on each card. In this example, it's an Nvidia Tesla T4 GPU card with about 15 GB of available RAM, and there are no processes currently running on it:

```

$ nvidia-smi
Sun Apr 10 04:52:10 2022
+-----+
| NVIDIA-SMI 460.32.03  Driver Version: 460.32.03  CUDA Version: 11.2 |
+-----+-----+-----+
| GPU Name      Persistence-M| Bus-Id      Disp.A | Volatile Uncorr. ECC | |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|          |          |          | MIG M.   |
+-----+-----+-----+-----+
| 0  Tesla T4      Off  | 00000000:00:04.0 Off |          0 | |
| N/A  34C  P8  9W / 70W | 3MiB / 15109MiB | 0%     Default |
|          |          |          | N/A      |
+-----+-----+-----+
+-----+
| Processes:                               |
| GPU  GI  CI      PID  Type  Process name        GPU Memory |
| GPU  GI  CI      PID  Type  Process name        GPU Memory |
  
```

ID	ID	Usage
<hr/>		
No running processes found		
<hr/>		

To check that TensorFlow actually sees your GPU, run the following commands and make sure the result is not empty:

```
>>> physical_gpus = tf.config.list_physical_devices("GPU")
>>> physical_gpus
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

Managing the GPU RAM

By default TensorFlow automatically grabs almost all the RAM in all available GPUs the first time you run a computation. It does this to limit GPU RAM fragmentation. This means that if you try to start a second TensorFlow program (or any program that requires the GPU), it will quickly run out of RAM. This does not happen as often as you might think, as you will most often have a single TensorFlow program running on a machine: usually a training script, a TF Serving node, or a Jupyter notebook. If you need to run multiple programs for some reason (e.g., to train two different models in parallel on the same machine), then you will need to split the GPU RAM between these processes more evenly.

If you have multiple GPU cards on your machine, a simple solution is to assign each of them to a single process. To do this, you can set the CUDA_VISIBLE_DEVICES environment variable so that each process only sees the appropriate GPU card(s). Also set the CUDA_DEVICE_ORDER environment variable to PCI_BUS_ID to ensure that each ID always refers to the same GPU card. For example, if you have four GPU cards, you could start two programs, assigning two GPUs to each of them, by executing commands like the following in two separate terminal windows:

```
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=0,1 python3  
program_1.py  
# and in another terminal:  
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=3,2 python3  
program_2.py
```

Program 1 will then only see GPU cards 0 and 1, named "/gpu:0" and "/gpu:1", respectively, in TensorFlow, and program 2 will only see GPU cards 2 and 3, named "/gpu:1" and "/gpu:0", respectively (note the order). Everything will work fine (see [Figure 19-8](#)). Of course, you can also define these environment variables in Python by setting os.environ["CUDA_DEVICE_ORDER"] and os.environ["CUDA_VISIBLE_DEVICES"], as long as you do so before using TensorFlow.

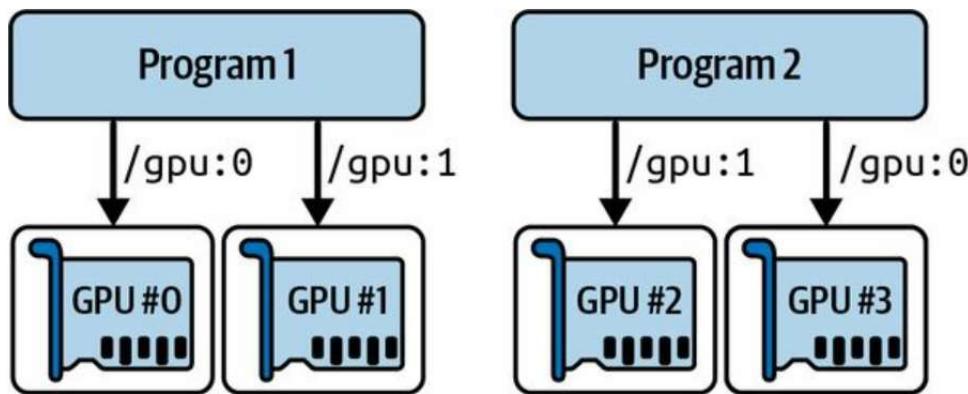


Figure 19-8. Each program gets two GPUs

Another option is to tell TensorFlow to grab only a specific amount of GPU RAM. This must be done immediately after importing TensorFlow. For example, to make TensorFlow grab only 2 GiB of RAM on each GPU, you must create a *logical GPU device* (sometimes called a *virtual GPU device*) for each physical GPU device and set its memory limit to 2 GiB (i.e., 2,048 MiB):

```
for gpu in physical_gpus:
    tf.config.set_logical_device_configuration(
        gpu,
        [tf.config.LogicalDeviceConfiguration(memory_limit=2048)])
)
```

Let's suppose you have four GPUs, each with at least 4 GiB of RAM: in this case, two programs like this one can run in parallel, each using all four GPU cards (see [Figure 19-9](#)). If you run the nvidia-smi command while both programs are running, you should see that each process holds 2 GiB of RAM on each card.

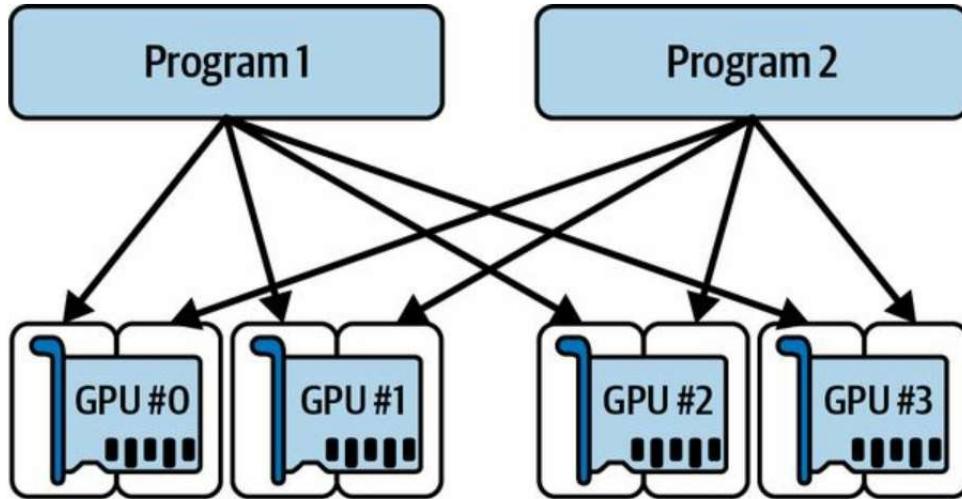


Figure 19-9. Each program gets all four GPUs, but with only 2 GiB of RAM on each GPU

Yet another option is to tell TensorFlow to grab memory only when it needs it. Again, this must be done immediately after importing TensorFlow:

```
for gpu in physical_gpus:  
    tf.config.experimental.set_memory_growth(gpu, True)
```

Another way to do this is to set the `TF_FORCE_GPU_ALLOW_GROWTH` environment variable to true. With this option, TensorFlow will never release memory once it has grabbed it (again, to avoid memory fragmentation), except of course when the program ends. It can be harder to guarantee deterministic behavior using this option (e.g., one program may crash because another program's memory usage went through the roof), so in production you'll probably want to stick with one of the previous options. However, there are some cases where it is very useful: for example, when you use a machine to run multiple Jupyter notebooks, several of which use TensorFlow. The `TF_FORCE_GPU_ALLOW_GROWTH` environment variable is set to true in Colab runtimes.

Lastly, in some cases you may want to split a GPU into two or more *logical devices*. For example, this is useful if you only have one physical GPU—like in a Colab runtime—but you want to test a multi-GPU algorithm. The

following code splits GPU #0 into two logical devices, with 2 GiB of RAM each (again, this must be done immediately after importing TensorFlow):

```
tf.config.set_logical_device_configuration(  
    physical_gpus[0],  
    [tf.config.LogicalDeviceConfiguration(memory_limit=2048),  
     tf.config.LogicalDeviceConfiguration(memory_limit=2048)]  
)
```

These two logical devices are called "/gpu:0" and "/gpu:1", and you can use them as if they were two normal GPUs. You can list all logical devices like this:

```
>>> logical_gpus = tf.config.list_logical_devices("GPU")  
>>> logical_gpus  
[LogicalDevice(name='/device:GPU:0', device_type='GPU'),  
 LogicalDevice(name='/device:GPU:1', device_type='GPU')]
```

Now let's see how TensorFlow decides which devices it should use to place variables and execute operations.

Placing Operations and Variables on Devices

Keras and tf.data generally do a good job of placing operations and variables where they belong, but you can also place operations and variables manually on each device, if you want more control:

- You generally want to place the data preprocessing operations on the CPU, and place the neural network operations on the GPUs.
- GPUs usually have a fairly limited communication bandwidth, so it is important to avoid unnecessary data transfers into and out of the GPUs.
- Adding more CPU RAM to a machine is simple and fairly cheap, so there's usually plenty of it, whereas the GPU RAM is baked into the GPU: it is an expensive and thus limited resource, so if a variable is not needed in the next few training steps, it should probably be placed on the CPU (e.g., datasets generally belong on the CPU).

By default, all variables and all operations will be placed on the first GPU (the one named "/gpu:0"), except for variables and operations that don't have a GPU kernel: ¹⁰ these are placed on the CPU (always named "/cpu:0"). A tensor or variable's device attribute tells you which device it was placed on: ¹¹

```
>>> a = tf.Variable([1., 2., 3.]) # float32 variable goes to the GPU
>>> a.device
'/job:localhost/replica:0/task:0/device:GPU:0'
>>> b = tf.Variable([1, 2, 3]) # int32 variable goes to the CPU
>>> b.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

You can safely ignore the prefix /job:localhost/replica:0/task:0 for now; we will discuss jobs, replicas, and tasks later in this chapter. As you can see, the first variable was placed on GPU #0, which is the default device. However, the second variable was placed on the CPU: this is because there are no GPU kernels for integer variables, or for operations involving integer tensors, so TensorFlow fell back to the CPU.

If you want to place an operation on a different device than the default one, use a `tf.device()` context:

```
>>> with tf.device("/cpu:0"):
...   c = tf.Variable([1., 2., 3.])
...
>>> c.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

NOTE

The CPU is always treated as a single device ("cpu:0"), even if your machine has multiple CPU cores. Any operation placed on the CPU may run in parallel across multiple cores if it has a multithreaded kernel.

If you explicitly try to place an operation or variable on a device that does not exist or for which there is no kernel, then TensorFlow will silently fall back to the device it would have chosen by default. This is useful when you want to be able to run the same code on different machines that don't have the same number of GPUs. However, you can run `tf.config.set_soft_device_placement(False)` if you prefer to get an exception.

Now, how exactly does TensorFlow execute operations across multiple devices?

Parallel Execution Across Multiple Devices

As we saw in [Chapter 12](#), one of the benefits of using TF functions is parallelism. Let's look at this a bit more closely. When TensorFlow runs a TF function, it starts by analyzing its graph to find the list of operations that need to be evaluated, and it counts how many dependencies each of them has. TensorFlow then adds each operation with zero dependencies (i.e., each source operation) to the evaluation queue of this operation's device (see [Figure 19-10](#)). Once an operation has been evaluated, the dependency counter of each operation that depends on it is decremented. Once an operation's dependency counter reaches zero, it is pushed to the evaluation queue of its device. And once all the outputs have been computed, they are returned.

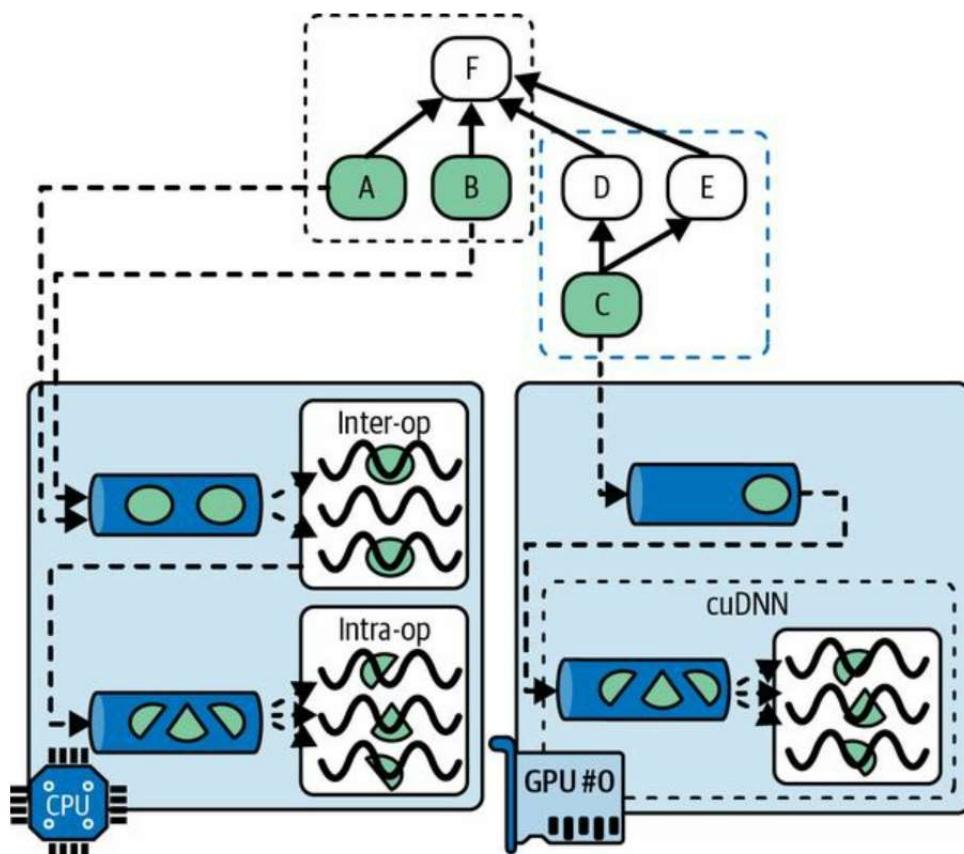


Figure 19-10. Parallelized execution of a TensorFlow graph

Operations in the CPU's evaluation queue are dispatched to a thread pool called the *inter-op thread pool*. If the CPU has multiple cores, then these operations will effectively be evaluated in parallel. Some operations have multithreaded CPU kernels: these kernels split their tasks into multiple suboperations, which are placed in another evaluation queue and dispatched to a second thread pool called the *intra-op thread pool* (shared by all multithreaded CPU kernels). In short, multiple operations and suboperations may be evaluated in parallel on different CPU cores.

For the GPU, things are a bit simpler. Operations in a GPU's evaluation queue are evaluated sequentially. However, most operations have multithreaded GPU kernels, typically implemented by libraries that TensorFlow depends on, such as CUDA and cuDNN. These implementations have their own thread pools, and they typically exploit as many GPU threads as they can (which is the reason why there is no need for an inter-op thread pool in GPUs: each operation already floods most GPU threads).

For example, in [Figure 19-10](#), operations A, B, and C are source ops, so they can immediately be evaluated. Operations A and B are placed on the CPU, so they are sent to the CPU's evaluation queue, then they are dispatched to the inter-op thread pool and immediately evaluated in parallel. Operation A happens to have a multithreaded kernel; its computations are split into three parts, which are executed in parallel by the intra-op thread pool. Operation C goes to GPU #0's evaluation queue, and in this example its GPU kernel happens to use cuDNN, which manages its own intra-op thread pool and runs the operation across many GPU threads in parallel. Suppose C finishes first. The dependency counters of D and E are decremented and they reach 0, so both operations are pushed to GPU #0's evaluation queue, and they are executed sequentially. Note that C only gets evaluated once, even though both D and E depend on it. Suppose B finishes next. Then F's dependency counter is decremented from 4 to 3, and since that's not 0, it does not run yet. Once A, D, and E are finished, then F's dependency counter reaches 0, and it is pushed to the CPU's evaluation queue and evaluated. Finally, TensorFlow returns the requested outputs.

An extra bit of magic that TensorFlow performs is when the TF function modifies a stateful resource, such as a variable: it ensures that the order of execution matches the order in the code, even if there is no explicit dependency between the statements. For example, if your TF function contains `v.assign_add(1)` followed by `v.assign(v * 2)`, TensorFlow will ensure that these operations are executed in that order.

TIP

You can control the number of threads in the inter-op thread pool by calling `tf.config.threading.set_inter_op_parallelism_threads()`. To set the number of intra-op threads, use `tf.config.threading.set_intra_op_parallelism_threads()`. This is useful if you do not want TensorFlow to use all the CPU cores or if you want it to be single-threaded. ¹²

With that, you have all you need to run any operation on any device, and exploit the power of your GPUs! Here are some of the things you could do:

- You could train several models in parallel, each on its own GPU: just write a training script for each model and run them in parallel, setting `CUDA_DEVICE_ORDER` and `CUDA_VISIBLE_DEVICES` so that each script only sees a single GPU device. This is great for hyperparameter tuning, as you can train in parallel multiple models with different hyperparameters. If you have a single machine with two GPUs, and it takes one hour to train one model on one GPU, then training two models in parallel, each on its own dedicated GPU, will take just one hour. Simple!
- You could train a model on a single GPU and perform all the preprocessing in parallel on the CPU, using the dataset's `prefetch()` method ¹³ to prepare the next few batches in advance so that they are ready when the GPU needs them (see [Chapter 13](#)).
- If your model takes two images as input and processes them using two CNNs before joining their outputs, ¹⁴ then it will probably run much faster if you place each CNN on a different GPU.

- You can create an efficient ensemble: just place a different trained model on each GPU so that you can get all the predictions much faster to produce the ensemble's final prediction.

But what if you want to speed up training by using multiple GPUs?

Training Models Across Multiple Devices

There are two main approaches to training a single model across multiple devices: *model parallelism*, where the model is split across the devices, and *data parallelism*, where the model is replicated across every device, and each replica is trained on a different subset of the data. Let's look at these two options.

Model Parallelism

So far we have trained each neural network on a single device. What if we want to train a single neural network across multiple devices? This requires chopping the model into separate chunks and running each chunk on a different device. Unfortunately, such model parallelism turns out to be pretty tricky, and its effectiveness really depends on the architecture of your neural network. For fully connected networks, there is generally not much to be gained from this approach (see [Figure 19-11](#)). Intuitively, it may seem that an easy way to split the model is to place each layer on a different device, but this does not work because each layer needs to wait for the output of the previous layer before it can do anything. So perhaps you can slice it vertically—for example, with the left half of each layer on one device, and the right part on another device? This is slightly better, since both halves of each layer can indeed work in parallel, but the problem is that each half of the next layer requires the output of both halves, so there will be a lot of cross-device communication (represented by the dashed arrows). This is likely to completely cancel out the benefit of the parallel computation, since cross-device communication is slow (and even more so when the devices are located on different machines).

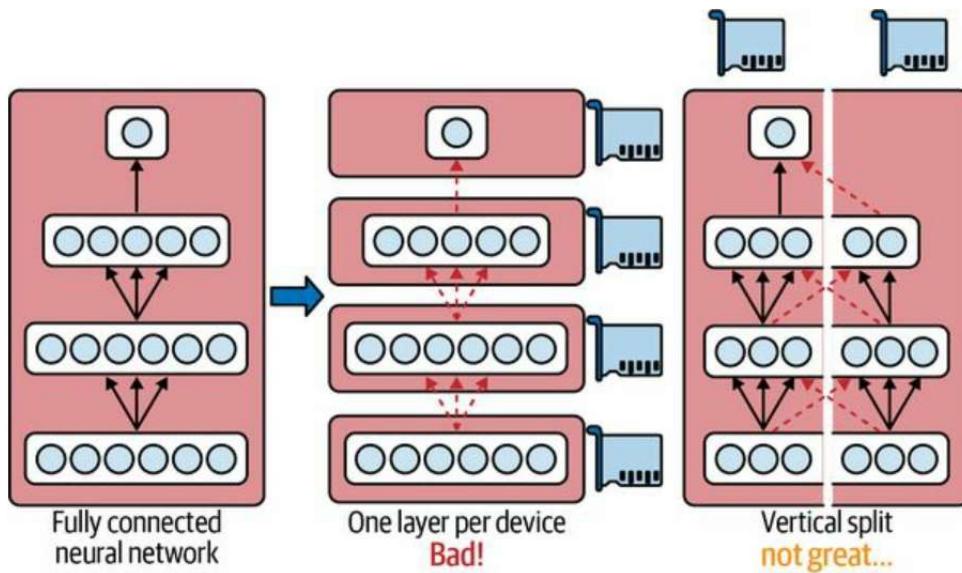


Figure 19-11. Splitting a fully connected neural network

Some neural network architectures, such as convolutional neural networks (see [Chapter 14](#)), contain layers that are only partially connected to the lower layers, so it is much easier to distribute chunks across devices in an efficient way ([Figure 19-12](#)).

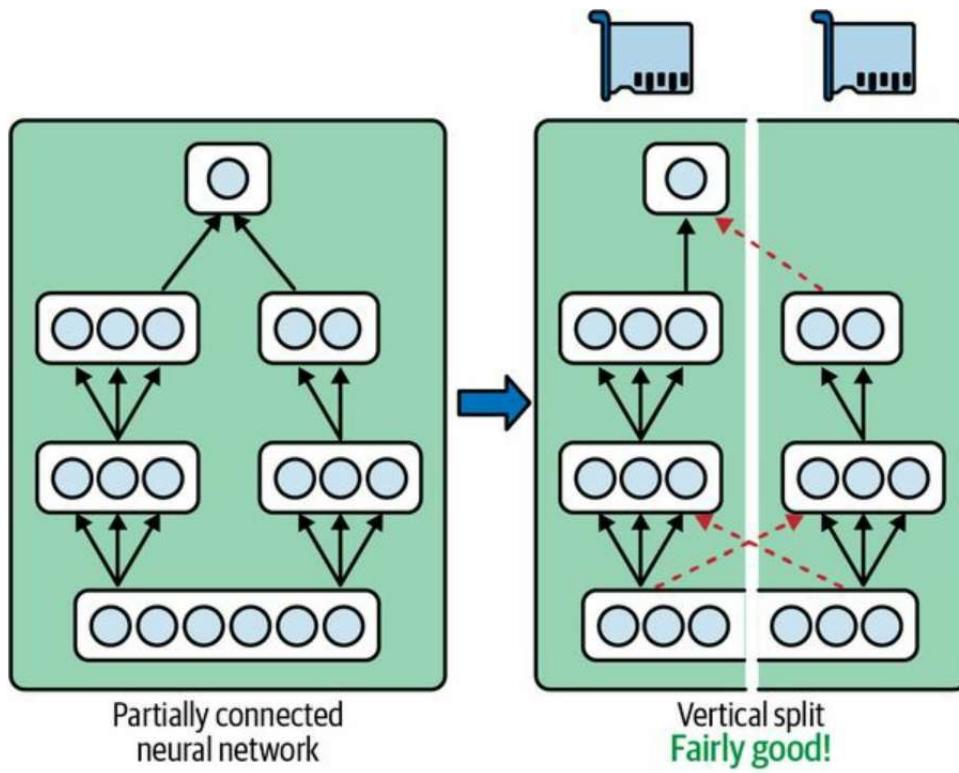


Figure 19-12. Splitting a partially connected neural network

Deep recurrent neural networks (see [Chapter 15](#)) can be split a bit more efficiently across multiple GPUs. If you split the network horizontally by placing each layer on a different device, and feed the network with an input sequence to process, then at the first time step only one device will be active (working on the sequence's first value), at the second step two will be active (the second layer will be handling the output of the first layer for the first value, while the first layer will be handling the second value), and by the time the signal propagates to the output layer, all devices will be active simultaneously ([Figure 19-13](#)). There is still a lot of cross-device communication going on, but since each cell may be fairly complex, the benefit of running multiple cells in parallel may (in theory) outweigh the communication penalty. However, in practice a regular stack of LSTM layers running on a single GPU actually runs much faster.

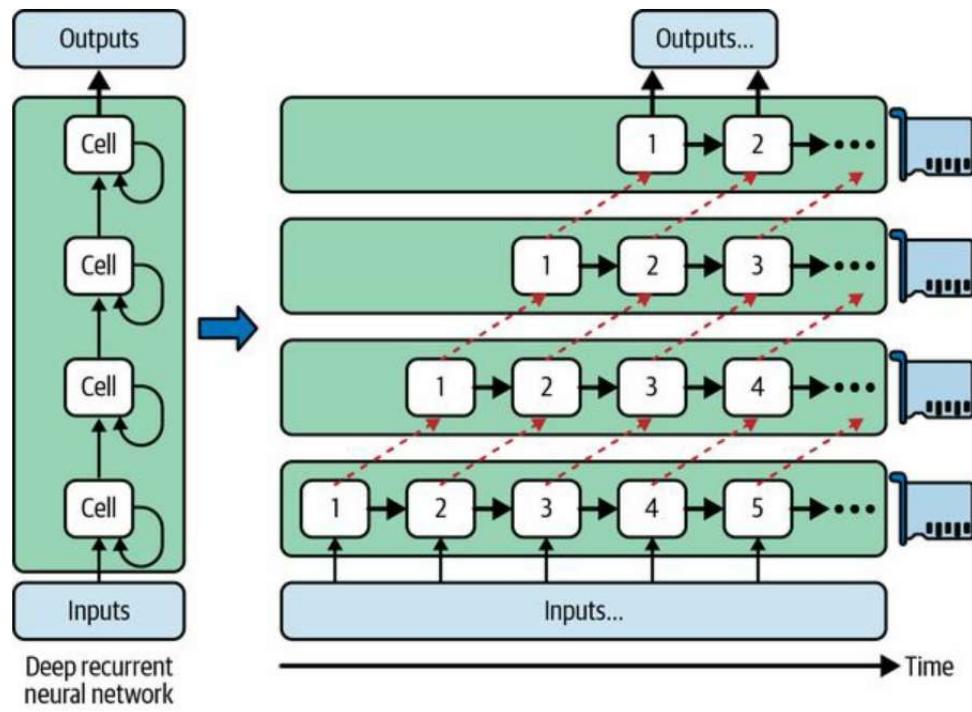


Figure 19-13. Splitting a deep recurrent neural network

In short, model parallelism may speed up running or training some types of neural networks, but not all, and it requires special care and tuning, such as making sure that devices that need to communicate the most run on the same machine.¹⁵ Next we'll look at a much simpler and generally more efficient option: data parallelism.

Data Parallelism

Another way to parallelize the training of a neural network is to replicate it on every device and run each training step simultaneously on all replicas, using a different mini-batch for each. The gradients computed by each replica are then averaged, and the result is used to update the model parameters. This is called *data parallelism*, or sometimes *single program, multiple data* (SPMD). There are many variants of this idea, so let's look at the most important ones.

Data parallelism using the mirrored strategy

Arguably the simplest approach is to completely mirror all the model parameters across all the GPUs and always apply the exact same parameter updates on every GPU. This way, all replicas always remain perfectly identical. This is called the *mirrored strategy*, and it turns out to be quite efficient, especially when using a single machine (see [Figure 19-14](#)).

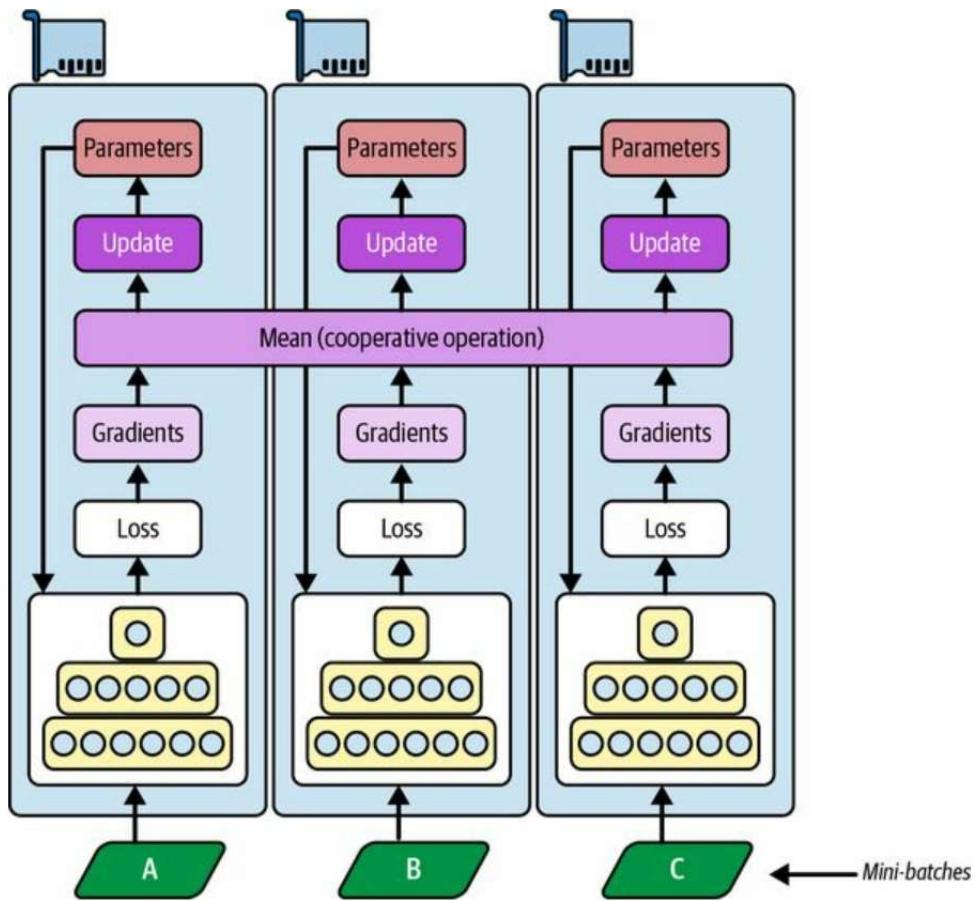


Figure 19-14. Data parallelism using the mirrored strategy

The tricky part when using this approach is to efficiently compute the mean of all the gradients from all the GPUs and distribute the result across all the GPUs. This can be done using an *AllReduce* algorithm, a class of algorithms where multiple nodes collaborate to efficiently perform a *reduce operation* (such as computing the mean, sum, and max), while ensuring that all nodes obtain the same final result. Fortunately, there are off-the-shelf implementations of such algorithms, as you will see.

Data parallelism with centralized parameters

Another approach is to store the model parameters outside of the GPU

devices performing the computations (called *workers*); for example, on the CPU (see [Figure 19-15](#)). In a distributed setup, you may place all the parameters on one or more CPU-only servers called *parameter servers*, whose only role is to host and update the parameters.

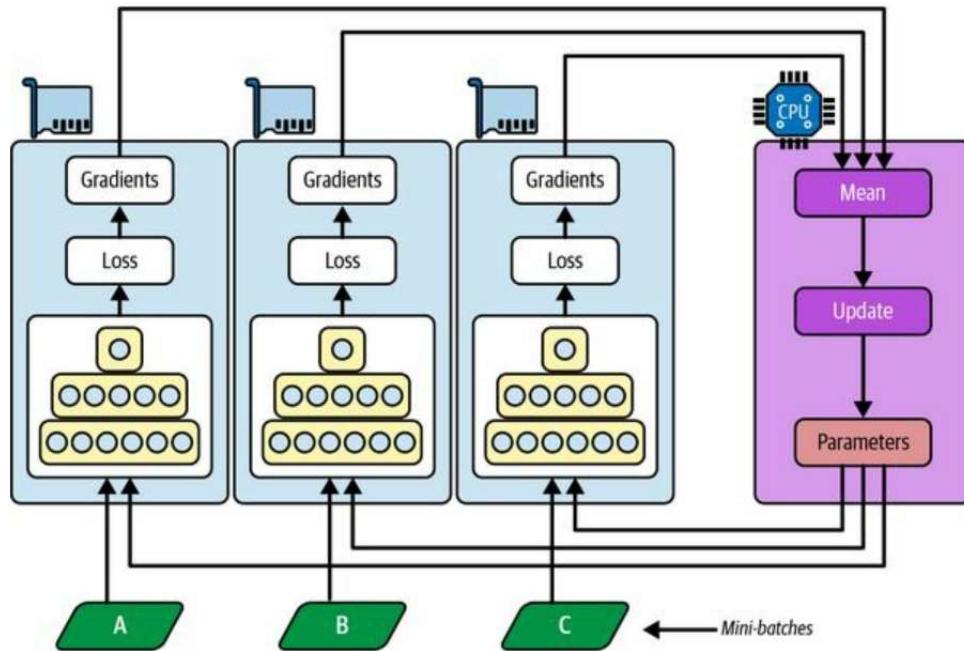


Figure 19-15. Data parallelism with centralized parameters

Whereas the mirrored strategy imposes synchronous weight updates across all GPUs, this centralized approach allows either synchronous or asynchronous updates. Let's take a look at the pros and cons of both options.

Synchronous updates

With *synchronous updates*, the aggregator waits until all gradients are available before it computes the average gradients and passes them to the optimizer, which will update the model parameters. Once a replica has finished computing its gradients, it must wait for the parameters to be updated before it can proceed to the next mini-batch. The downside is that some devices may be slower than others, so the fast devices will have to wait for the slow ones at every step, making the whole process as slow as the

slowest device. Moreover, the parameters will be copied to every device almost at the same time (immediately after the gradients are applied), which may saturate the parameter servers' bandwidth.

TIP

To reduce the waiting time at each step, you could ignore the gradients from the slowest few replicas (typically $\sim 10\%$). For example, you could run 20 replicas, but only aggregate the gradients from the fastest 18 replicas at each step, and just ignore the gradients from the last 2. As soon as the parameters are updated, the first 18 replicas can start working again immediately, without having to wait for the 2 slowest replicas. This setup is generally described as having 18 replicas plus 2 *spare replicas*.¹⁶

Asynchronous updates

With asynchronous updates, whenever a replica has finished computing the gradients, the gradients are immediately used to update the model parameters. There is no aggregation (it removes the “mean” step in [Figure 19-15](#)) and no synchronization. Replicas work independently of the other replicas. Since there is no waiting for the other replicas, this approach runs more training steps per minute. Moreover, although the parameters still need to be copied to every device at every step, this happens at different times for each replica, so the risk of bandwidth saturation is reduced.

Data parallelism with asynchronous updates is an attractive choice because of its simplicity, the absence of synchronization delay, and its better use of the bandwidth. However, although it works reasonably well in practice, it is almost surprising that it works at all! Indeed, by the time a replica has finished computing the gradients based on some parameter values, these parameters will have been updated several times by other replicas (on average $N - 1$ times, if there are N replicas), and there is no guarantee that the computed gradients will still be pointing in the right direction (see [Figure 19-16](#)). When gradients are severely out of date, they are called *stale gradients*: they can slow down convergence, introducing noise and wobble effects (the learning curve may contain temporary oscillations), or they can even make the training algorithm diverge.

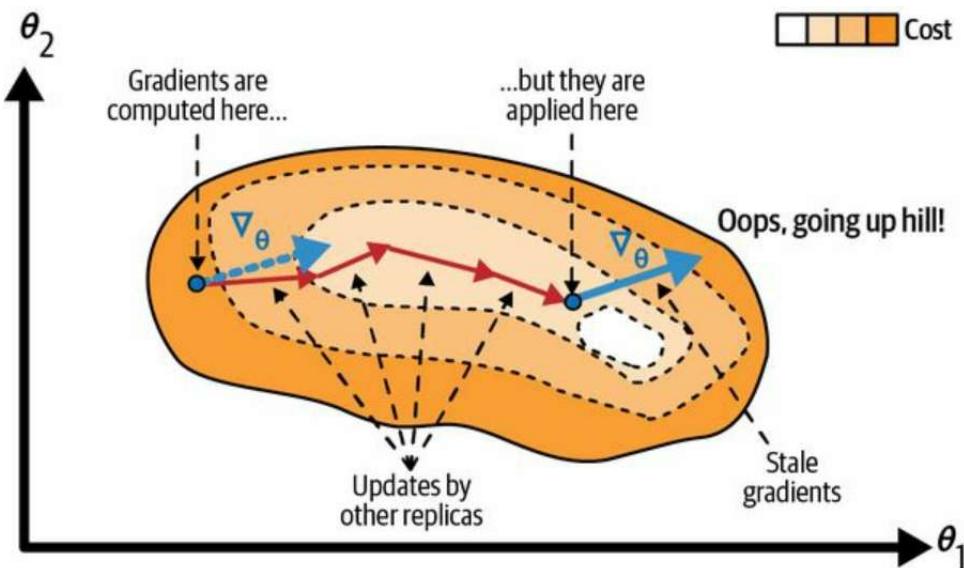


Figure 19-16. Stale gradients when using asynchronous updates

There are a few ways you can reduce the effect of stale gradients:

- Reduce the learning rate.
- Drop stale gradients or scale them down.
- Adjust the mini-batch size.
- Start the first few epochs using just one replica (this is called the *warmup phase*). Stale gradients tend to be more damaging at the beginning of training, when gradients are typically large and the parameters have not settled into a valley of the cost function yet, so different replicas may push the parameters in quite different directions.

A paper published by the Google Brain team in 2016¹⁷ benchmarked various approaches and found that using synchronous updates with a few spare replicas was more efficient than using asynchronous updates, not only converging faster but also producing a better model. However, this is still an active area of research, so you should not rule out asynchronous updates just yet.

Bandwidth saturation

Whether you use synchronous or asynchronous updates, data parallelism with centralized parameters still requires communicating the model parameters from the parameter servers to every replica at the beginning of each training step, and the gradients in the other direction at the end of each training step. Similarly, when using the mirrored strategy, the gradients produced by each GPU will need to be shared with every other GPU. Unfortunately, there often comes a point where adding an extra GPU will not improve performance at all because the time spent moving the data into and out of GPU RAM (and across the network in a distributed setup) will outweigh the speedup obtained by splitting the computation load. At that point, adding more GPUs will just worsen the bandwidth saturation and actually slow down training.

Saturation is more severe for large dense models, since they have a lot of parameters and gradients to transfer. It is less severe for small models (but the parallelization gain is limited) and for large sparse models, where the gradients are typically mostly zeros and so can be communicated efficiently. Jeff Dean, initiator and lead of the Google Brain project, [reported](#) typical speedups of $25\text{--}40\times$ when distributing computations across 50 GPUs for dense models, and a $300\times$ speedup for sparser models trained across 500 GPUs. As you can see, sparse models really do scale better. Here are a few concrete examples:

- Neural machine translation: $6\times$ speedup on 8 GPUs
- Inception/ImageNet: $32\times$ speedup on 50 GPUs
- RankBrain: $300\times$ speedup on 500 GPUs

There is plenty of research going on to alleviate the bandwidth saturation issue, with the goal of allowing training to scale linearly with the number of GPUs available. For example, a [2018 paper¹⁸](#) by a team of researchers from Carnegie Mellon University, Stanford University, and Microsoft Research proposed a system called *PipeDream* that managed to reduce network communications by over 90%, making it possible to train large models across many machines. They achieved this using a new technique called *pipeline*

parallelism, which combines model parallelism and data parallelism: the model is chopped into consecutive parts, called *stages*, each of which is trained on a different machine. This results in an asynchronous pipeline in which all machines work in parallel with very little idle time. During training, each stage alternates one round of forward propagation and one round of backpropagation (see [Figure 19-17](#)): it pulls a mini-batch from its input queue, processes it, and sends the outputs to the next stage's input queue, then it pulls one mini-batch of gradients from its gradient queue, backpropagates these gradients and updates its own model parameters, and pushes the backpropagated gradients to the previous stage's gradient queue. It then repeats the whole process again and again. Each stage can also use regular data parallelism (e.g., using the mirrored strategy), independently from the other stages.

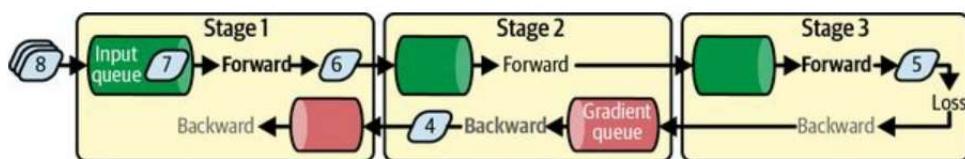


Figure 19-17. PipeDream's pipeline parallelism

However, as it's presented here, PipeDream would not work so well. To understand why, consider mini-batch #5 in [Figure 19-17](#): when it went through stage 1 during the forward pass, the gradients from mini-batch #4 had not yet been backpropagated through that stage, but by the time #5's gradients flow back to stage 1, #4's gradients will have been used to update the model parameters, so #5's gradients will be a bit stale. As we have seen, this can degrade training speed and accuracy, and even make it diverge: the more stages there are, the worse this problem becomes. The paper's authors proposed methods to mitigate this issue, though: for example, each stage saves weights during forward propagation and restores them during backpropagation, to ensure that the same weights are used for both the forward pass and the backward pass. This is called *weight stashing*. Thanks to this, PipeDream demonstrates impressive scaling capability, well beyond simple data parallelism.

The latest breakthrough in this field of research was published in a [2022](#)

[paper¹⁹](#) by Google researchers: they developed a system called *Pathways* that uses automated model parallelism, asynchronous gang scheduling, and other techniques to reach close to 100% hardware utilization across thousands of TPUs! *Scheduling* means organizing when and where each task must run, and *gang scheduling* means running related tasks at the same time in parallel and close to each other to reduce the time tasks have to wait for the others' outputs. As we saw in [Chapter 16](#), this system was used to train a massive language model across over 6,000 TPUs, with close to 100% hardware utilization: that's a mindblowing engineering feat.

At the time of writing, *Pathways* is not public yet, but it's likely that in the near future you will be able to train huge models on Vertex AI using *Pathways* or a similar system. In the meantime, to reduce the saturation problem, you'll probably want to use a few powerful GPUs rather than plenty of weak GPUs, and if you need to train a model across multiple servers, you should group your GPUs on few and very well interconnected servers. You can also try dropping the float precision from 32 bits (`tf.float32`) to 16 bits (`tf.bfloat16`). This will cut in half the amount of data to transfer, often without much impact on the convergence rate or the model's performance. Lastly, if you are using centralized parameters, you can shard (split) the parameters across multiple parameter servers: adding more parameter servers will reduce the network load on each server and limit the risk of bandwidth saturation.

OK, now that we've gone through all the theory, let's actually train a model across multiple GPUs!

Training at Scale Using the Distribution Strategies API

Luckily, TensorFlow comes with a very nice API that takes care of all the complexity of distributing your model across multiple devices and machines: the *distribution strategies API*. To train a Keras model across all available GPUs (on a single machine, for now) using data parallelism with the mirrored strategy, just create a MirroredStrategy object, call its scope() method to get a distribution context, and wrap the creation and compilation of your model inside that context. Then call the model's fit() method normally:

```
strategy = tf.distribute.MirroredStrategy()

with strategy.scope():
    model = tf.keras.Sequential([...]) # create a Keras model normally
    model.compile([...]) # compile the model normally

batch_size = 100 # preferably divisible by the number of replicas
model.fit(X_train, y_train, epochs=10,
           validation_data=(X_valid, y_valid), batch_size=batch_size)
```

Under the hood, Keras is distribution-aware, so in this MirroredStrategy context it knows that it must replicate all variables and operations across all available GPU devices. If you look at the model's weights, they are of type MirroredVariable:

```
>>> type(model.weights[0])
tensorflow.distribute.values.MirroredVariable
```

Note that the fit() method will automatically split each training batch across all the replicas, so it's preferable to ensure that the batch size is divisible by the number of replicas (i.e., the number of available GPUs) so that all replicas get batches of the same size. And that's all! Training will generally be significantly faster than using a single device, and the code change was really minimal.

Once you have finished training your model, you can use it to make predictions efficiently: call the predict() method, and it will automatically

split the batch across all replicas, making predictions in parallel. Again, the batch size must be divisible by the number of replicas. If you call the model’s `save()` method, it will be saved as a regular model, *not* as a mirrored model with multiple replicas. So when you load it, it will run like a regular model, on a single device: by default on GPU #0, or on the CPU if there are no GPUs. If you want to load a model and run it on all available devices, you must call `tf.keras.models.load_model()` within a distribution context:

```
with strategy.scope():
    model = tf.keras.models.load_model("my_mirrored_model")
```

If you only want to use a subset of all the available GPU devices, you can pass the list to the `MirroredStrategy`’s constructor:

```
strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
```

By default, the `MirroredStrategy` class uses the *NVIDIA Collective Communications Library* (NCCL) for the AllReduce mean operation, but you can change it by setting the `cross_device_ops` argument to an instance of the `tf.distribute.HierarchicalCopyAllReduce` class, or an instance of the `tf.distribute.ReductionToOneDevice` class. The default NCCL option is based on the `tf.distribute.NcclAllReduce` class, which is usually faster, but this depends on the number and types of GPUs, so you may want to give the alternatives a try.²⁰

If you want to try using data parallelism with centralized parameters, replace the `MirroredStrategy` with the `CentralStorageStrategy`:

```
strategy = tf.distribute.experimental.CentralStorageStrategy()
```

You can optionally set the `compute_devices` argument to specify the list of devices you want to use as workers—by default it will use all available GPUs—and you can optionally set the `parameter_device` argument to specify the device you want to store the parameters on. By default it will use the CPU, or the GPU if there is just one.

Now let's see how to train a model across a cluster of TensorFlow servers!

Training a Model on a TensorFlow Cluster

A *TensorFlow cluster* is a group of TensorFlow processes running in parallel, usually on different machines, and talking to each other to complete some work—for example, training or executing a neural network model. Each TF process in the cluster is called a *task*, or a *TF server*. It has an IP address, a port, and a type (also called its *role* or its *job*). The type can be either "worker", "chief", "ps" (parameter server), or "evaluator":

- Each *worker* performs computations, usually on a machine with one or more GPUs.
- The *chief* performs computations as well (it is a worker), but it also handles extra work such as writing TensorBoard logs or saving checkpoints. There is a single chief in a cluster. If no chief is specified explicitly, then by convention the first worker is the chief.
- A *parameter server* only keeps track of variable values, and it is usually on a CPU-only machine. This type of task is only used with the ParameterServerStrategy.
- An *evaluator* obviously takes care of evaluation. This type is not used often, and when it's used, there's usually just one evaluator.

To start a TensorFlow cluster, you must first define its specification. This means defining each task's IP address, TCP port, and type. For example, the following *cluster specification* defines a cluster with three tasks (two workers and one parameter server; see Figure 19-18). The cluster spec is a dictionary with one key per job, and the values are lists of task addresses (*IP:port*):

```
cluster_spec = {
    "worker": [
        "machine-a.example.com:2222",  # /job:worker/task:0
        "machine-b.example.com:2222"   # /job:worker/task:1
    ],
    "ps": ["machine-a.example.com:2221"] # /job:ps/task:0
}
```

In general there will be a single task per machine, but as this example shows, you can configure multiple tasks on the same machine if you want. In this case, if they share the same GPUs, make sure the RAM is split appropriately, as discussed earlier.

WARNING

By default, every task in the cluster may communicate with every other task, so make sure to configure your firewall to authorize all communications between these machines on these ports (it's usually simpler if you use the same port on every machine).

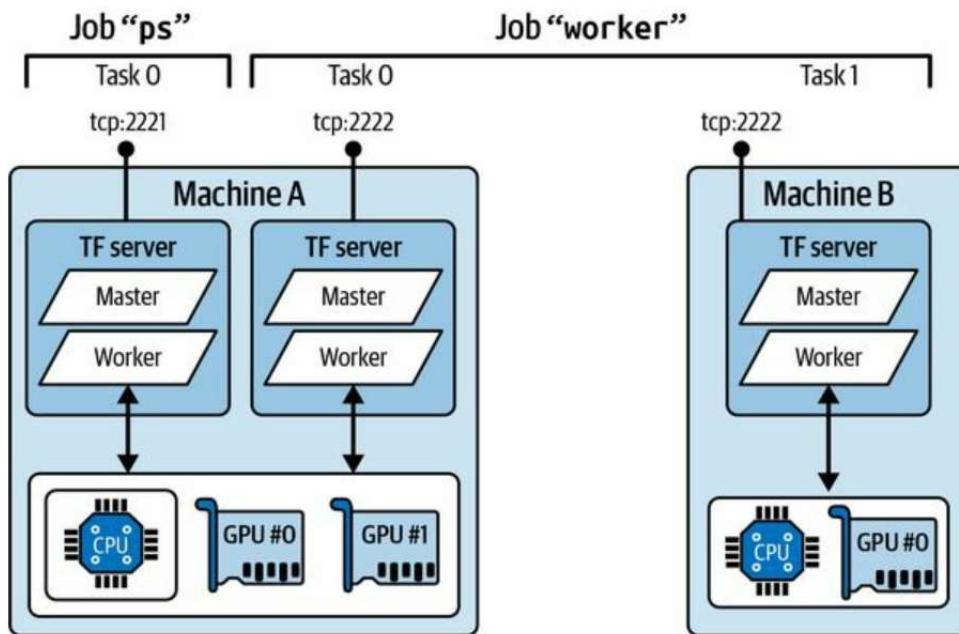


Figure 19-18. An example TensorFlow cluster

When you start a task, you must give it the cluster spec, and you must also tell it what its type and index are (e.g., worker #0). The simplest way to specify everything at once (both the cluster spec and the current task's type and index) is to set the `TF_CONFIG` environment variable before starting TensorFlow. It must be a JSON-encoded dictionary containing a cluster specification (under the "cluster" key) and the type and index of the current

task (under the "task" key). For example, the following TF_CONFIG environment variable uses the cluster we just defined and specifies that the task to start is worker #0:

```
os.environ["TF_CONFIG"] = json.dumps({  
    "cluster": cluster_spec,  
    "task": {"type": "worker", "index": 0}  
})
```

TIP

In general you want to define the TF_CONFIG environment variable outside of Python, so the code does not need to include the current task's type and index (this makes it possible to use the same code across all workers).

Now let's train a model on a cluster! We will start with the mirrored strategy. First, you need to set the TF_CONFIG environment variable appropriately for each task. There should be no parameter server (remove the "ps" key in the cluster spec), and in general you will want a single worker per machine. Make extra sure you set a different task index for each task. Finally, run the following script on every worker:

```
import tempfile  
import tensorflow as tf  
  
strategy = tf.distribute.MultiWorkerMirroredStrategy() # at the start!  
resolver = tf.distribute.cluster_resolver.TFConfigClusterResolver()  
print(f"Starting task {resolver.task_type} #{resolver.task_id}")  
[...] # load and split the MNIST dataset  
  
with strategy.scope():  
    model = tf.keras.Sequential([...]) # build the Keras model  
    model.compile([...]) # compile the model  
  
model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=10)  
  
if resolver.task_id == 0: # the chief saves the model to the right location  
    model.save("my_mnist_multiworker_model", save_format="tf")  
else:  
    tmpdir = tempfile.mkdtemp() # other workers save to a temporary directory
```

```
model.save(tmpdir, save_format="tf")
tf.io.gfile.rmtree(tmpdir) # and we can delete this directory at the end!
```

That's almost the same code you used earlier, except this time you are using the `MultiWorkerMirroredStrategy`. When you start this script on the first workers, they will remain blocked at the `AllReduce` step, but training will begin as soon as the last worker starts up, and you will see them all advancing at exactly the same rate since they synchronize at each step.

WARNING

When using the `MultiWorkerMirroredStrategy`, it's important to ensure that all workers do the same thing, including saving model checkpoints or writing TensorBoard logs, even though you will only keep what the chief writes. This is because these operations may need to run the `AllReduce` operations, so all workers must be in sync.

There are two `AllReduce` implementations for this distribution strategy: a ring `AllReduce` algorithm based on gRPC for the network communications, and NCCL's implementation. The best algorithm to use depends on the number of workers, the number and types of GPUs, and the network. By default, TensorFlow will apply some heuristics to select the right algorithm for you, but you can force NCCL (or RING) like this:

```
strategy = tf.distribute.MultiWorkerMirroredStrategy(
    communication_options=tf.distribute.experimental.CommunicationOptions(
        implementation=tf.distribute.experimental.CollectiveCommunication.NCCL))
```

If you prefer to implement asynchronous data parallelism with parameter servers, change the strategy to `ParameterServerStrategy`, add one or more parameter servers, and configure `TF_CONFIG` appropriately for each task. Note that although the workers will work asynchronously, the replicas on each worker will work synchronously.

Lastly, if you have access to [TPUs on Google Cloud](#)—for example, if you use Colab and you set the accelerator type to TPU—then you can create a `TPUStrategy` like this:

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()  
tf.tpu.experimental.initialize_tpu_system(resolver)  
strategy = tf.distribute.experimental.TPUStrategy(resolver)
```

This needs to be run right after importing TensorFlow. You can then use this strategy normally.

TIP

If you are a researcher, you may be eligible to use TPUs for free; see <https://tensorflow.org/tfrc> for more details.

You can now train models across multiple GPUs and multiple servers: give yourself a pat on the back! If you want to train a very large model, however, you will need many GPUs, across many servers, which will require either buying a lot of hardware or managing a lot of cloud virtual machines. In many cases, it's less hassle and less expensive to use a cloud service that takes care of provisioning and managing all this infrastructure for you, just when you need it. Let's see how to do that using Vertex AI.

Running Large Training Jobs on Vertex AI

Vertex AI allows you to create custom training jobs with your own training code. In fact, you can use almost the same training code as you would use on your own TF cluster. The main thing you must change is where the chief should save the model, the checkpoints, and the TensorBoard logs. Instead of saving the model to a local directory, the chief must save it to GCS, using the path provided by Vertex AI in the AIP_MODEL_DIR environment variable. For the model checkpoints and TensorBoard logs, you should use the paths contained in the AIP_CHECKPOINT_DIR and AIP_TENSORBOARD_LOG_DIR environment variables, respectively. Of course, you must also make sure that the training data can be accessed from the virtual machines, such as on GCS, or another GCP service like BigQuery, or directly from the web. Lastly, Vertex AI sets the "chief" task type explicitly, so you should identify the chief using resolved.task_type == "chief" instead of resolved.task_id == 0:

```
import os
[...] # other imports, create MultiWorkerMirroredStrategy, and resolver

if resolver.task_type == "chief":
    model_dir = os.getenv("AIP_MODEL_DIR") # paths provided by Vertex AI
    tensorboard_log_dir = os.getenv("AIP_TENSORBOARD_LOG_DIR")
    checkpoint_dir = os.getenv("AIP_CHECKPOINT_DIR")
else:
    tmp_dir = Path(tempfile.mkdtemp()) # other workers use temporary dirs
    model_dir = tmp_dir / "model"
    tensorboard_log_dir = tmp_dir / "logs"
    checkpoint_dir = tmp_dir / "ckpt"

callbacks = [tf.keras.callbacks.TensorBoard(tensorboard_log_dir),
            tf.keras.callbacks.ModelCheckpoint(checkpoint_dir)]
[...] # build and compile using the strategy scope, just like earlier
model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=10,
           callbacks=callbacks)
model.save(model_dir, save_format="tf")
```

TIP

If you place the training data on GCS, you can create a `tf.data.TextLineDataset` or `tf.data.TFRecordDataset` to access it: just use the GCS paths as the filenames (e.g., `gs://my_bucket/data/001.csv`). These datasets rely on the `tf.io.gfile` package to access files: it supports both local files and GCS files.

Now you can create a custom training job on Vertex AI, based on this script. You'll need to specify the job name, the path to your training script, the Docker image to use for training, the one to use for predictions (after training), any additional Python libraries you may need, and lastly the bucket that Vertex AI should use as a staging directory to store the training script. By default, that's also where the training script will save the trained model, as well as the TensorBoard logs and model checkpoints (if any). Let's create the job:

```
custom_training_job = aiplatform.CustomTrainingJob(  
    display_name="my_custom_training_job",  
    script_path="my_vertex_ai_training_task.py",  
    container_uri="gcr.io/cloud-aiplatform/training/tf-gpu.2-4:latest",  
    model_serving_container_image_uri=server_image,  
    requirements=["gcsfs==2022.3.0"], # not needed, this is just an example  
    staging_bucket=f"gs://{bucket_name}/staging"  
)
```

And now let's run it on two workers, each with two GPUs:

```
mnist_model2 = custom_training_job.run(  
    machine_type="n1-standard-4",  
    replica_count=2,  
    accelerator_type="NVIDIA_TESLA_K80",  
    accelerator_count=2,  
)
```

And that's it: Vertex AI will provision the compute nodes you requested (within your quotas), and it will run your training script across them. Once the job is complete, the `run()` method will return a trained model that you can use exactly like the one you created earlier: you can deploy it to an endpoint, or use it to make batch predictions. If anything goes wrong during training, you can view the logs in the GCP console: in the  navigation menu, select

Vertex AI → Training, click on your training job, and click VIEW LOGS. Alternatively, you can click the CUSTOM JOBS tab and copy the job's ID (e.g., 1234), then select Logging from the  navigation menu and query resource.labels.job_id=1234.

TIP

To visualize the training progress, just start TensorBoard and point its --logdir to the GCS path of the logs. It will use *application default credentials*, which you can set up using gcloud auth application-default login. Vertex AI also offers hosted TensorBoard servers if you prefer.

If you want to try out a few hyperparameter values, one option is to run multiple jobs. You can pass the hyperparameter values to your script as command-line arguments by setting the args parameter when calling the run() method, or you can pass them as environment variables using the environment_variables parameter.

However, if you want to run a large hyperparameter tuning job on the cloud, a much better option is to use Vertex AI's hyperparameter tuning service. Let's see how.

Hyperparameter Tuning on Vertex AI

Vertex AI's hyperparameter tuning service is based on a Bayesian optimization algorithm, capable of quickly finding optimal combinations of hyperparameters. To use it, you first need to create a training script that accepts hyperparameter values as command-line arguments. For example, your script could use the argparse standard library like this:

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("--n_hidden", type=int, default=2)
parser.add_argument("--n_neurons", type=int, default=256)
parser.add_argument("--learning_rate", type=float, default=1e-2)
parser.add_argument("--optimizer", default="adam")
args = parser.parse_args()
```

The hyperparameter tuning service will call your script multiple times, each time with different hyperparameter values: each run is called a *trial*, and the set of trials is called a *study*. Your training script must then use the given hyperparameter values to build and compile a model. You can use a mirrored distribution strategy if you want, in case each trial runs on a multi-GPU machine. Then the script can load the dataset and train the model. For example:

```
[...] # load the dataset  
model = build_model(args)  
history = model.fit([...])
```

TIP

You can use the AIP_* environment variables we mentioned earlier to determine where to save the checkpoints, the TensorBoard logs, and the final model.

Lastly, the script must report the model's performance back to Vertex AI's hyperparameter tuning service, so it can decide which hyperparameters to try next. For this, you must use the hypertune library, which is automatically installed on Vertex AI training VMs:

```
import hypertune  
  
hypertune = hypertune.HyperTune()  
hypertune.report_hyperparameter_tuning_metric(  
    hyperparameter_metric_tag="accuracy", # name of the reported metric  
    metric_value=max(history.history["val_accuracy"]), # metric value  
    global_step=model.optimizer.iterations.numpy(),  
)
```

Now that your training script is ready, you need to define the type of machine you would like to run it on. For this, you must define a custom job, which Vertex AI will use as a template for each trial:

```
trial_job = aiplatform.CustomJob.from_local_script(  
    display_name="my_search_trial_job",  
    script_path="my_vertex_ai_trial.py", # path to your training script  
    container_uri="gcr.io/cloud-aiplatform/training/tf-gpu.2-4:latest",  
    staging_bucket=f"gs://{bucket_name}/staging",  
    accelerator_type="NVIDIA_TESLA_K80",  
    accelerator_count=2, # in this example, each trial will have 2 GPUs  
)
```

Finally, you're ready to create and run the hyperparameter tuning job:

```
from google.cloud.aiplatform import hyperparameter_tuning as hpt
```

```

hp_job = aiplatform.HyperparameterTuningJob(
    display_name="my_hp_search_job",
    custom_job=trial_job,
    metric_spec={"accuracy": "maximize"},
    parameter_spec={
        "learning_rate": hpt.DoubleParameterSpec(min=1e-3, max=10, scale="log"),
        "n_neurons": hpt.IntegerParameterSpec(min=1, max=300, scale="linear"),
        "n_hidden": hpt.IntegerParameterSpec(min=1, max=10, scale="linear"),
        "optimizer": hpt.CategoricalParameterSpec(["sgd", "adam"]),
    },
    max_trial_count=100,
    parallel_trial_count=20,
)
hp_job.run()

```

Here, we tell Vertex AI to maximize the metric named "accuracy": this name must match the name of the metric reported by the training script. We also define the search space, using a log scale for the learning rate and a linear (i.e., uniform) scale for the other hyperparameters. The hyperparameter names must match the command-line arguments of the training script. Then we set the maximum number of trials to 100, and the maximum number of trials running in parallel to 20. If you increase the number of parallel trials to (say) 60, the total search time will be reduced significantly, by a factor of up to 3. But the first 60 trials will be started in parallel, so they will not benefit from the other trials' feedback. Therefore, you should increase the max number of trials to compensate—for example, up to about 140.

This will take quite a while. Once the job is completed, you can fetch the trial results using `hp_job.trials`. Each trial result is represented as a protobuf object, containing the hyperparameter values and the resulting metrics. Let's find the best trial:

```

def get_final_metric(trial, metric_id):
    for metric in trial.final_measurement.metrics:
        if metric.metric_id == metric_id:
            return metric.value

trials = hp_job.trials
trial_accuracies = [get_final_metric(trial, "accuracy") for trial in trials]
best_trial = trials[np.argmax(trial_accuracies)]

```

Now let's look at this trial's accuracy, and its hyperparameter values:

```
>>> max(trial_accuracies)
0.977400004863739
>>> best_trial.id
'98'
>>> best_trial.parameters
[parameter_id: "learning_rate" value { number_value: 0.001 },
 parameter_id: "n_hidden" value { number_value: 8.0 },
 parameter_id: "n_neurons" value { number_value: 216.0 },
 parameter_id: "optimizer" value { string_value: "adam" }
]
```

That's it! Now you can get this trial's SavedModel, optionally train it a bit more, and deploy it to production.

TIP

Vertex AI also includes an AutoML service, which completely takes care of finding the right model architecture and training it for you. All you need to do is upload your dataset to Vertex AI using a special format that depends on the type of dataset (images, text, tabular, video, etc.), then create an AutoML training job, pointing to the dataset and specifying the maximum number of compute hours you're willing to spend. See the notebook for an example.

HYPERPARAMETER TUNING USING KERAS TUNER ON VERTEX AI

Instead of using Vertex AI's hyperparameter tuning service, you can use Keras Tuner (introduced in [Chapter 10](#)) and run it on Vertex AI VMs. Keras Tuner provides a simple way to scale hyperparameter search by distributing it across multiple machines: it only requires setting three environment variables on each machine, then running your regular Keras Tuner code on each machine. You can use the exact same script on all machines. One of the machines acts as the chief (i.e., the oracle), and the others act as workers. Each worker asks the chief which hyperparameter values to try, then the worker trains the model using these

hyperparameter values, and finally it reports the model's performance back to the chief, which can then decide which hyperparameter values the worker should try next.

The three environment variables you need to set on each machine are:

KERASTUNER_TUNER_ID

This is equal to "chief" on the chief machine, or a unique identifier on each worker machine, such as "worker0", "worker1", etc.

KERASTUNER_ORACLE_IP

This is the IP address or hostname of the chief machine. The chief itself should generally use "0.0.0.0" to listen on every IP address on the machine.

KERASTUNER_ORACLE_PORT

This is the TCP port that the chief will be listening on.

You can distribute Keras Tuner across any set of machines. If you want to run it on Vertex AI machines, then you can spawn a regular training job, and just modify the training script to set the environment variables properly before using Keras Tuner. See the notebook for an example.

Now you have all the tools and knowledge you need to create state-of-the-art neural net architectures and train them at scale using various distribution strategies, on your own infrastructure or on the cloud, and then deploy them anywhere. In other words, you now have superpowers: use them well!

Exercises

1. What does a SavedModel contain? How do you inspect its content?
2. When should you use TF Serving? What are its main features? What are some tools you can use to deploy it?
3. How do you deploy a model across multiple TF Serving instances?
4. When should you use the gRPC API rather than the REST API to query a model served by TF Serving?
5. What are the different ways TFLite reduces a model's size to make it run on a mobile or embedded device?
6. What is quantization-aware training, and why would you need it?
7. What are model parallelism and data parallelism? Why is the latter generally recommended?
8. When training a model across multiple servers, what distribution strategies can you use? How do you choose which one to use?
9. Train a model (any model you like) and deploy it to TF Serving or Google Vertex AI. Write the client code to query it using the REST API or the gRPC API. Update the model and deploy the new version. Your client code will now query the new version. Roll back to the first version.
10. Train any model across multiple GPUs on the same machine using the MirroredStrategy (if you do not have access to GPUs, you can use Google Colab with a GPU runtime and create two logical GPUs). Train the model again using the CentralStorageStrategy and compare the training time.
11. Fine-tune a model of your choice on Vertex AI, using either Keras Tuner or Vertex AI's hyperparameter tuning service.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

Thank You!

Before we close the last chapter of this book, I would like to thank you for reading it up to the last paragraph. I truly hope that you had as much fun reading this book as I had writing it, and that it will be useful for your projects, big or small.

If you find errors, please send feedback. More generally, I would love to know what you think, so please don't hesitate to contact me via O'Reilly, through the [ageron/handson-ml3](https://github.com/ageron/handson-ml3) GitHub project, or on Twitter at [@aureliengeron](https://twitter.com/aureliengeron).

Going forward, my best advice to you is to practice and practice: try going through all the exercises (if you have not done so already), play with the notebooks, join Kaggle or some other ML community, watch ML courses, read papers, attend conferences, and meet experts. Things move fast, so try to keep up to date. Several YouTube channels regularly present deep learning papers in great detail, in a very approachable way. I particularly recommend the channels by Yannic Kilcher, Letitia Parcalabescu, and Xander Steenbrugge. For fascinating ML discussions and higher-level insights, make sure to check out ML Street Talk, and Lex Fridman's channel. It also helps tremendously to have a concrete project to work on, whether it is for work or for fun (ideally for both), so if there's anything you have always dreamed of building, give it a shot! Work incrementally; don't shoot for the moon right away, but stay focused on your project and build it piece by piece. It will require patience and perseverance, but when you have a walking robot, or a working chatbot, or whatever else you fancy building, it will be immensely rewarding!

My greatest hope is that this book will inspire you to build a wonderful ML application that will benefit all of us. What will it be?

—Aurélien Géron

¹ An A/B experiment consists in testing two different versions of your product on different

subsets of users in order to check which version works best and get other insights.

- 2 Google AI Platform (formerly known as Google ML Engine) and Google AutoML merged in 2021 to form Google Vertex AI.
- 3 A REST (or RESTful) API is an API that uses standard HTTP verbs, such as GET, POST, PUT, and DELETE, and uses JSON inputs and outputs. The gRPC protocol is more complex but more efficient; data is exchanged using protocol buffers (see [Chapter 13](#)).
- 4 If you are not familiar with Docker, it allows you to easily download a set of applications packaged in a *Docker image* (including all their dependencies and usually some good default configuration) and then run them on your system using a *Docker engine*. When you run an image, the engine creates a *Docker container* that keeps the applications well isolated from your own system—but you can give it some limited access if you want. It is similar to a virtual machine, but much faster and lighter, as the container relies directly on the host's kernel. This means that the image does not need to include or run its own kernel.
- 5 There are also GPU images available, and other installation options. For more details, please check out the official [installation instructions](#).
- 6 To be fair, this can be mitigated by serializing the data first and encoding it to Base64 before creating the REST request. Moreover, REST requests can be compressed using gzip, which reduces the payload size significantly.
- 7 Also check out TensorFlow's [Graph Transform Tool](#) for modifying and optimizing computational graphs.
- 8 For example, a PWA must include icons of various sizes for different mobile devices, it must be served via HTTPS, it must include a manifest file containing metadata such as the name of the app and the background color.
- 9 Please check the TensorFlow docs for detailed and up-to-date installation instructions, as they change quite often.
- 10 As we saw in [Chapter 12](#), a kernel is an operation's implementation for a specific data type and device type. For example, there is a GPU kernel for the float32 tf.matmul() operation, but there is no GPU kernel for int32 tf.matmul(), only a CPU kernel.
- 11 You can also use `tf.debugging.set_log_device_placement(True)` to log all device placements.
- 12 This can be useful if you want to guarantee perfect reproducibility, as I explain in [this video](#), based on TF 1.
- 13 At the time of writing, it only prefetches the data to the CPU RAM, but use `tf.data.experimental.pre_fetch_to_device()` to make it prefetch the data and push it to the device of your choice so that the GPU does not waste time waiting for the data to be transferred.
- 14 If the two CNNs are identical, then it is called a *Siamese neural network*.
- 15 If you are interested in going further with model parallelism, check out [Mesh TensorFlow](#).
- 16 This name is slightly confusing because it sounds like some replicas are special, doing nothing. In reality, all replicas are equivalent: they all work hard to be among the fastest at each training step, and the losers vary at every step (unless some devices are really slower than others).

However, it does mean that if one or two servers crash, training will continue just fine.

- 17 Jianmin Chen et al., “Revisiting Distributed Synchronous SGD”, arXiv preprint arXiv:1604.00981 (2016).
- 18 Aaron Harlap et al., “PipeDream: Fast and Efficient Pipeline Parallel DNN Training”, arXiv preprint arXiv:1806.03377 (2018).
- 19 Paul Barham et al., “Pathways: Asynchronous Distributed Dataflow for ML”, arXiv preprint arXiv:2203.12533 (2022).
- 20 For more details on AllReduce algorithms, read [Yuichiro Ueno’s post](#) on the technologies behind deep learning and [Sylvain Jeaugey’s post](#) on massively scaling deep learning training with NCCL.

Appendix A. Machine Learning Project Checklist

This checklist can guide you through your machine learning projects. There are eight main steps:

1. Frame the problem and look at the big picture.
2. Get the data.
3. Explore the data to gain insights.
4. Prepare the data to better expose the underlying data patterns to machine learning algorithms.
5. Explore many different models and shortlist the best ones.
6. Fine-tune your models and combine them into a great solution.
7. Present your solution.
8. Launch, monitor, and maintain your system.

Obviously, you should feel free to adapt this checklist to your needs.

Frame the Problem and Look at the Big Picture

1. Define the objective in business terms.
2. How will your solution be used?
3. What are the current solutions/workarounds (if any)?
4. How should you frame this problem (supervised/unsupervised, online/offline, etc.)?
5. How should performance be measured?
6. Is the performance measure aligned with the business objective?
7. What would be the minimum performance needed to reach the business objective?
8. What are comparable problems? Can you reuse experience or tools?
9. Is human expertise available?
10. How would you solve the problem manually?
11. List the assumptions you (or others) have made so far.
12. Verify assumptions if possible.

Get the Data

Note: automate as much as possible so you can easily get fresh data.

1. List the data you need and how much you need.
2. Find and document where you can get that data.
3. Check how much space it will take.
4. Check legal obligations, and get authorization if necessary.
5. Get access authorizations.
6. Create a workspace (with enough storage space).
7. Get the data.
8. Convert the data to a format you can easily manipulate (without changing the data itself).
9. Ensure sensitive information is deleted or protected (e.g., anonymized).
10. Check the size and type of data (time series, sample, geographical, etc.).
11. Sample a test set, put it aside, and never look at it (no data snooping!).

Explore the Data

Note: try to get insights from a field expert for these steps.

1. Create a copy of the data for exploration (sampling it down to a manageable size if necessary).
2. Create a Jupyter notebook to keep a record of your data exploration.
3. Study each attribute and its characteristics:
 - Name
 - Type (categorical, int/float, bounded/unbounded, text, structured, etc.)
 - % of missing values
 - Noisiness and type of noise (stochastic, outliers, rounding errors, etc.)
 - Usefulness for the task
 - Type of distribution (Gaussian, uniform, logarithmic, etc.)
4. For supervised learning tasks, identify the target attribute(s).
5. Visualize the data.
6. Study the correlations between attributes.
7. Study how you would solve the problem manually.
8. Identify the promising transformations you may want to apply.
9. Identify extra data that would be useful (go back to “[Get the Data](#)”).
10. Document what you have learned.

Prepare the Data

Notes:

- Work on copies of the data (keep the original dataset intact).
- Write functions for all data transformations you apply, for five reasons:
 - So you can easily prepare the data the next time you get a fresh dataset
 - So you can apply these transformations in future projects
 - To clean and prepare the test set
 - To clean and prepare new data instances once your solution is live
 - To make it easy to treat your preparation choices as hyperparameters

1. Clean the data:

- Fix or remove outliers (optional).
- Fill in missing values (e.g., with zero, mean, median...) or drop their rows (or columns).

2. Perform feature selection (optional):

- Drop the attributes that provide no useful information for the task.

3. Perform feature engineering, where appropriate:

- Discretize continuous features.
- Decompose features (e.g., categorical, date/time, etc.).
- Add promising transformations of features (e.g., $\log(x)$, \sqrt{x} , x^2 , etc.).

- Aggregate features into promising new features.
4. Perform feature scaling:
- Standardize or normalize features.

Shortlist Promising Models

Notes:

- If the data is huge, you may want to sample smaller training sets so you can train many different models in a reasonable time (be aware that this penalizes complex models such as large neural nets or random forests).
 - Once again, try to automate these steps as much as possible.
1. Train many quick-and-dirty models from different categories (e.g., linear, naive Bayes, SVM, random forest, neural net, etc.) using standard parameters.
 2. Measure and compare their performance:
 - For each model, use N -fold cross-validation and compute the mean and standard deviation of the performance measure on the N folds.
 3. Analyze the most significant variables for each algorithm.
 4. Analyze the types of errors the models make:
 - What data would a human have used to avoid these errors?
 5. Perform a quick round of feature selection and engineering.
 6. Perform one or two more quick iterations of the five previous steps.
 7. Shortlist the top three to five most promising models, preferring models that make different types of errors.

Fine-Tune the System

Notes:

- You will want to use as much data as possible for this step, especially as you move toward the end of fine-tuning.
- As always, automate what you can.

1. Fine-tune the hyperparameters using cross-validation:

- Treat your data transformation choices as hyperparameters, especially when you are not sure about them (e.g., if you're not sure whether to replace missing values with zeros or with the median value, or to just drop the rows).
 - Unless there are very few hyperparameter values to explore, prefer random search over grid search. If training is very long, you may prefer a Bayesian optimization approach (e.g., using Gaussian process priors, as described by [Jasper Snoek et al.¹](#)).
2. Try ensemble methods. Combining your best models will often produce better performance than running them individually.
 3. Once you are confident about your final model, measure its performance on the test set to estimate the generalization error.

WARNING

Don't tweak your model after measuring the generalization error: you would just start overfitting the test set.

Present Your Solution

1. Document what you have done.
2. Create a nice presentation:
 - Make sure you highlight the big picture first.
3. Explain why your solution achieves the business objective.
4. Don't forget to present interesting points you noticed along the way:
 - Describe what worked and what did not.
 - List your assumptions and your system's limitations.
5. Ensure your key findings are communicated through beautiful visualizations or easy-to-remember statements (e.g., "the median income is the number-one predictor of housing prices").

Launch!

1. Get your solution ready for production (plug into production data inputs, write unit tests, etc.).
2. Write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops:
 - Beware of slow degradation: models tend to “rot” as data evolves.
 - Measuring performance may require a human pipeline (e.g., via a crowdsourcing service).
 - Also monitor your inputs' quality (e.g., a malfunctioning sensor sending random values, or another team's output becoming stale). This is particularly important for online learning systems.
3. Retrain your models on a regular basis on fresh data (automate as much as possible).

¹ Jasper Snoek et al., “Practical Bayesian Optimization of Machine Learning Algorithms”, *Proceedings of the 25th International Conference on Neural Information Processing Systems 2* (2012): 2951–2959.

Appendix B. Autodiff

This appendix explains how TensorFlow's autodifferentiation (autodiff) feature works, and how it compares to other solutions.

Suppose you define a function $f(x, y) = x^2y + y + 2$, and you need its partial derivatives $\partial f / \partial x$ and $\partial f / \partial y$, typically to perform gradient descent (or some other optimization algorithm). Your main options are manual differentiation, finite difference approximation, forward-mode autodiff, and reverse-mode autodiff. TensorFlow implements reverse-mode autodiff, but to understand it, it's useful to look at the other options first. So let's go through each of them, starting with manual differentiation.

Manual Differentiation

The first approach to compute derivatives is to pick up a pencil and a piece of paper and use your calculus knowledge to derive the appropriate equation. For the function $f(x, y)$ just defined, it is not too hard; you just need to use five rules:

- The derivative of a constant is 0.
- The derivative of λx is λ (where λ is a constant).
- The derivative of x^λ is $\lambda x^{\lambda-1}$, so the derivative of x^2 is $2x$.
- The derivative of a sum of functions is the sum of these functions' derivatives.
- The derivative of λ times a function is λ times its derivative.

From these rules, you can derive [Equation B-1](#).

Equation B-1. Partial derivatives of $f(x, y)$

$$\begin{aligned}\partial f / \partial x &= \partial(x^2 y) / \partial x + \partial y / \partial x + \partial(2) / \partial x = y \partial(x^2) / \partial x + 0 + 0 = 2x y \\ \partial f / \partial y &= \partial(x^2 y) / \partial y + \partial y / \partial y + \partial(2) / \partial y = x^2 + 1 + 0 = x^2 + 1\end{aligned}$$

This approach can become very tedious for more complex functions, and you run the risk of making mistakes. Fortunately, there are other options. Let's look at finite difference approximation now.

Finite Difference Approximation

Recall that the derivative $h'(x_0)$ of a function $h(x)$ at a point x_0 is the slope of the function at that point. More precisely, the derivative is defined as the limit of the slope of a straight line going through this point x_0 and another point x on the function, as x gets infinitely close to x_0 (see [Equation B-2](#)).

Equation B-2. Definition of the derivative of a function $h(x)$ at point x_0

$$h'(x_0) = \lim_{x \rightarrow x_0} \frac{h(x) - h(x_0)}{x - x_0} = \lim_{\epsilon \rightarrow 0} \frac{h(x_0 + \epsilon) - h(x_0)}{\epsilon}$$

So, if we wanted to calculate the partial derivative of $f(x, y)$ with regard to x at $x = 3$ and $y = 4$, we could compute $f(3 + \epsilon, 4) - f(3, 4)$ and divide the result by ϵ , using a very small value for ϵ . This type of numerical approximation of the derivative is called a *finite difference approximation*, and this specific equation is called *Newton's difference quotient*. That's exactly what the following code does:

```
def f(x, y):
    return x**2*y + y + 2

def derivative(f, x, y, x_eps, y_eps):
    return (f(x + x_eps, y + y_eps) - f(x, y)) / (x_eps + y_eps)

df_dx = derivative(f, 3, 4, 0.00001, 0)
df_dy = derivative(f, 3, 4, 0.00001)
```

Unfortunately, the result is imprecise (and it gets worse for more complicated functions). The correct results are respectively 24 and 10, but instead we get:

```
>>> df_dx
24.000039999805264
>>> df_dy
10.000000000331966
```

Notice that to compute both partial derivatives, we have to call $f()$ at least three times (we called it four times in the preceding code, but it could be optimized). If there were 1,000 parameters, we would need to call $f()$ at least

1,001 times. When you are dealing with large neural networks, this makes finite difference approximation way too inefficient.

However, this method is so simple to implement that it is a great tool to check that the other methods are implemented correctly. For example, if it disagrees with your manually derived function, then your function probably contains a mistake.

So far, we have considered two ways to compute gradients: using manual differentiation and using finite difference approximation. Unfortunately, both are fatally flawed for training a large-scale neural network. So let's turn to autodiff, starting with forward mode.

Forward-Mode Autodiff

Figure B-1 shows how forward-mode autodiff works on an even simpler function, $g(x, y) = 5 + xy$. The graph for that function is represented on the left. After forward-mode autodiff, we get the graph on the right, which represents the partial derivative $\partial g/\partial x = 0 + (0 \times x + y \times 1) = y$ (we could similarly obtain the partial derivative with regard to y).

The algorithm will go through the computation graph from the inputs to the outputs (hence the name “forward mode”). It starts by getting the partial derivatives of the leaf nodes. The constant node (5) returns the constant 0, since the derivative of a constant is always 0. The variable x returns the constant 1 since $\partial x/\partial x = 1$, and the variable y returns the constant 0 since $\partial y/\partial x = 0$ (if we were looking for the partial derivative with regard to y , it would be the reverse).

Now we have all we need to move up the graph to the multiplication node in function g . Calculus tells us that the derivative of the product of two functions u and v is $\partial(u \times v)/\partial x = \partial v/\partial x \times u + v \times \partial u/\partial x$. We can therefore construct a large part of the graph on the right, representing $0 \times x + y \times 1$.

Finally, we can go up to the addition node in function g . As mentioned, the derivative of a sum of functions is the sum of these functions’ derivatives, so we just need to create an addition node and connect it to the parts of the graph we have already computed. We get the correct partial derivative: $\partial g/\partial x = 0 + (0 \times x + y \times 1)$.

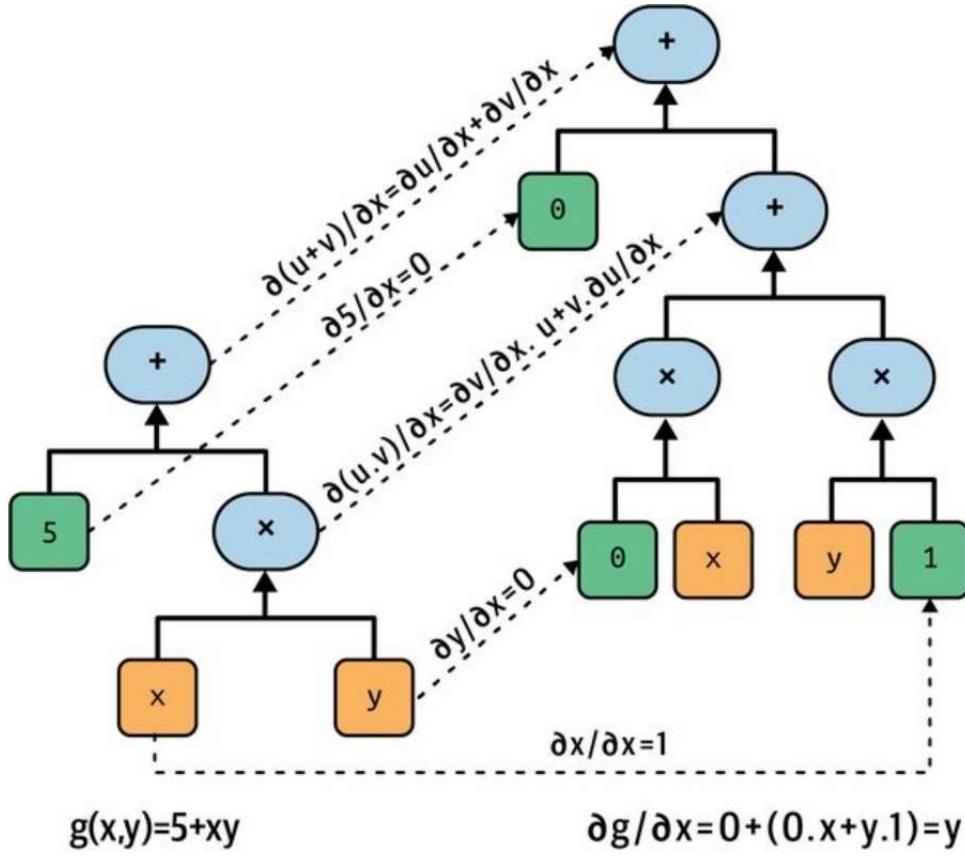


Figure B-1. Forward-mode autodiff

However, this equation can be simplified (a lot). By applying a few pruning steps to the computation graph to get rid of all the unnecessary operations, we get a much smaller graph with just one node: $\frac{\partial g}{\partial x} = y$. In this case simplification is fairly easy, but for a more complex function forward-mode autodiff can produce a huge graph that may be tough to simplify and lead to suboptimal performance.

Note that we started with a computation graph, and forward-mode autodiff produced another computation graph. This is called *symbolic differentiation*, and it has two nice features: first, once the computation graph of the derivative has been produced, we can use it as many times as we want to compute the derivatives of the given function for any value of x and y ;

second, we can run forward-mode autodiff again on the resulting graph to get second-order derivatives if we ever need to (i.e., derivatives of derivatives). We could even compute third-order derivatives, and so on.

But it is also possible to run forward-mode autodiff without constructing a graph (i.e., numerically, not symbolically), just by computing intermediate results on the fly. One way to do this is to use *dual numbers*, which are weird but fascinating numbers of the form $a + b\epsilon$, where a and b are real numbers and ϵ is an infinitesimal number such that $\epsilon^2 = 0$ (but $\epsilon \neq 0$). You can think of the dual number $42 + 24\epsilon$ as something akin to $42.0000\cdots000024$ with an infinite number of 0s (but of course this is simplified just to give you some idea of what dual numbers are). A dual number is represented in memory as a pair of floats. For example, $42 + 24\epsilon$ is represented by the pair $(42.0, 24.0)$.

Dual numbers can be added, multiplied, and so on, as shown in [Equation B-3](#).

Equation B-3. A few operations with dual numbers

$$\begin{aligned}\lambda(a + b\epsilon) &= \lambda a + \lambda b\epsilon (a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon (a + b\epsilon) \\ \epsilon \times (c + d\epsilon) &= a c + (a d + b c)\epsilon + (b d)\epsilon^2 = a c + (a d + b c)\epsilon\end{aligned}$$

Most importantly, it can be shown that $h(a + b\epsilon) = h(a) + b \times h'(a)\epsilon$, so computing $h(a + \epsilon)$ gives you both $h(a)$ and the derivative $h'(a)$ in just one shot. [Figure B-2](#) shows that the partial derivative of $f(x, y)$ with regard to x at $x = 3$ and $y = 4$ (which I will write $\partial f / \partial x(3, 4)$) can be computed using dual numbers. All we need to do is compute $f(3 + \epsilon, 4)$; this will output a dual number whose first component is equal to $f(3, 4)$ and whose second component is equal to $\partial f / \partial x(3, 4)$.

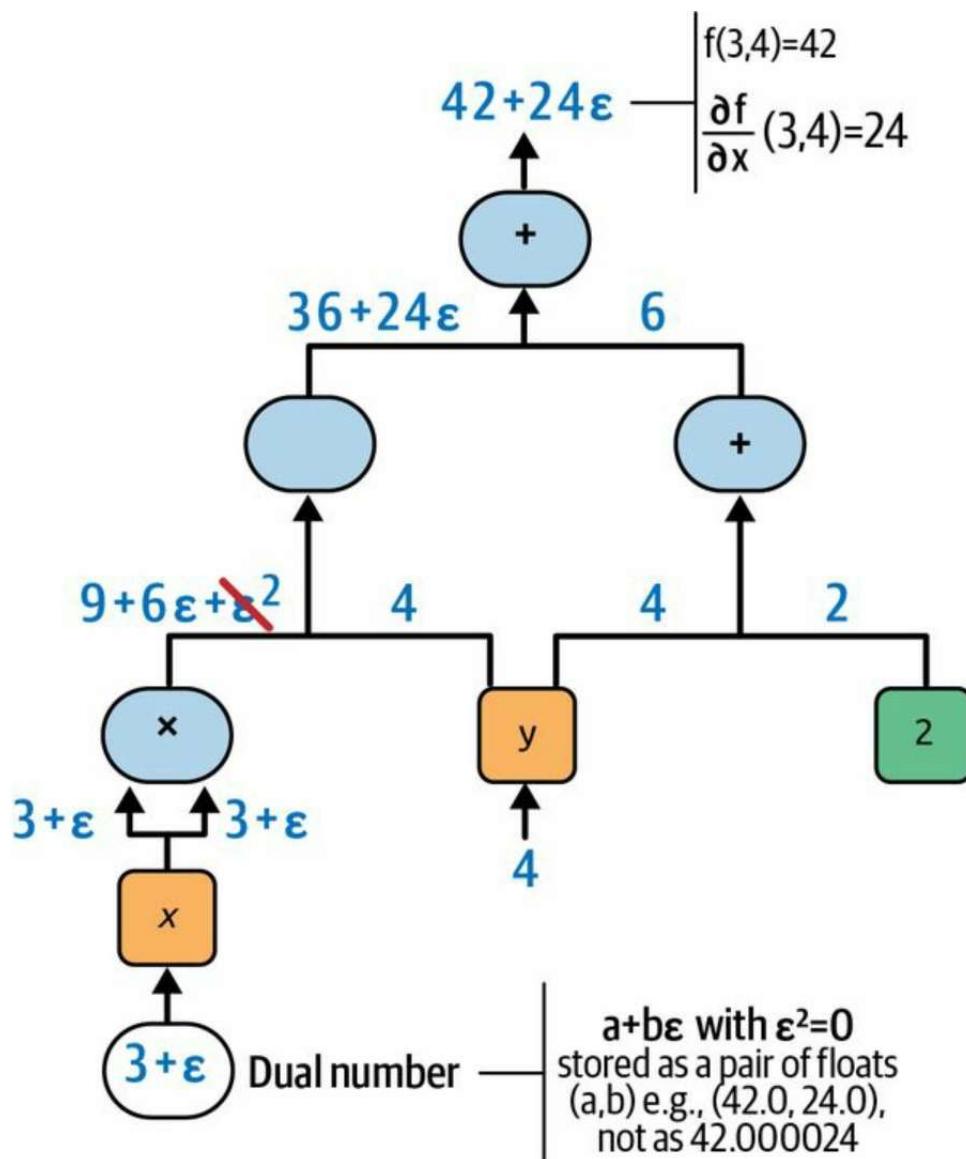


Figure B-2. Forward-mode autodiff using dual numbers

To compute $\frac{\partial f}{\partial y}(3, 4)$ we would have to go through the graph again, but this time with $x = 3$ and $y = 4 + \epsilon$.

So, forward-mode autodiff is much more accurate than finite difference

approximation, but it suffers from the same major flaw, at least when there are many inputs and few outputs (as is the case when dealing with neural networks): if there were 1,000 parameters, it would require 1,000 passes through the graph to compute all the partial derivatives. This is where reverse-mode autodiff shines: it can compute all of them in just two passes through the graph. Let's see how.

Reverse-Mode Autodiff

Reverse-mode autodiff is the solution implemented by TensorFlow. It first goes through the graph in the forward direction (i.e., from the inputs to the output) to compute the value of each node. Then it does a second pass, this time in the reverse direction (i.e., from the output to the inputs), to compute all the partial derivatives. The name “reverse mode” comes from this second pass through the graph, where gradients flow in the reverse direction.

Figure B-3 represents the second pass. During the first pass, all the node values were computed, starting from $x = 3$ and $y = 4$. You can see those values at the bottom right of each node (e.g., $x \times x = 9$). The nodes are labeled n_1 to n_7 for clarity. The output node is n_7 : $f(3, 4) = n_7 = 42$.

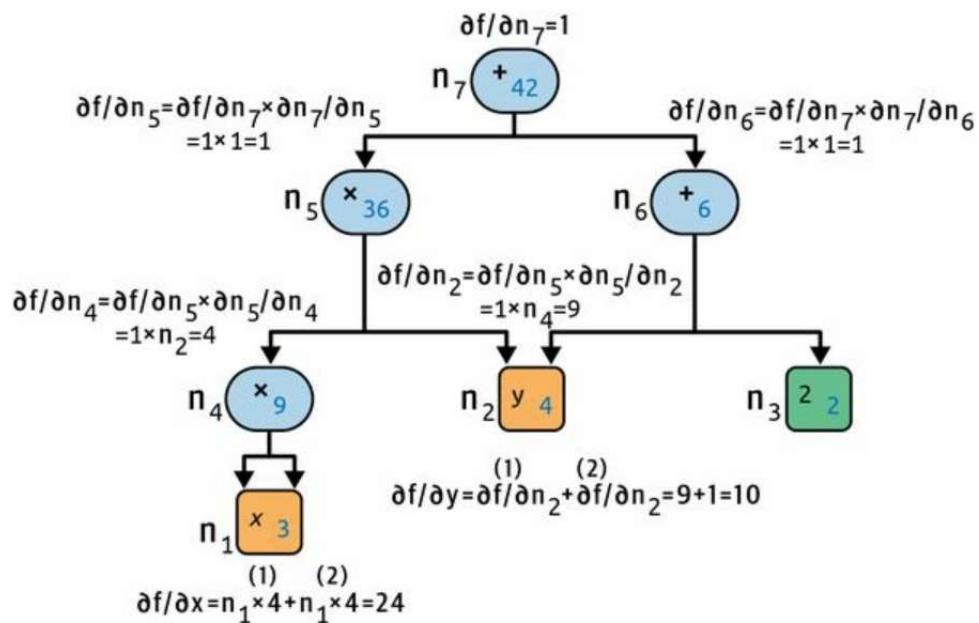


Figure B-3. Reverse-mode autodiff

The idea is to gradually go down the graph, computing the partial derivative of $f(x, y)$ with regard to each consecutive node, until we reach the variable nodes. For this, reverse-mode autodiff relies heavily on the *chain rule*, shown

in [Equation B-4](#).

Equation B-4. Chain rule

$$\partial f / \partial x = \partial f / \partial n_i \times \partial n_i / \partial x$$

Since n_7 is the output node, $f = n_7$ so $\partial f / \partial n_7 = 1$.

Let's continue down the graph to n_5 : how much does f vary when n_5 varies? The answer is $\partial f / \partial n_5 = \partial f / \partial n_7 \times \partial n_7 / \partial n_5$. We already know that $\partial f / \partial n_7 = 1$, so all we need is $\partial n_7 / \partial n_5$. Since n_7 simply performs the sum $n_5 + n_6$, we find that $\partial n_7 / \partial n_5 = 1$, so $\partial f / \partial n_5 = 1 \times 1 = 1$.

Now we can proceed to node n_4 : how much does f vary when n_4 varies? The answer is $\partial f / \partial n_4 = \partial f / \partial n_5 \times \partial n_5 / \partial n_4$. Since $n_5 = n_4 \times n_2$, we find that $\partial n_5 / \partial n_4 = n_2$, so $\partial f / \partial n_4 = 1 \times n_2 = 4$.

The process continues until we reach the bottom of the graph. At that point we will have calculated all the partial derivatives of $f(x, y)$ at the point $x = 3$ and $y = 4$. In this example, we find $\partial f / \partial x = 24$ and $\partial f / \partial y = 10$. Sounds about right!

Reverse-mode autodiff is a very powerful and accurate technique, especially when there are many inputs and few outputs, since it requires only one forward pass plus one reverse pass per output to compute all the partial derivatives for all outputs with regard to all the inputs. When training neural networks, we generally want to minimize the loss, so there is a single output (the loss), and hence only two passes through the graph are needed to compute the gradients. Reverse-mode autodiff can also handle functions that are not entirely differentiable, as long as you ask it to compute the partial derivatives at points that are differentiable.

In [Figure B-3](#), the numerical results are computed on the fly, at each node. However, that's not exactly what TensorFlow does: instead, it creates a new computation graph. In other words, it implements *symbolic* reverse-mode autodiff. This way, the computation graph to compute the gradients of the loss with regard to all the parameters in the neural network only needs to be generated once, and then it can be executed over and over again, whenever the optimizer needs to compute the gradients. Moreover, this makes it

possible to compute higher-order derivatives if needed.

TIP

If you ever want to implement a new type of low-level TensorFlow operation in C++, and you want to make it compatible with autodiff, then you will need to provide a function that returns the partial derivatives of the function's outputs with regard to its inputs. For example, suppose you implement a function that computes the square of its input: $f(x) = x^2$. In that case you would need to provide the corresponding derivative function: $f'(x) = 2x$.

Appendix C. Special Data Structures

In this appendix we will take a very quick look at the data structures supported by TensorFlow, beyond regular float or integer tensors. This includes strings, ragged tensors, sparse tensors, tensor arrays, sets, and queues.

Strings

Tensors can hold byte strings, which is useful in particular for natural language processing (see [Chapter 16](#)):

```
>>> tf.constant(b"hello world")
<tf.Tensor: shape=(), dtype=string, numpy=b'hello world'>
```

If you try to build a tensor with a Unicode string, TensorFlow automatically encodes it to UTF-8:

```
>>> tf.constant("café")
<tf.Tensor: shape=(), dtype=string, numpy=b'caf\xc3\xa9'>
```

It is also possible to create tensors representing Unicode strings. Just create an array of 32-bit integers, each representing a single Unicode code point: ¹

```
>>> u = tf.constant([ord(c) for c in "café"])
>>> u
<tf.Tensor: shape=(4,), [...], numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

NOTE

In tensors of type `tf.string`, the string length is not part of the tensor's shape. In other words, strings are considered as atomic values. However, in a Unicode string tensor (i.e., an `int32` tensor), the length of the string *is* part of the tensor's shape.

The `tf.strings` package contains several functions to manipulate string tensors, such as `length()` to count the number of bytes in a byte string (or the number of code points if you set `unit="UTF8_CHAR"`), `unicode_encode()` to convert a Unicode string tensor (i.e., `int32` tensor) to a byte string tensor, and `unicode_decode()` to do the reverse:

```
>>> b = tf.strings.unicode_encode(u, "UTF-8")
>>> b
```

```
<tf.Tensor: shape=(), dtype=string, numpy=b'caf\xc3\xaa9'>
>>> tf.strings.length(b, unit="UTF8_CHAR")
<tf.Tensor: shape=(), dtype=int32, numpy=4>
>>> tf.strings.unicode_decode(b, "UTF-8")
<tf.Tensor: shape=(4,), [...], numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

You can also manipulate tensors containing multiple strings:

```
>>> p = tf.constant(["Café", "Coffee", "caffè", "咖啡"])
>>> tf.strings.length(p, unit="UTF8_CHAR")
<tf.Tensor: shape=(4,), dtype=int32, numpy=array([4, 6, 5, 2], dtype=int32)>
>>> r = tf.strings.unicode_decode(p, "UTF8")
>>> r
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101], [99, 97,
102, 102, 232], [21654, 21857]]>
```

Notice that the decoded strings are stored in a RaggedTensor. What is that?

Ragged Tensors

A *ragged tensor* is a special kind of tensor that represents a list of arrays of different sizes. More generally, it is a tensor with one or more *ragged dimensions*, meaning dimensions whose slices may have different lengths. In the ragged tensor `r`, the second dimension is a ragged dimension. In all ragged tensors, the first dimension is always a regular dimension (also called a *uniform dimension*).

All the elements of the ragged tensor `r` are regular tensors. For example, let's look at the second element of the ragged tensor:

```
>>> r[1]
<tf.Tensor: [...], numpy=array([ 67, 111, 102, 102, 101, 101], dtype=int32)>
```

The `tf.ragged` package contains several functions to create and manipulate ragged tensors. Let's create a second ragged tensor using `tf.ragged.constant()` and concatenate it with the first ragged tensor, along axis 0:

```
>>> r2 = tf.ragged.constant([[65, 66], [], [67]])
>>> tf.concat([r, r2], axis=0)
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101], [99, 97,
102, 102, 232], [21654, 21857], [65, 66], [], [67]]>
```

The result is not too surprising: the tensors in `r2` were appended after the tensors in `r` along axis 0. But what if we concatenate `r` and another ragged tensor along axis 1?

```
>>> r3 = tf.ragged.constant([[68, 69, 70], [71], [], [72, 73]])
>>> print(tf.concat([r, r3], axis=1))
<tf.RaggedTensor [[67, 97, 102, 233, 68, 69, 70], [67, 111, 102, 102, 101, 101,
71], [99, 97, 102, 102, 232], [21654, 21857, 72, 73]]>
```

This time, notice that the i^{th} tensor in `r` and the i^{th} tensor in `r3` were concatenated. Now that's more unusual, since all of these tensors can have different lengths.

If you call the `to_tensor()` method, the ragged tensor gets converted to a regular tensor, padding shorter tensors with zeros to get tensors of equal lengths (you can change the default value by setting the `default_value` argument):

```
>>> r.to_tensor()
<tf.Tensor: shape=(4, 6), dtype=int32, numpy=
array([[ 67,  97, 102, 233,  0,  0],
       [ 67, 111, 102, 102, 101, 101],
       [ 99,  97, 102, 102, 232,  0],
       [21654, 21857,  0,  0,  0,  0]], dtype=int32)>
```

Many TF operations support ragged tensors. For the full list, see the documentation of the `tf.RaggedTensor` class.

Sparse Tensors

TensorFlow can also efficiently represent *sparse tensors* (i.e., tensors containing mostly zeros). Just create a `tf.SparseTensor`, specifying the indices and values of the nonzero elements and the tensor's shape. The indices must be listed in “reading order” (from left to right, and top to bottom). If you are unsure, just use `tf.sparse.reorder()`. You can convert a sparse tensor to a dense tensor (i.e., a regular tensor) using `tf.sparse.to_dense()`:

```
>>> s = tf.SparseTensor(indices=[[0, 1], [1, 0], [2, 3]],
...                      values=[1., 2., 3.],
...                      dense_shape=[3, 4])
...
>>> tf.sparse.to_dense(s)
<tf.Tensor: shape=(3, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [2., 0., 0., 0.],
       [0., 0., 0., 3.]], dtype=float32)>
```

Note that sparse tensors do not support as many operations as dense tensors. For example, you can multiply a sparse tensor by any scalar value, and you get a new sparse tensor, but you cannot add a scalar value to a sparse tensor, as this would not return a sparse tensor:

```
>>> s * 42.0
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x7f84a6749f10>
>>> s + 42.0
[...] TypeError: unsupported operand type(s) for +: 'SparseTensor' and 'float'
```

Tensor Arrays

A `tf.TensorArray` represents a list of tensors. This can be handy in dynamic models containing loops, to accumulate results and later compute some statistics. You can read or write tensors at any location in the array:

```
array = tf.TensorArray(dtype=tf.float32, size=3)
array = array.write(0, tf.constant([1., 2.]))
array = array.write(1, tf.constant([3., 10.]))
array = array.write(2, tf.constant([5., 7.]))
tensor1 = array.read(1) # => returns (and zeros out!) tf.constant([3., 10.])
```

By default, reading an item also replaces it with a tensor of the same shape but full of zeros. You can set `clear_after_read` to `False` if you don't want this.

WARNING

When you write to the array, you must assign the output back to the array, as shown in this code example. If you don't, although your code will work fine in eager mode, it will break in graph mode (these modes are discussed in [Chapter 12](#)).

By default, a `TensorArray` has a fixed size that is set upon creation. Alternatively, you can set `size=0` and `dynamic_size=True` to let the array grow automatically when needed. However, this will hinder performance, so if you know the size in advance, it's better to use a fixed-size array. You must also specify the `dtype`, and all elements must have the same shape as the first one written to the array.

You can stack all the items into a regular tensor by calling the `stack()` method:

```
>>> array.stack()
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[1., 2.],
       [0., 0.],
       [5., 7.]], dtype=float32)>
```

Sets

TensorFlow supports sets of integers or strings (but not floats). It represents sets using regular tensors. For example, the set {1, 5, 9} is just represented as the tensor [[1, 5, 9]]. Note that the tensor must have at least two dimensions, and the sets must be in the last dimension. For example, [[1, 5, 9], [2, 5, 11]] is a tensor holding two independent sets: {1, 5, 9} and {2, 5, 11}.

The `tf.sets` package contains several functions to manipulate sets. For example, let's create two sets and compute their union (the result is a sparse tensor, so we call `to_dense()` to display it):

```
>>> a = tf.constant([[1, 5, 9]])
>>> b = tf.constant([[5, 6, 9, 11]])
>>> u = tf.sets.union(a, b)
>>> u
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x132b60d30>
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11]], dtype=int32)>
```

You can also compute the union of multiple pairs of sets simultaneously. If some sets are shorter than others, you must pad them with a padding value, such as 0:

```
>>> a = tf.constant([[1, 5, 9], [10, 0, 0]])
>>> b = tf.constant([[5, 6, 9, 11], [13, 0, 0, 0]])
>>> u = tf.sets.union(a, b)
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],
   [ 0, 10, 13,  0,  0]], dtype=int32)>
```

If you prefer to use a different padding value, such as `-1`, then you must set `default_value=-1` (or your preferred value) when calling `to_dense()`.

WARNING

The default `default_value` is 0, so when dealing with string sets, you must set this

parameter (e.g., to an empty string).

Other functions available in `tf.sets` include `difference()`, `intersection()`, and `size()`, which are self-explanatory. If you want to check whether or not a set contains some given values, you can compute the intersection of that set and the values. If you want to add some values to a set, you can compute the union of the set and the values.

Queues

A queue is a data structure to which you can push data records, and later pull them out. TensorFlow implements several types of queues in the `tf.queue` package. They used to be very important when implementing efficient data loading and preprocessing pipelines, but the `tf.data` API has essentially rendered them useless (except perhaps in some rare cases) because it is much simpler to use and provides all the tools you need to build efficient pipelines. For the sake of completeness, though, let's take a quick look at them.

The simplest kind of queue is the first-in, first-out (FIFO) queue. To build it, you need to specify the maximum number of records it can contain.

Moreover, each record is a tuple of tensors, so you must specify the type of each tensor, and optionally their shapes. For example, the following code example creates a FIFO queue with a maximum of three records, each containing a tuple with a 32-bit integer and a string. Then it pushes two records to it, looks at the size (which is 2 at this point), and pulls a record out:

```
>>> q = tf.queue.FIFOQueue(3, [tf.int32, tf.string], shapes=[(), ()])
>>> q.enqueue([10, b"windy"])
>>> q.enqueue([15, b"sunny"])
>>> q.size()
<tf.Tensor: shape=(), dtype=int32, numpy=2>
>>> q.dequeue()
[<tf.Tensor: shape=(), dtype=int32, numpy=10>,
 <tf.Tensor: shape=(), dtype=string, numpy=b'windy'>]
```

It is also possible to enqueue and dequeue multiple records at once using `enqueue_many()` and `dequeue_many()` (to use `dequeue_many()`, you must specify the `shapes` argument when you create the queue, as we did previously):

```
>>> q.enqueue_many([[13, 16], [b'cloudy', b'raining']])
>>> q.dequeue_many(3)
[<tf.Tensor: [...], numpy=array([15, 13, 16], dtype=int32)>,
 <tf.Tensor: [...], numpy=array([b'sunny', b'cloudy', b'raining'], dtype=object)>]
```

Other queue types include:

PaddingFIFOQueue

Same as FIFOQueue, but its `dequeue_many()` method supports dequeuing multiple records of different shapes. It automatically pads the shortest records to ensure all the records in the batch have the same shape.

PriorityQueue

A queue that dequeues records in a prioritized order. The priority must be a 64-bit integer included as the first element of each record. Surprisingly, records with a lower priority will be dequeued first. Records with the same priority will be dequeued in FIFO order.

RandomShuffleQueue

A queue whose records are dequeued in random order. This was useful to implement a shuffle buffer before `tf.data` existed.

If a queue is already full and you try to enqueue another record, the `enqueue*`() method will freeze until a record is dequeued by another thread. Similarly, if a queue is empty and you try to dequeue a record, the `dequeue*()` method will freeze until records are pushed to the queue by another thread.

¹ If you are not familiar with Unicode code points, please check out <https://homl.info/unicode>.

Appendix D. TensorFlow Graphs

In this appendix, we will explore the graphs generated by TF functions (see [Chapter 12](#)).

TF Functions and Concrete Functions

TF functions are polymorphic, meaning they support inputs of different types (and shapes). For example, consider the following `tf_cube()` function:

```
@tf.function
def tf_cube(x):
    return x ** 3
```

Every time you call a TF function with a new combination of input types or shapes, it generates a new *concrete function*, with its own graph specialized for this particular combination. Such a combination of argument types and shapes is called an *input signature*. If you call the TF function with an input signature it has already seen before, it will reuse the concrete function it generated earlier. For example, if you call `tf_cube(tf.constant(3.0))`, the TF function will reuse the same concrete function it used for `tf_cube(tf.constant(2.0))` (for float32 scalar tensors). But it will generate a new concrete function if you call `tf_cube(tf.constant([2.0]))` or `tf_cube(tf.constant([3.0]))` (for float32 tensors of shape [1]), and yet another for `tf_cube(tf.constant([[1.0, 2.0], [3.0, 4.0]]))` (for float32 tensors of shape [2, 2]). You can get the concrete function for a particular combination of inputs by calling the TF function's `get_concrete_function()` method. It can then be called like a regular function, but it will only support one input signature (in this example, float32 scalar tensors):

```
>>> concrete_function = tf_cube.get_concrete_function(tf.constant(2.0))
>>> concrete_function
<ConcreteFunction tf_cube(x) at 0x7F84411F4250>
>>> concrete_function(tf.constant(2.0))
<tf.Tensor: shape=(), dtype=float32, numpy=8.0>
```

Figure D-1 shows the `tf_cube()` TF function, after we called `tf_cube(2)` and `tf_cube(tf.constant(2.0))`: two concrete functions were generated, one for each signature, each with its own optimized *function graph* (`FuncGraph`) and its own *function definition* (`FunctionDef`). A function definition points to the

parts of the graph that correspond to the function's inputs and outputs. In each FuncGraph, the nodes (ovals) represent operations (e.g., power, constants, or placeholders for arguments like x), while the edges (the solid arrows) represent the tensors that will flow through the graph. The concrete function on the left is specialized for $x=2$, so TensorFlow managed to simplify it to just output 8 all the time (note that the function definition does not even have an input). The concrete function on the right is specialized for float32 scalar tensors, and it could not be simplified. If we call `tf_cube(tf.constant(5.0))`, the second concrete function will be called, the placeholder operation for x will output 5.0, then the power operation will compute $5.0^{**} 3$, so the output will be 125.0.

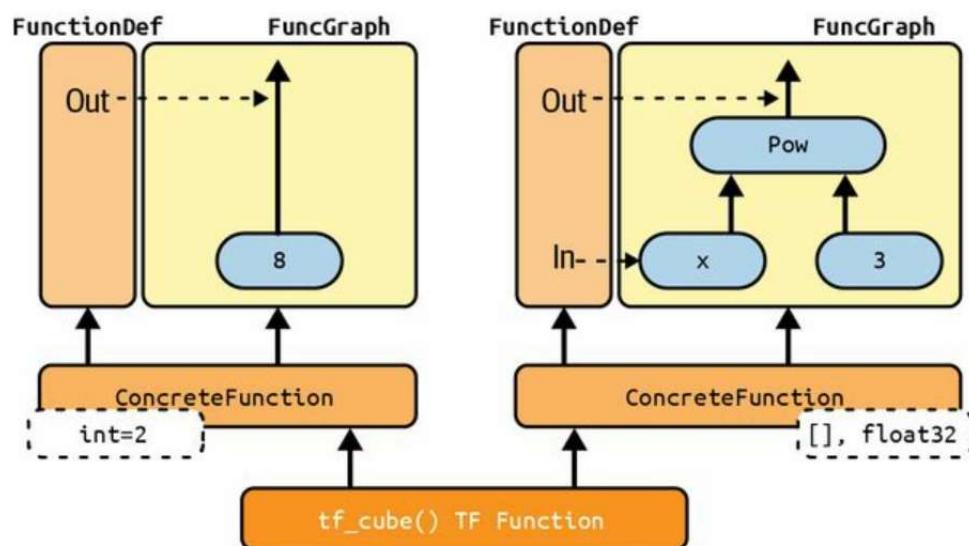


Figure D-1. The `tf_cube()` TF function, with its `ConcreteFunctions` and their `FuncGraphs`

The tensors in these graphs are *symbolic tensors*, meaning they don't have an actual value, just a data type, a shape, and a name. They represent the future tensors that will flow through the graph once an actual value is fed to the placeholder x and the graph is executed. Symbolic tensors make it possible to specify ahead of time how to connect operations, and they also allow TensorFlow to recursively infer the data types and shapes of all tensors, given the data types and shapes of their inputs.

Now let's continue to peek under the hood, and see how to access function definitions and function graphs and how to explore a graph's operations and tensors.

Exploring Function Definitions and Graphs

You can access a concrete function’s computation graph using the graph attribute, and get the list of its operations by calling the graph’s get_operations() method:

```
>>> concrete_function.graph
<tensorflow.python.framework.func_graph.FuncGraph at 0x7f84411f4790>
>>> ops = concrete_function.graph.get_operations()
>>> ops
[<tf.Operation 'x' type=Placeholder>,
 <tf.Operation 'pow/y' type=Const>,
 <tf.Operation 'pow' type=Pow>,
 <tf.Operation 'Identity' type=Identity>]
```

In this example, the first operation represents the input argument x (it is called a *placeholder*), the second “operation” represents the constant 3, the third operation represents the power operation (**), and the final operation represents the output of this function (it is an identity operation, meaning it will do nothing more than copy the output of the power operation ¹). Each operation has a list of input and output tensors that you can easily access using the operation’s inputs and outputs attributes. For example, let’s get the list of inputs and outputs of the power operation:

```
>>> pow_op = ops[2]
>>> list(pow_op.inputs)
[<tf.Tensor 'x:0' shape=() dtype=float32>,
 <tf.Tensor 'pow/y:0' shape=() dtype=float32>]
>>> pow_op.outputs
[<tf.Tensor 'pow:0' shape=() dtype=float32>]
```

This computation graph is represented in [Figure D-2](#).

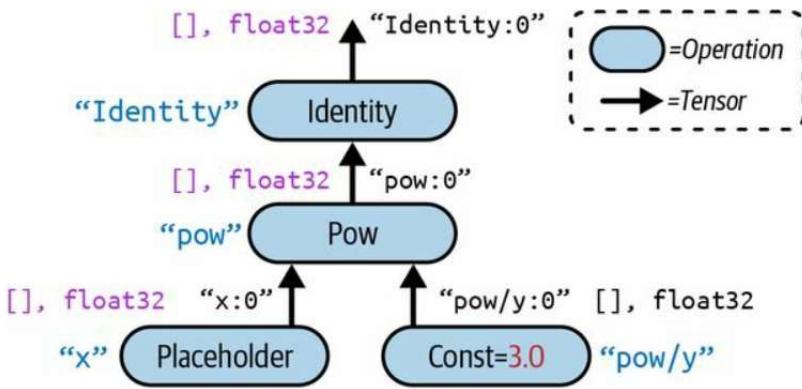


Figure D-2. Example of a computation graph

Note that each operation has a name. It defaults to the name of the operation (e.g., "pow"), but you can define it manually when calling the operation (e.g., `tf.pow(x, 3, name="other_name")`). If a name already exists, TensorFlow automatically adds a unique index (e.g., "pow_1", "pow_2", etc.). Each tensor also has a unique name: it is always the name of the operation that outputs this tensor, plus :0 if it is the operation's first output, or :1 if it is the second output, and so on. You can fetch an operation or a tensor by name using the graph's `get_operation_by_name()` or `get_tensor_by_name()` methods:

```

>>> concrete_function.graph.get_operation_by_name('x')
<tf.Operation 'x' type=Placeholder>
>>> concrete_function.graph.get_tensor_by_name('Identity:0')
<tf.Tensor 'Identity:0' shape=() dtype=float32>

```

The concrete function also contains the function definition (represented as a protocol buffer ²), which includes the function's signature. This signature allows the concrete function to know which placeholders to feed with the input values, and which tensors to return:

```

>>> concrete_function.function_def.signature
name: "__inference_tf_cube_3515903"
input_arg {
    name: "x"
    type: DT_FLOAT
}
output_arg {
    name: "identity"
}

```

```
    type: DT_FLOAT  
}
```

Now let's look more closely at tracing.

A Closer Look at Tracing

Let's tweak the `tf_cube()` function to print its input:

```
@tf.function
def tf_cube(x):
    print(f"x = {x}")
    return x ** 3
```

Now let's call it:

```
>>> result = tf_cube(tf.constant(2.0))
x = Tensor("x:0", shape=(), dtype=float32)
>>> result
<tf.Tensor: shape=(), dtype=float32, numpy=8.0>
```

The result looks good, but look at what was printed: `x` is a symbolic tensor! It has a shape and a data type, but no value. Plus it has a name ("`x:0`"). This is because the `print()` function is not a TensorFlow operation, so it will only run when the Python function is traced, which happens in graph mode, with arguments replaced with symbolic tensors (same type and shape, but no value). Since the `print()` function was not captured into the graph, the next times we call `tf_cube()` with float32 scalar tensors, nothing is printed:

```
>>> result = tf_cube(tf.constant(3.0))
>>> result = tf_cube(tf.constant(4.0))
```

But if we call `tf_cube()` with a tensor of a different type or shape, or with a new Python value, the function will be traced again, so the `print()` function will be called:

```
>>> result = tf_cube(2) # new Python value: trace!
x = 2
>>> result = tf_cube(3) # new Python value: trace!
x = 3
>>> result = tf_cube(tf.constant([[1., 2.]])) # new shape: trace!
x = Tensor("x:0", shape=(1, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[3., 4.], [5., 6.]])) # new shape: trace!
```

```
x = Tensor("x:0", shape=(None, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[7., 8.], [9., 10.]])) # same shape: no trace
```

WARNING

If your function has Python side effects (e.g., it saves some logs to disk), be aware that this code will only run when the function is traced (i.e., every time the TF function is called with a new input signature). It's best to assume that the function may be traced (or not) any time the TF function is called.

In some cases, you may want to restrict a TF function to a specific input signature. For example, suppose you know that you will only ever call a TF function with batches of 28×28 -pixel images, but the batches will have very different sizes. You may not want TensorFlow to generate a different concrete function for each batch size, or count on it to figure out on its own when to use None. In this case, you can specify the input signature like this:

```
@tf.function(input_signature=[tf.TensorSpec([None, 28, 28], tf.float32)])
def shrink(images):
    return images[:, ::2, ::2] # drop half the rows and columns
```

This TF function will accept any float32 tensor of shape $[*, 28, 28]$, and it will reuse the same concrete function every time:

```
img_batch_1 = tf.random.uniform(shape=[100, 28, 28])
img_batch_2 = tf.random.uniform(shape=[50, 28, 28])
preprocessed_images = shrink(img_batch_1) # works fine, traces the function
preprocessed_images = shrink(img_batch_2) # works fine, same concrete function
```

However, if you try to call this TF function with a Python value, or a tensor of an unexpected data type or shape, you will get an exception:

```
img_batch_3 = tf.random.uniform(shape=[2, 2, 2])
preprocessed_images = shrink(img_batch_3) # ValueError! Incompatible inputs
```

Using AutoGraph to Capture Control Flow

If your function contains a simple for loop, what do you expect will happen? For example, let's write a function that will add 10 to its input, by just adding 1 10 times:

```
@tf.function
def add_10(x):
    for i in range(10):
        x += 1
    return x
```

It works fine, but when we look at its graph, we find that it does not contain a loop: it just contains 10 addition operations!

```
>>> add_10(tf.constant(0))
<tf.Tensor: shape=(), dtype=int32, numpy=15>
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'add' type=AddV2>, [...],
 <tf.Operation 'add_1' type=AddV2>, [...],
 <tf.Operation 'add_2' type=AddV2>, [...],
 [...]
 <tf.Operation 'add_9' type=AddV2>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

This actually makes sense: when the function got traced, the loop ran 10 times, so the `x += 1` operation was run 10 times, and since it was in graph mode, it recorded this operation 10 times in the graph. You can think of this for loop as a “static” loop that gets unrolled when the graph is created.

If you want the graph to contain a “dynamic” loop instead (i.e., one that runs when the graph is executed), you can create one manually using the `tf.while_loop()` operation, but it is not very intuitive (see the “Using AutoGraph to Capture Control Flow” section of the Chapter 12 notebook for an example). Instead, it is much simpler to use TensorFlow’s *AutoGraph* feature, discussed in [Chapter 12](#). AutoGraph is actually activated by default (if you ever need to turn it off, you can pass `autograph=False` to `tf.function()`).

So if it is on, why didn't it capture the for loop in the add_10() function? It only captures for loops that iterate over tensors of tf.data.Dataset objects, so you should use tf.range(), not range(). This is to give you the choice:

- If you use range(), the for loop will be static, meaning it will only be executed when the function is traced. The loop will be “unrolled” into a set of operations for each iteration, as we saw.
- If you use tf.range(), the loop will be dynamic, meaning that it will be included in the graph itself (but it will not run during tracing).

Let's look at the graph that gets generated if we just replace range() with tf.range() in the add_10() function:

```
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()  
[<tf.Operation 'x' type=Placeholder>, [...],  
<tf.Operation 'while' type=StatelessWhile>, [...]]
```

As you can see, the graph now contains a While loop operation, as if we had called the tf.while_loop() function.

Handling Variables and Other Resources in TF Functions

In TensorFlow, variables and other stateful objects, such as queues or datasets, are called *resources*. TF functions treat them with special care: any operation that reads or updates a resource is considered stateful, and TF functions ensure that stateful operations are executed in the order they appear (as opposed to stateless operations, which may be run in parallel, so their order of execution is not guaranteed). Moreover, when you pass a resource as an argument to a TF function, it gets passed by reference, so the function may modify it. For example:

```
counter = tf.Variable(0)

@tf.function
def increment(counter, c=1):
    return counter.assign_add(c)

increment(counter) # counter is now equal to 1
increment(counter) # counter is now equal to 2
```

If you peek at the function definition, the first argument is marked as a resource:

```
>>> function_def = increment.get_concrete_function(counter).function_def
>>> function_def.signature.input_arg[0]
name: "counter"
type: DT_RESOURCE
```

It is also possible to use a `tf.Variable` defined outside of the function, without explicitly passing it as an argument:

```
counter = tf.Variable(0)

@tf.function
def increment(c=1):
    return counter.assign_add(c)
```

The TF function will treat this as an implicit first argument, so it will actually end up with the same signature (except for the name of the argument). However, using global variables can quickly become messy, so you should generally wrap variables (and other resources) inside classes. The good news is @tf.function works fine with methods too:

```
class Counter:  
    def __init__(self):  
        self.counter = tf.Variable(0)  
  
    @tf.function  
    def increment(self, c=1):  
        return self.counter.assign_add(c)
```

WARNING

Do not use `=`, `+=`, `-=`, or any other Python assignment operator with TF variables. Instead, you must use the `assign()`, `assign_add()`, or `assign_sub()` methods. If you try to use a Python assignment operator, you will get an exception when you call the method.

A good example of this object-oriented approach is, of course, Keras. Let's see how to use TF functions with Keras.

Using TF Functions with Keras (or Not)

By default, any custom function, layer, or model you use with Keras will automatically be converted to a TF function; you do not need to do anything at all! However, in some cases you may want to deactivate this automatic conversion—for example, if your custom code cannot be turned into a TF function, or if you just want to debug your code (which is much easier in eager mode). To do this, you can simply pass `dynamic=True` when creating the model or any of its layers:

```
model = MyModel(dynamic=True)
```

If your custom model or layer will always be dynamic, you can instead call the base class's constructor with `dynamic=True`:

```
class MyDense(tf.keras.layers.Layer):
    def __init__(self, units, **kwargs):
        super().__init__(dynamic=True, **kwargs)
        [...]
```

Alternatively, you can pass `run_eagerly=True` when calling the `compile()` method:

```
model.compile(loss=my_mse, optimizer="nadam", metrics=[my_mae],
              run_eagerly=True)
```

Now you know how TF functions handle polymorphism (with multiple concrete functions), how graphs are automatically generated using AutoGraph and tracing, what graphs look like, how to explore their symbolic operations and tensors, how to handle variables and resources, and how to use TF functions with Keras.

¹ You can safely ignore it—it is only here for technical reasons, to ensure that TF functions don't leak internal structures.

2 A popular binary format discussed in [Chapter 13](#).

Index

Symbols

$@$ operator (matrix multiplication), [The Normal Equation](#)

β (momentum), [Momentum](#)

γ (gamma) value, [Gaussian RBF Kernel](#)

ϵ (tolerance), [Batch Gradient Descent](#), [SVM Classes and Computational Complexity](#)

ϵ greedy policy, [Exploration Policies](#)

ϵ neighborhood, [DBSCAN](#)

ϵ sensitive, [SVM Regression](#)

χ^2 test, [Regularization Hyperparameters](#)

ℓ_0 norm, [Select a Performance Measure](#)

ℓ_1 norm, [Select a Performance Measure](#)

ℓ_2 norm, [Select a Performance Measure](#)

ℓ_k norm, [Select a Performance Measure](#)

A

A/B experiments, [Training and Deploying TensorFlow Models at Scale](#)

accelerated k-means, [Accelerated k-means and mini-batch k-means](#)

accelerated linear algebra (XLA), [TensorFlow Functions and Graphs](#)

accuracy performance measure, [What Is Machine Learning?](#), [Measuring](#)

Accuracy Using Cross-Validation

ACF (autocorrelation function), [The ARMA Model Family](#)

action advantage, reinforcement learning, [Evaluating Actions: The Credit Assignment Problem](#)

action potentials (APs), [Biological Neurons](#)

actions, in reinforcement learning, [Learning to Optimize Rewards](#), [Evaluating Actions: The Credit Assignment Problem](#)-[Evaluating Actions: The Credit Assignment Problem](#)

activation functions, [The Perceptron](#), [The Multilayer Perceptron and Backpropagation](#)-[The Multilayer Perceptron and Backpropagation](#)

for Conv2D layer, [Implementing Convolutional Layers with Keras](#)

custom models, [Custom Activation Functions](#), [Initializers](#), [Regularizers](#), and [Constraints](#)

ELU, [ELU and SELU-GELU](#), Swish, and Mish

GELU, [GELU](#), Swish, and Mish-GELU, Swish, and Mish

hyperbolic tangent (htan), [The Multilayer Perceptron and Backpropagation](#), [Fighting the Unstable Gradients Problem](#)

hyperparameters, [Learning Rate](#), [Batch Size](#), and [Other Hyperparameters](#)

initialization parameters, [Glorot and He Initialization](#)

leakyReLU, [Leaky ReLU](#)-Leaky ReLU

Mish, [GELU](#), Swish, and Mish

PReLU, [Leaky ReLU](#)

ReLU (see ReLU)

RReLU, Leaky ReLU

SELU, ELU and SELU, Dropout

sigmoid, Estimating Probabilities, The Multilayer Perceptron and Backpropagation, The Vanishing/Exploding Gradients Problems, Sparse Autoencoders

SiLU, GELU, Swish, and Mish

softmax, Softmax Regression, Classification MLPs, Creating the model using the sequential API, An Encoder–Decoder Network for Neural Machine Translation

softplus, Regression MLPs

Swish, GELU, Swish, and Mish

active learning, Using Clustering for Semi-Supervised Learning

actor-critic, Overview of Some Popular RL Algorithms

actual class, Confusion Matrices

actual versus estimated probabilities, The ROC Curve

AdaBoost, AdaBoost-AdaBoost

AdaGrad, AdaGrad

Adam optimization, Adam, ℓ_1 and ℓ_2 Regularization

AdaMax, AdaMax

AdamW, AdamW, ℓ_1 and ℓ_2 Regularization

adaptive boosting (AdaBoost), Boosting-AdaBoost

adaptive instance normalization (AdaIN), StyleGANs

adaptive learning rate algorithms, AdaGrad-AdamW

adaptive moment estimation (Adam), [Adam](#)
additive attention, [Attention Mechanisms](#)
advantage actor-critic (A2C), [Overview of Some Popular RL Algorithms](#)
adversarial learning, [Semantic Segmentation](#), [Autoencoders](#), [GANs](#), and [Diffusion Models](#)
affine transformations, [StyleGANs](#)
affinity function, [Clustering Algorithms: k-means and DBSCAN](#)
affinity propagation, [Other Clustering Algorithms](#)
agents, reinforcement learning, [Reinforcement learning](#), [Learning to Optimize Rewards](#), [Policy Gradients](#), [Q-Learning](#)
agglomerative clustering, [Other Clustering Algorithms](#)
Akaike information criterion (AIC), [Selecting the Number of Clusters](#)
AlexNet, [AlexNet](#)
algorithms, preparing data for, [Prepare the Data for Machine Learning Algorithms-Transformation Pipelines](#)
alignment model, [Attention Mechanisms](#)
AllReduce algorithm, [Data parallelism using the mirrored strategy](#)
alpha dropout, [Dropout](#)
AlphaGo, [Reinforcement learning](#), [Reinforcement Learning](#), [Overview of Some Popular RL Algorithms](#)
anchor priors, [You Only Look Once](#)
ANNs (see [artificial neural networks](#))
anomaly detection, [Examples of Applications](#), [Unsupervised learning](#)

clustering for, [Clustering Algorithms: k-means and DBSCAN GMM](#), [Using Gaussian Mixtures for Anomaly Detection-Using Gaussian Mixtures for Anomaly Detection](#)
isolation forest, [Other Algorithms for Anomaly and Novelty Detection](#)

AP (average precision), [You Only Look Once](#)

APs (action potentials), [Biological Neurons](#)

area under the curve (AUC), [The ROC Curve](#)

argmax(), [Softmax Regression](#)

ARIMA model, [The ARMA Model Family-The ARMA Model Family](#)

ARMA model family, [The ARMA Model Family-The ARMA Model Family](#)

artificial neural networks (ANNs), [Introduction to Artificial Neural Networks with Keras-Learning Rate, Batch Size, and Other Hyperparameters](#)

autoencoders (see autoencoders)

backpropagation, [The Multilayer Perceptron and Backpropagation-The Multilayer Perceptron and Backpropagation](#)

biological neurons as background to, [Biological Neurons-Biological Neurons](#)

evolution of, [From Biological to Artificial Neurons-From Biological to Artificial Neurons](#)

hyperparameter fine-tuning, [Fine-Tuning Neural Network Hyperparameters-Learning Rate, Batch Size, and Other Hyperparameters](#)

implementing MLPs with Keras, [Implementing MLPs with Keras-Using TensorBoard for Visualization](#)

logical computations with neurons, [Logical Computations with](#)

Neurons-Logical Computations with Neurons

perceptrons (see multilayer perceptrons)

reinforcement learning policies, Neural Network Policies

artificial neuron, Logical Computations with Neurons

association rule learning, Unsupervised learning

assumptions, checking in model building, Data Mismatch, Check the Assumptions

asynchronous advantage actor-critic (A3C), Overview of Some Popular RL Algorithms

asynchronous gang scheduling, Bandwidth saturation

asynchronous updates, with centralized parameters, Asynchronous updates

à-trous convolutional layer, Semantic Segmentation

attention mechanisms, Natural Language Processing with RNNs and Attention, Attention Mechanisms-Multi-head attention, Vision Transformers

(see also transformer models)

attributes, Take a Quick Look at the Data Structure-Take a Quick Look at the Data Structure

categorical, Handling Text and Categorical Attributes, Handling Text and Categorical Attributes

combinations of, Experiment with Attribute Combinations-Experiment with Attribute Combinations

preprocessed, Take a Quick Look at the Data Structure

target, Take a Quick Look at the Data Structure

unsupervised learning, Supervised learning

AUC (area under the curve), [The ROC Curve](#)

autocorrelated time series, [Forecasting a Time Series](#)

autocorrelation function (ACF), [The ARMA Model Family](#)

autodiff (automatic differentiation), [Autodiff-Reverse-Mode Autodiff](#)

for computing gradients, [Computing Gradients Using Autodiff](#)-
[Computing Gradients Using Autodiff](#)

finite difference approximation, [Finite Difference Approximation](#)

forward-mode, [Forward-Mode Autodiff](#)-[Forward-Mode Autodiff](#)

manual differentiation, [Manual Differentiation](#)

reverse-mode, [Reverse-Mode Autodiff](#)-[Reverse-Mode Autodiff](#)

autoencoders, [Autoencoders, GANs, and Diffusion Models](#)-[Generating Fashion MNIST Images](#)

convolutional, [Convolutional Autoencoders](#)-[Convolutional Autoencoders](#)

denoising, [Denoising Autoencoders](#)-[Denoising Autoencoders](#)

efficient data representations, [Efficient Data Representations](#)-[Efficient Data Representations](#)

overcomplete, [Convolutional Autoencoders](#)

PCA with undercomplete linear autoencoder, [Performing PCA with an Undercomplete Linear Autoencoder](#)-[Performing PCA with an Undercomplete Linear Autoencoder](#)

sparse, [Sparse Autoencoders](#)-[Sparse Autoencoders](#)

stacked, [Stacked Autoencoders](#)-[Training One Autoencoder at a Time](#)

training one at a time, [Training One Autoencoder at a Time](#)-[Training](#)

One Autoencoder at a Time
undercomplete, Efficient Data Representations
variational, Variational Autoencoders-Variational Autoencoders
Autograph, AutoGraph and Tracing
AutoGraph, Using AutoGraph to Capture Control Flow
AutoML service, Fine-Tuning Neural Network Hyperparameters,
Hyperparameter Tuning on Vertex AI
autoregressive integrated moving average (ARIMA) model, The ARMA
Model Family-The ARMA Model Family
autoregressive model, The ARMA Model Family
autoregressive moving average (ARMA) model family, The ARMA Model
Family-The ARMA Model Family
auxiliary task, pretraining on, Pretraining on an Auxiliary Task
average absolute deviation, Select a Performance Measure
average pooling layer, Implementing Pooling Layers with Keras
average precision (AP), You Only Look Once

B

backbone, model, GoogLeNet
backpropagation, The Multilayer Perceptron and Backpropagation-The
Multilayer Perceptron and Backpropagation, The Vanishing/Exploding
Gradients Problems, Computing Gradients Using Autodiff, Generative
Adversarial Networks
backpropagation through time (BPTT), Training RNNs
bagging (bootstrap aggregating), Bagging and Pasting-Random Patches and

Random Subspaces

Bahdanau attention, [Attention Mechanisms](#)

balanced iterative reducing and clustering using hierarchies (BIRCH), [Other Clustering Algorithms](#)

bandwidth saturation, [Bandwidth saturation-Bandwidth saturation](#)

BaseEstimator, [Custom Transformers](#)

batch gradient descent, [Batch Gradient Descent](#)-[Batch Gradient Descent](#), [Ridge Regression](#)

batch learning, [Batch learning-Batch learning](#)

batch normalization (BN), [Batch Normalization](#)-[Implementing batch normalization with Keras](#), [Fighting the Unstable Gradients Problem](#)

batch predictions, [Deploying a new model version](#), [Creating a Prediction Service on Vertex AI](#), [Running Batch Prediction Jobs on Vertex AI](#)

Bayesian Gaussian mixtures, [Bayesian Gaussian Mixture Models](#)

Bayesian information criterion (BIC), [Selecting the Number of Clusters](#)

beam search, [Beam Search](#)-[Beam Search](#)

Bellman optimality equation, [Markov Decision Processes](#)

bias, [Learning Curves](#)

bias and fairness, NLP transformers, [Hugging Face's Transformers Library](#)

bias term constant, [Linear Regression](#), [The Normal Equation](#)

bias/variance trade-off, [Learning Curves](#)

BIC (Bayesian information criterion), [Selecting the Number of Clusters](#)

bidirectional recurrent layer, [Bidirectional RNNs](#)

binary classifiers, [Training a Binary Classifier](#), [Logistic Regression](#)

binary logarithm, [Computational Complexity](#)

binary trees, [Making Predictions](#), [Other Clustering Algorithms](#)

biological neural networks (BNNs), [Biological Neurons-Biological Neurons](#)

BIRCH (balanced iterative reducing and clustering using hierarchies), [Other Clustering Algorithms](#)

black box models, [Making Predictions](#)

blending, in stacking, [Stacking](#)

BN (batch normalization), [Batch Normalization](#), [Fighting the Unstable Gradients Problem](#)

BNNs (biological neural networks), [Biological Neurons-Biological Neurons](#)

boosting, [Boosting-Histogram-Based Gradient Boosting](#)

bootstrap aggregating (bagging), [Bagging and Pasting-Random Patches and Random Subspaces](#)

bootstrapping, [Bagging and Pasting](#)

bottleneck layers, [GoogLeNet](#)

bounding boxes, image identification, [Classification and Localization-You Only Look Once](#)

BPTT (backpropagation through time), [Training RNNs](#)

bucketizing a feature, [Feature Scaling and Transformation](#)

byte pair encoding (BPE), [Sentiment Analysis](#)

C

California Housing Prices dataset, [Working with Real Data-Check the](#)

Assumptions

callbacks, [Using Callbacks](#)-[Using Callbacks](#)

CART (Classification and Regression Tree) algorithm, [Making Predictions](#), [The CART Training Algorithm](#), [Regression](#)

catastrophic forgetting, [Implementing Deep Q-Learning](#)

categorical attributes, [Handling Text and Categorical Attributes](#), [Handling Text and Categorical Attributes](#)

categorical features, encoding, [The Hashing Layer](#)-[Encoding Categorical Features Using Embeddings](#)

CategoryEncoding layer, [The CategoryEncoding Layer](#)

causal model, [Forecasting Using a Sequence-to-Sequence Model](#), [Bidirectional RNNs](#)

centralized parameters, [Data parallelism with centralized parameters](#)-[Asynchronous updates](#)

centroid, cluster, [Clustering Algorithms: k-means and DBSCAN](#), [k-means](#), [The k-means algorithm](#)-Centroid initialization methods

chain rule, [The Multilayer Perceptron and Backpropagation](#)

chain-of-thought prompting, [An Avalanche of Transformer Models](#)

ChainClassifier, [Multilabel Classification](#)

chaining transformations, [Chaining Transformations](#)-[Chaining Transformations](#)

char-RNN model, [Generating Shakespearean Text Using a Character RNN](#)-[Stateful RNN](#)

chatbot or personal assistant, [Examples of Applications](#)

check_estimator(), [Custom Transformers](#)

chi-squared (χ^2) test, [Regularization Hyperparameters](#)

classification, [Classification-Multioutput Classification](#)

application examples of, [Examples of Applications-Examples of Applications](#)

binary classifier, [Training a Binary Classifier](#), [Logistic Regression](#)

CNNs, [Classification and Localization-You Only Look Once](#)

error analysis, [Error Analysis-Error Analysis](#)

hard margin, [Soft Margin Classification](#), [Under the Hood of Linear SVM Classifiers](#)

hard voting classifiers, [Voting Classifiers](#)

image (see images)

logistic regression (see logistic regression)

MLPs for, [Classification MLPs-Classification MLPs](#)

MNIST dataset, [MNIST-MNIST](#)

multiclass, [Multiclass Classification-Multiclass Classification](#), [Classification MLPs-Classification MLPs](#)

multilabel, [Multilabel Classification-Multilabel Classification](#)

multioutput, [Multioutput Classification-Multioutput Classification](#)

performance measures, [Performance Measures-The ROC Curve](#)

and regression, [Supervised learning](#), [Multioutput Classification](#)

soft margin, [Soft Margin Classification-Soft Margin Classification](#)

softmax regression, [Softmax Regression-Softmax Regression](#)

SVMs (see support vector machines)

text, **Sentiment Analysis-Reusing Pretrained Embeddings and Language Models**

voting classifiers, **Voting Classifiers-Voting Classifiers**

Classification and Regression Tree (CART) algorithm, **Making Predictions, The CART Training Algorithm, Regression**

clone(), Early Stopping

closed-form equation/solution, **Training Models, The Normal Equation, Ridge Regression, Training and Cost Function**

cloud platform deployment with Vertex AI, **Creating a Prediction Service on Vertex AI-Creating a Prediction Service on Vertex AI**

clustering algorithms, **Examples of Applications, Unsupervised learning, Unsupervised Learning Techniques-Other Clustering Algorithms**

affinity propagation, **Other Clustering Algorithms**

agglomerative clustering, **Other Clustering Algorithms**

applications for, **Clustering Algorithms: k-means and DBSCAN-Clustering Algorithms: k-means and DBSCAN**

BIRCH, **Other Clustering Algorithms**

DBSCAN, **DBSCAN-DBSCAN**

GMM, **Gaussian Mixtures-Other Algorithms for Anomaly and Novelty Detection**

image segmentation, **Clustering Algorithms: k-means and DBSCAN, Using Clustering for Image Segmentation-Using Clustering for Image Segmentation**

k-means (see k-means algorithm)

mean-shift, **Other Clustering Algorithms**

responsibilities of clusters for instances, [Gaussian Mixtures](#)

semi-supervised learning with, [Using Clustering for Semi-Supervised Learning](#)-[Using Clustering for Semi-Supervised Learning](#)

spectral clustering, [Other Clustering Algorithms](#)

CNNs (see convolutional neural networks)

Colab, [Running the Code Examples Using Google Colab](#)-[Running the Code Examples Using Google Colab](#)

color channels, [Stacking Multiple Feature Maps](#)

color segmentation, images, [Using Clustering for Image Segmentation](#)

column vectors, [Linear Regression](#)

ColumnTransformer, [Transformation Pipelines](#)

complex models with functional API, [Building Complex Models Using the Functional API](#)-[Building Complex Models Using the Functional API](#)

compound scaling, [Other Noteworthy Architectures](#)

compressed TFRecord files, [Compressed TFRecord Files](#)

compression and decompression, PCA, [PCA for Compression](#)-[PCA for Compression](#)

computation graphs, [A Quick Tour of TensorFlow](#)

computational complexity

DBSCAN, [DBSCAN](#)

decision trees, [Computational Complexity](#)

Gaussian mixture model, [Gaussian Mixtures](#)

histogram-based gradient boosting, [Histogram-Based Gradient Boosting](#)

k-means algorithm, [The k-means algorithm](#)
Normal equation, [Computational Complexity](#)
and SVM classes, [SVM Classes and Computational Complexity](#)
concatenative attention, [Attention Mechanisms](#)
concrete function, [TF Functions and Concrete Functions-Exploring Function Definitions and Graphs](#)
conditional GAN, [Deep Convolutional GANs](#)
conditional probability, [Beam Search](#)
confidence interval, [Evaluate Your System on the Test Set](#)
confusion matrix (CM), [Confusion Matrices-Confusion Matrices, Error Analysis-Error Analysis](#)
`ConfusionMatrixDisplay`, [Error Analysis](#)
connectionism, [From Biological to Artificial Neurons](#)
constrained optimization, [Under the Hood of Linear SVM Classifiers](#)
constraints, custom models, [Custom Activation Functions, Initializers, Regularizers, and Constraints](#)
convergence rate, [Batch Gradient Descent](#)
convex function, [Gradient Descent](#)
convex quadratic optimization, [Under the Hood of Linear SVM Classifiers](#)
convolution kernels (kernels), [Filters, CNN Architectures](#)
convolutional neural networks (CNNs), [Deep Computer Vision Using Convolutional Neural Networks-Semantic Segmentation](#)
architectures, [CNN Architectures-Choosing the Right CNN Architecture](#)

autoencoders, [Convolutional Autoencoders](#)-[Convolutional Autoencoders](#)

classification and localization, [Classification and Localization](#)-[You Only Look Once](#)

convolutional layers, [Convolutional Layers](#)-[Memory Requirements](#),
[Semantic Segmentation](#)-[Semantic Segmentation](#), Using 1D convolutional layers to process sequences, [WaveNet](#)-[WaveNet](#), Masking

evolution of, [Deep Computer Vision](#) Using Convolutional Neural Networks

GANs, [Deep Convolutional GANs](#)-[Deep Convolutional GANs](#)

object detection, [Object Detection](#)-[You Only Look Once](#)

object tracking, [Object Tracking](#)

pooling layers, [Pooling Layers](#)-[Implementing Pooling Layers with Keras](#)

pretrained models from Keras, [Using Pretrained Models from Keras](#)-[Using Pretrained Models from Keras](#)

ResNet-34 CNN using Keras, [Implementing a ResNet-34 CNN Using Keras](#)

semantic segmentation, [Examples of Applications](#), [Using Clustering for Image Segmentation](#), [Semantic Segmentation](#)-[Semantic Segmentation](#)

splitting across devices, [Model Parallelism](#)

transfer learning pretrained models, [Pretrained Models for Transfer Learning](#)-[Pretrained Models for Transfer Learning](#)

U-Net, [Diffusion Models](#)

and vision transformers, [Vision Transformers](#)

visual cortex architecture, [The Architecture of the Visual Cortex](#)

WaveNet, [WaveNet](#)-[WaveNet](#)

copy.deepcopy(), [Early Stopping](#)

core instance, [DBSCAN](#)

correlation coefficient, [Look for Correlations-Look for Correlations](#)

cost function, [Model-based learning and a typical machine learning workflow](#), [Select a Performance Measure](#)

in AdaBoost, [AdaBoost](#)

in autoencoders, [Performing PCA with an Undercomplete Linear Autoencoder](#)

in bounding box prediction model, [Classification and Localization](#)

in CART training algorithm, [The CART Training Algorithm, Regression](#)

in elastic net, [Elastic Net Regression](#)

in gradient descent, [Training Models, Gradient Descent-Gradient Descent](#), [Stochastic Gradient Descent-Stochastic Gradient Descent, The Vanishing/Exploding Gradients Problems](#)

in lasso regression, [Lasso Regression-Lasso Regression](#)

in Nesterov accelerated gradient, [Nesterov Accelerated Gradient](#)

in linear regression, [Linear Regression](#)

in logistic regression, [Training and Cost Function-Training and Cost Function](#)

in momentum optimization, [Momentum](#)

in ridge regression, [Ridge Regression](#)

in variational autoencoders, [Variational Autoencoders](#)

credit assignment problem, [Evaluating Actions: The Credit Assignment](#)

Problem-Evaluating Actions: The Credit Assignment Problem

cross entropy, Softmax Regression

cross-validation, Hyperparameter Tuning and Model Selection, Better Evaluation Using Cross-Validation-Better Evaluation Using Cross-Validation, Evaluate Your System on the Test Set, Measuring Accuracy Using Cross-Validation-Measuring Accuracy Using Cross-Validation, Learning Curves-Learning Curves

cross_val_predict(), Confusion Matrices, The Precision/Recall Trade-off, The ROC Curve, Error Analysis, Stacking

cross_val_score(), Better Evaluation Using Cross-Validation, Measuring Accuracy Using Cross-Validation

CUDA library, Getting Your Own GPU

curiosity-based exploration, Overview of Some Popular RL Algorithms

curriculum learning, Overview of Some Popular RL Algorithms

custom models and training algorithms, Customizing Models and Training Algorithms-Custom Training Loops

activation functions, Custom Activation Functions, Initializers, Regularizers, and Constraints

autodiff for computing gradients, Computing Gradients Using Autodiff-Computing Gradients Using Autodiff

constraints, Custom Activation Functions, Initializers, Regularizers, and Constraints

initializers, Custom Activation Functions, Initializers, Regularizers, and Constraints

layers, Custom Layers-Custom Layers

loss functions, Custom Loss Functions, Losses and Metrics Based on

Model Internals-Losses and Metrics Based on Model Internals

metrics, Custom Metrics-Custom Metrics, Losses and Metrics Based on Model Internals-Losses and Metrics Based on Model Internals

models, Custom Models-Custom Models

regularizers, Custom Activation Functions, Initializers, Regularizers, and Constraints

saving and loading models, Saving and Loading Models That Contain Custom Components-Saving and Loading Models That Contain Custom Components

training loops, Custom Training Loops-Custom Training Loops

custom transformers, Custom Transformers-Custom Transformers

customer segmentation, Clustering Algorithms: k-means and DBSCAN

D

DALL-E, Vision Transformers

data

downloading, Saving Your Code Changes and Your Data-Saving Your Code Changes and Your Data

efficient data representations, Efficient Data Representations-Efficient Data Representations

enqueueing and dequeuing, Queues

finding correlations in, Look for Correlations-Look for Correlations

making assumptions about, Data Mismatch

overfitting (see overfitting of data)

preparing for ML algorithms, Prepare the Data for Machine Learning

Algorithms-Transformation Pipelines

preparing for ML models, Preparing the Data for Machine Learning
Models-Preparing the Data for Machine Learning Models

preprocessing (see loading and preprocessing data)

shuffling of, Shuffling the Data

test data (see test set)

time series (see time series data)

training data (see training set)

underfitting of, Train and Evaluate on the Training Set, Learning
Curves-Learning Curves, Polynomial Kernel

unreasonable effectiveness, Insufficient Quantity of Training Data

visualizing (see visualization)

working with real data, Working with Real Data-Working with Real
Data

data analysis, clustering for, Clustering Algorithms: k-means and DBSCAN

data augmentation, Exercises, AlexNet, Pretrained Models for Transfer
Learning

data cleaning, Clean the Data-Clean the Data

data drift, Batch learning

data mining, Why Use Machine Learning?

data mismatch, Data Mismatch

data parallelism, Data Parallelism-Bandwidth saturation

data pipeline, Frame the Problem

data snooping bias, [Create a Test Set](#)

data structure, [Take a Quick Look at the Data Structure-Take a Quick Look at the Data Structure](#), [Special Data Structures-Queues](#)

data-efficient image transformers (DeiT), [Vision Transformers](#)

DataFrame, [Clean the Data](#), [Handling Text and Categorical Attributes](#), [Feature Scaling and Transformation](#)

Dataquest, [Other Resources](#)

Datasets library, [The TensorFlow Datasets Project](#)

DBSCAN, [DBSCAN-DBSCAN](#)

DCGANs (deep convolutional GANS), [Deep Convolutional GANs-Deep Convolutional GANs](#)

DDPM (denoising diffusion probabilistic model), [Autoencoders, GANs, and Diffusion Models](#), [Diffusion Models-Diffusion Models](#)

DDQN (dueling DQN), [Dueling DQN](#)

decision boundaries, [Decision Boundaries-Decision Boundaries](#), [Softmax Regression](#), [Making Predictions](#), [Sensitivity to Axis Orientation](#), [k-means](#)

decision function, [The Precision/Recall Trade-off](#), [Under the Hood of Linear SVM Classifiers-Under the Hood of Linear SVM Classifiers](#)

decision stumps, [AdaBoost](#)

decision threshold, [The Precision/Recall Trade-off-The Precision/Recall Trade-off](#)

decision trees, [Decision Trees-Decision Trees Have a High Variance](#), [Ensemble Learning and Random Forests](#)

(see also ensemble learning)

bagging and pasting, [Bagging and Pasting in Scikit-Learn](#)

CART training algorithm, [The CART Training Algorithm](#)

class probability estimates, [Estimating Class Probabilities](#)

computational complexity, [Computational Complexity](#)

and decision boundaries, [Making Predictions](#)

GINI impurity or entropy measures, [Gini Impurity or Entropy?](#)

high variance with, [Decision Trees Have a High Variance](#)

predictions, [Making Predictions-The CART Training Algorithm](#)

regression tasks, [Regression-Regression](#)

regularization hyperparameters, [Regularization Hyperparameters-](#)

[Regularization Hyperparameters](#)

sensitivity to axis orientation, [Sensitivity to Axis Orientation](#)

training and visualizing, [Training and Visualizing a Decision Tree-](#)

[Training and Visualizing a Decision Tree](#)

in training the model, [Train and Evaluate on the Training Set-Better](#)

[Evaluation Using Cross-Validation](#)

[DecisionTreeClassifier](#), [Training and Visualizing a Decision Tree](#), [Gini Impurity or Entropy?](#), [Regularization Hyperparameters](#), [Sensitivity to Axis Orientation](#), [Random Forests](#)

[DecisionTreeRegressor](#), [Train and Evaluate on the Training Set](#), [Decision Trees](#), [Regression](#), [Gradient Boosting](#)

`decision_function()`, [The Precision/Recall Trade-off](#)

deconvolution layer, [Semantic Segmentation](#)

deep autoencoders (see stacked autoencoders)

deep convolutional GANS (DCGANs), [Deep Convolutional GANs-Deep](#)

Convolutional GANs

deep Gaussian process, Monte Carlo (MC) Dropout

deep learning, The Machine Learning Tsunami

(see also deep neural networks; reinforcement learning)

deep neural networks (DNNs), Training Deep Neural Networks-Summary and Practical Guidelines

CNNs (see convolutional neural networks)

default configuration, Summary and Practical Guidelines

faster optimizers for, Faster Optimizers-AdamW

learning rate scheduling, Learning Rate Scheduling-Learning Rate Scheduling

MLPs (see multilayer perceptrons)

regularization, Avoiding Overfitting Through Regularization-Max-Norm Regularization

reusing pretrained layers, Reusing Pretrained Layers-Pretraining on an Auxiliary Task

RNNs (see recurrent neural networks)

and transfer learning, Self-supervised learning

unstable gradients, The Vanishing/Exploding Gradients Problems

vanishing and exploding gradients, The Vanishing/Exploding Gradients Problems-Gradient Clipping

deep neuroevolution, Fine-Tuning Neural Network Hyperparameters

deep Q-learning, Approximate Q-Learning and Deep Q-Learning-Dueling DQN

deep Q-networks (DQNs) (see Q-learning algorithm)

deepcopy(), Early Stopping

DeepMind, Reinforcement learning

degrees of freedom, Overfitting the Training Data, Learning Curves

DeiT (data-efficient image transformers), Vision Transformers

denoising autoencoders, Denoising Autoencoders-Denoising Autoencoders

denoising diffusion probabilistic model (DDPM), Autoencoders, GANs, and Diffusion Models, Diffusion Models-Diffusion Models

Dense layer, The Perceptron, Creating the model using the sequential API, Creating the model using the sequential API, Building Complex Models Using the Functional API

dense matrix, Feature Scaling and Transformation, Transformation Pipelines

density estimation, Unsupervised Learning Techniques, DBSCAN-DBSCAN, Gaussian Mixtures

density threshold, Using Gaussian Mixtures for Anomaly Detection

depth concatenation layer, GoogLeNet

depthwise separable convolution layer, Xception

deque, Implementing Deep Q-Learning

describe(), Take a Quick Look at the Data Structure

development set (dev set), Hyperparameter Tuning and Model Selection

differencing, time series forecasting, Forecasting a Time Series, Forecasting a Time Series, The ARMA Model Family

diffusion models, Autoencoders, GANs, and Diffusion Models, Diffusion Models-Diffusion Models

dilated filter, [Semantic Segmentation](#)

dilation rate, [Semantic Segmentation](#)

dimensionality reduction, [Unsupervised learning](#), [Dimensionality Reduction-Other Dimensionality Reduction Techniques](#)

approaches to, [Main Approaches for Dimensionality Reduction-Manifold Learning](#)

autoencoders, [Autoencoders, GANs, and Diffusion Models](#), [Efficient Data Representations-Performing PCA with an Undercomplete Linear Autoencoder](#)

choosing the right number of dimensions, [Choosing the Right Number of Dimensions](#)

clustering, [Clustering Algorithms: k-means and DBSCAN](#)

curse of dimensionality, [The Curse of Dimensionality-The Curse of Dimensionality](#)

for data visualization, [Dimensionality Reduction](#)

information loss from, [Dimensionality Reduction](#)

Isomap, [Other Dimensionality Reduction Techniques](#)

linear discriminant analysis, [Other Dimensionality Reduction Techniques](#)

LLE, [LLE-LLE](#)

multidimensional scaling, [Other Dimensionality Reduction Techniques](#)

PCA (see principal component analysis)

random projection algorithm, [Random Projection-Random Projection](#)

t-distributed stochastic neighbor embedding, [Other Dimensionality Reduction Techniques](#), [Visualizing the Fashion MNIST Dataset](#)

discount factor γ in reward system, [Evaluating Actions: The Credit Assignment Problem](#)

discounted rewards, [Evaluating Actions: The Credit Assignment Problem](#), [Policy Gradients](#)

Discretization layer, [The Discretization Layer](#)

discriminator, GAN, [Autoencoders, GANs, and Diffusion Models](#), [Generative Adversarial Networks-The Difficulties of Training GANs](#)

DistilBERT model, [An Avalanche of Transformer Models](#), Hugging Face's [Transformers Library](#)-Hugging Face's [Transformers Library](#)

distillation, [An Avalanche of Transformer Models](#)

distribution strategies API, [Training at Scale Using the Distribution Strategies API](#)

Docker container, [Installing and starting TensorFlow Serving](#)

dot product, [Attention Mechanisms](#)

Dota 2, [Overview of Some Popular RL Algorithms](#)

Double DQN, [Double DQN](#)

downloading data, [Saving Your Code Changes and Your Data-Saving Your Code Changes and Your Data](#)

DQNs (deep Q-networks) (see Q-learning algorithm)

drop(), [Prepare the Data for Machine Learning Algorithms](#)

dropna(), [Clean the Data](#)

Dropout, [Denoising Autoencoders](#)

dropout rate, [Dropout](#)

dropout regularization, [Dropout-Dropout](#)

dual numbers, [Forward-Mode Autodiff](#)

dual problem, [The Dual Problem-Kernelized SVMs](#)

dueling DQN (DDQN), [Dueling DQN](#)

dummy attributes, [Handling Text and Categorical Attributes](#)

dying ReLU problem, [Better Activation Functions](#)

dynamic models with subclassing API, [Using the Subclassing API to Build Dynamic Models-Using the Subclassing API to Build Dynamic Models](#)

dynamic programming, [Markov Decision Processes](#)

E

eager execution (eager mode), [AutoGraph and Tracing](#)

early stopping regularization, [Early Stopping-Early Stopping](#), [Gradient Boosting](#), [Forecasting Using a Linear Model](#)

edge computing, [Deploying a Model to a Mobile or Embedded Device](#)

efficient data representations, autoencoders, [Efficient Data Representations-Efficient Data Representations](#)

elastic net, [Elastic Net Regression](#)

EllipticEnvelope, [Other Algorithms for Anomaly and Novelty Detection](#)

ELMo (Embeddings from Language Models), [Reusing Pretrained Embeddings and Language Models](#)

ELU (exponential linear unit), [ELU and SELU-GELU](#), [Swish](#), and [Mish](#)

EM (expectation-maximization), [Gaussian Mixtures](#)

embedded device, deploying model to, [Deploying a Model to a Mobile or Embedded Device](#)-[Deploying a Model to a Mobile or Embedded Device](#)

embedded Reber grammars, [Exercises](#)

embedding matrix, [Encoding Categorical Features Using Embeddings](#), [An Encoder–Decoder Network for Neural Machine Translation](#)

embedding size, [An Encoder–Decoder Network for Neural Machine Translation](#), [Positional encodings](#)

embeddings, [Handling Text and Categorical Attributes](#)

encoding categorical features using, [Encoding Categorical Features Using Embeddings](#)-[Encoding Categorical Features Using Embeddings](#)

reusing pretrained, [Reusing Pretrained Embeddings and Language Models](#)-[Reusing Pretrained Embeddings and Language Models](#)

sentiment analysis, [Sentiment Analysis](#)

Embeddings from Language Models (ELMo), [Reusing Pretrained Embeddings and Language Models](#)

encoder, [Efficient Data Representations](#)

(see also autoencoders)

encoder–decoder models, [Input and Output Sequences](#), [Natural Language Processing with RNNs and Attention](#), [An Encoder–Decoder Network for Neural Machine Translation](#)-[Beam Search](#)

(see also attention mechanisms)

end-to-end ML project exercise, [End-to-End Machine Learning Project-Try It Out!](#)

building the model, [Look at the Big Picture](#)-[Check the Assumptions](#)

discovering and visualizing data, [Explore and Visualize the Data to Gain Insights](#)-[Experiment with Attribute Combinations](#)

fine-tuning your model, [Fine-Tune Your Model](#)-[Evaluate Your System](#)

on the Test Set

getting the data, [Get the Data-Create a Test Set](#)

preparing data for ML algorithms, [Prepare the Data for Machine Learning Algorithms-Transformation Pipelines](#)

real data, advantages of working with, [Working with Real Data-Working with Real Data](#)

selecting and training a model, [Train and Evaluate on the Training Set-Train and Evaluate on the Training Set](#)

endpoint, deploying model on GCP, [Creating a Prediction Service on Vertex AI](#)

ensemble learning, [Ensemble Learning and Random Forests-Stacking](#)

bagging and pasting, [Bagging and Pasting-Random Patches and Random Subspaces](#)

boosting, [Boosting-Histogram-Based Gradient Boosting](#)

cross-validation, [Better Evaluation Using Cross-Validation](#)

fine-tuning the system, [Ensemble Methods](#)

random forests (see random forests)

stacking, [Stacking-Stacking](#)

voting classifiers, [Voting Classifiers-Voting Classifiers](#)

entailment, [An Avalanche of Transformer Models](#)

entropy impurity measure, [Gini Impurity or Entropy?](#)

environments, reinforcement learning, [Learning to Optimize Rewards, Introduction to OpenAI Gym-Introduction to OpenAI Gym](#)

epochs, [Batch Gradient Descent](#)

equalized learning rate, GAN, [Progressive Growing of GANs](#)

equivariance, [Pooling Layers](#)

error analysis, classification, [Error Analysis-Error Analysis](#)

estimated versus actual probabilities, [The ROC Curve](#)

estimators, [Clean the Data](#), [Transformation Pipelines](#), [Voting Classifiers](#)

Euclidean norm, [Select a Performance Measure](#)

event files, TensorBoard, [Using TensorBoard for Visualization](#)

Example protobuf, [TensorFlow Protobufs](#)

exclusive or (XOR) problem, [The Perceptron](#)

exemplars, affinity propagation, [Other Clustering Algorithms](#)

expectation-maximization (EM), [Gaussian Mixtures](#)

experience replay, [The Difficulties of Training GANs](#)

explainability, attention mechanisms, [Vision Transformers](#)

explained variance ratio, [Explained Variance Ratio](#)

explained variance, plotting, [Choosing the Right Number of Dimensions](#)-
[Choosing the Right Number of Dimensions](#)

exploding gradients, [The Vanishing/Exploding Gradients Problems](#)

(see also vanishing and exploding gradients)

exploration policies, [Temporal Difference Learning](#), [Exploration Policies](#)

exploration/exploitation dilemma, reinforcement learning, [Neural Network Policies](#)

exponential linear unit (ELU), [ELU and SELU-GELU](#), [Swish](#), and [Mish](#)

exponential scheduling, [Learning Rate Scheduling](#), [Learning Rate Scheduling](#)

`export_graphviz()`, [Training and Visualizing a Decision Tree](#)

`extra-trees`, random forest, [Extra-Trees](#)

extremely randomized trees ensemble (`extra-trees`), [Extra-Trees](#)

F

face-recognition classifier, [Multilabel Classification](#)

false negatives, confusion matrix, [Confusion Matrices](#)

false positive rate (FPR) or fall-out, [The ROC Curve](#)

false positives, confusion matrix, [Confusion Matrices](#)

fan-in/fan-out, [Glorot and He Initialization](#)

fast-MCD, [Other Algorithms for Anomaly and Novelty Detection](#)

FCNs (fully convolutional networks), [Fully Convolutional Networks-Fully Convolutional Networks, Semantic Segmentation](#)

feature engineering, [Irrelevant Features](#), [Clustering Algorithms: k-means and DBSCAN](#)

feature extraction, [Unsupervised learning](#), [Irrelevant Features](#)

feature maps, [Stacking Multiple Feature Maps](#)-[Stacking Multiple Feature Maps](#), [Implementing Convolutional Layers with Keras](#), [ResNet](#), [SENet](#)

feature scaling, [Feature Scaling and Transformation](#)-[Feature Scaling and Transformation](#), [Gradient Descent](#), [Linear SVM Classification](#)

feature selection, [Irrelevant Features](#), [Analyzing the Best Models and Their Errors](#), [Lasso Regression](#), [Feature Importance](#)

feature vectors, [Select a Performance Measure](#), [Linear Regression](#), [Under the Hood of Linear SVM Classifiers](#)

features, [Supervised learning](#)

federated learning, [Running a Model in a Web Page](#)

feedforward neural network (FNN), [The Multilayer Perceptron and Backpropagation](#), [Recurrent Neurons and Layers](#)

`fetch_openml()`, [MNIST](#)

`fillna()`, [Clean the Data](#)

filters, convolutional layers, [Filters](#), [Implementing Convolutional Layers with Keras](#), [CNN Architectures](#), [Xception](#)

first moment (mean of gradient), [Adam](#)

first-order partial derivatives (Jacobians), [AdamW](#)

`fit()`

and custom transformers, [Custom Transformers](#), [Transformation Pipelines](#)

data cleaning, [Clean the Data](#)

versus `partial_fit()`, [Stochastic Gradient Descent](#)

using only with training set, [Feature Scaling and Transformation](#)

fitness function, [Model-based learning and a typical machine learning workflow](#)

`fit_transform()`, [Clean the Data](#), [Feature Scaling and Transformation](#), [Custom Transformers](#), [Transformation Pipelines](#)

fixed Q-value targets, [Fixed Q-value Targets](#)

flat dataset, [Preparing the Data for Machine Learning Models](#)

flowers dataset, [Pretrained Models for Transfer Learning](#)-[Pretrained Models for Transfer Learning](#)

FNN (feedforward neural network), [The Multilayer Perceptron and](#)

[Backpropagation](#), [Recurrent Neurons](#) and [Layers](#)

[folds](#), [Better Evaluation Using Cross-Validation](#), [MNIST](#), [Measuring Accuracy Using Cross-Validation](#), [Measuring Accuracy Using Cross-Validation](#)

forecasting time series (see time series data)

forget gate, LSTM, [LSTM cells](#)

forward pass, in backpropagation, [The Multilayer Perceptron](#) and [Backpropagation](#)

forward process, diffusion model, [Diffusion Models](#)-[Diffusion Models](#)

FPR (false positive rate) or fall-out, [The ROC Curve](#)

from_predictions(), [Error Analysis](#)

full gradient descent, [Batch Gradient Descent](#)

fully connected layer, [The Perceptron](#), [The Architecture of the Visual Cortex](#), [Memory Requirements](#)

fully convolutional networks (FCNs), [Fully Convolutional Networks](#)-[Fully Convolutional Networks](#), [Semantic Segmentation](#)

function definition (FuncDef), [TF Functions](#) and [Concrete Functions](#)

function graph (FuncGraph), [TF Functions](#) and [Concrete Functions](#)

functional API, complex models with, [Building Complex Models Using the Functional API](#)-[Building Complex Models Using the Functional API](#)

FunctionTransformer, [Custom Transformers](#)

F1 score, [Precision](#) and [Recall](#)

G

game play (see reinforcement learning)

gamma (γ) value, **Gaussian RBF Kernel**

GANs (see generative adversarial networks)

gate controllers, LSTM, **LSTM cells**

gated activation units, **WaveNet**

gated recurrent unit (GRU) cell, **GRU cells-GRU cells**, Masking

Gaussian distribution, Feature Scaling and Transformation, Training and Cost Function, **Variational Autoencoders-Variational Autoencoders**

Gaussian mixture model (GMM), **Gaussian Mixtures-Other Algorithms for Anomaly and Novelty Detection**

anomaly detection, **Using Gaussian Mixtures for Anomaly Detection-Using Gaussian Mixtures for Anomaly Detection**

Bayesian Gaussian mixtures, **Bayesian Gaussian Mixture Models**

fast-MCD, **Other Algorithms for Anomaly and Novelty Detection**

inverse_transform() with PCA, **Other Algorithms for Anomaly and Novelty Detection**

isolation forest, **Other Algorithms for Anomaly and Novelty Detection**

and k-means limitations, **Limits of k-means**

local outlier factor, **Other Algorithms for Anomaly and Novelty Detection**

one-class SVM, **Other Algorithms for Anomaly and Novelty Detection**

selecting number of clusters, **Selecting the Number of Clusters-Selecting the Number of Clusters**

Gaussian process, **Fine-Tuning Neural Network Hyperparameters**

Gaussian RBF kernel, **Custom Transformers, Gaussian RBF Kernel-SVM**

Classes and Computational Complexity, Kernelized SVMs

GBRT (gradient boosted regression trees), Gradient Boosting, Gradient Boosting-Histogram-Based Gradient Boosting

GCP (Google Cloud Platform), Creating a Prediction Service on Vertex AI-Creating a Prediction Service on Vertex AI

GCS (Google Cloud Storage), Creating a Prediction Service on Vertex AI

GD (see gradient descent)

GELU, GELU, Swish, and Mish-GELU, Swish, and Mish

generalization error, Testing and Validating-Hyperparameter Tuning and Model Selection

generative adversarial networks (GANs), Generative Adversarial Networks-StyleGANs

deep convolutional, Deep Convolutional GANs-Deep Convolutional GANs

progressive growing of, Progressive Growing of GANs-Progressive Growing of GANs

StyleGANs, StyleGANs-StyleGANs

training difficulties, The Difficulties of Training GANs-The Difficulties of Training GANs

generative autoencoders, Variational Autoencoders

generative models, Autoencoders, GANs, and Diffusion Models

(see also Gaussian mixture model)

generator, GAN, Autoencoders, GANs, and Diffusion Models, Generative Adversarial Networks-The Difficulties of Training GANs

genetic algorithm, Policy Search

geodesic distance, [Other Dimensionality Reduction Techniques](#)

geographic data, visualizing, [Visualizing Geographical Data](#)-[Visualizing Geographical Data](#)

`get_dummies()`, [Handling Text and Categorical Attributes](#)

`get_feature_names_out()`, [Custom Transformers](#)

`get_params()`, [Custom Transformers](#)

GINI impurity, [Making Predictions](#), [Gini Impurity or Entropy?](#)

global average pooling layer, [Implementing Pooling Layers with Keras](#)

global versus local minimum, gradient descent, [Gradient Descent](#)

Glorot initialization, [Glorot and He Initialization](#)-[Glorot and He Initialization](#)

GMM (see Gaussian mixture model)

Google Cloud Platform (GCP), [Creating a Prediction Service on Vertex AI](#)-[Creating a Prediction Service on Vertex AI](#)

Google Cloud Storage (GCS), [Creating a Prediction Service on Vertex AI](#)

Google Colab, [Running the Code Examples Using Google Colab](#)-[Running the Code Examples Using Google Colab](#)

Google Vertex AI (see Vertex AI)

GoogLeNet, [GoogLeNet](#)-[GoogLeNet](#), [Xception](#)

GPU implementation, [Mini-Batch Gradient Descent](#), [Using GPUs to Speed Up Computations](#)-[Parallel Execution Across Multiple Devices](#), [Training at Scale Using the Distribution Strategies API](#)

getting your own GPU, [Getting Your Own GPU](#)-[Getting Your Own GPU](#)

managing RAM, [Managing the GPU RAM](#)-[Managing the GPU RAM](#)

operations handling, [Parallel Execution Across Multiple Devices](#)

parallel execution across multiple devices, [Parallel Execution Across Multiple Devices](#)-[Parallel Execution Across Multiple Devices](#)

placing operations and variables on devices, [Placing Operations and Variables on Devices](#)-[Placing Operations and Variables on Devices](#)

gradient ascent, [Policy Search](#)

gradient boosted regression trees (GBRT), [Gradient Boosting](#), [Gradient Boosting](#)-[Histogram-Based Gradient Boosting](#)

gradient boosting, [Gradient Boosting](#)-[Gradient Boosting](#)

gradient clipping, [Gradient Clipping](#)

gradient descent (GD), [Training Models](#), [Gradient Descent](#)-[Mini-Batch Gradient Descent](#)

algorithm comparisons, [Mini-Batch Gradient Descent](#)

batch gradient descent, [Batch Gradient Descent](#)-[Batch Gradient Descent](#), [Ridge Regression](#)

local versus global minimum, [Gradient Descent](#)

mini-batch gradient descent, [Mini-Batch Gradient Descent](#)-[Mini-Batch Gradient Descent](#)

minimizing hinge loss, [Under the Hood of Linear SVM Classifiers](#)

versus momentum optimization, [Momentum](#)

with optimizers, [Faster Optimizers](#)-[AdamW](#)

shuffling data, [Shuffling the Data](#)

stochastic gradient descent, [Stochastic Gradient Descent](#)-[Stochastic Gradient Descent](#)

gradient tree boosting, [Gradient Boosting](#)

gradients

autodiff for computing, [Computing Gradients Using Autodiff](#)-
[Computing Gradients Using Autodiff](#)

bandwidth saturation issue, [Bandwidth saturation](#)

PG algorithm, [Policy Search](#), [Policy Gradients-Policy Gradients](#)

stale, [Asynchronous updates](#)

unstable (see vanishing and exploding gradients)

graph mode, [AutoGraph and Tracing](#)

graphical processing units (see GPU implementation)

graphs and functions, TensorFlow, [A Quick Tour of TensorFlow](#),
[TensorFlow Functions and Graphs](#)-[TF Function Rules](#), [TensorFlow Graphs](#)-
[Using TF Functions with Keras \(or Not\)](#)

Graphviz, [Training and Visualizing a Decision Tree](#)

greedy algorithm, CART as, [The CART Training Algorithm](#)

greedy decoding, [Generating Fake Shakespearean Text](#)

greedy layer-wise pretraining, [Unsupervised Pretraining](#), [Training One Autoencoder at a Time](#), [Progressive Growing of GANs](#)

grid search, [Grid Search-Grid Search](#)

GridSearchCV, [Grid Search-Grid Search](#)

gRPC API, querying through, [Querying TF Serving through the gRPC API](#)

GRU (gated recurrent unit) cell, [GRU cells-GRU cells](#), [Masking](#)

H

hard clustering, [k-means](#)

hard margin classification, [Soft Margin Classification](#), [Under the Hood of Linear SVM Classifiers](#)

hard voting classifiers, [Voting Classifiers](#)

harmonic mean, [Precision and Recall](#)

hashing collision, [The StringLookup Layer](#)

Hashing layer, [The Hashing Layer](#)

hashing trick, [The StringLookup Layer](#)

HDBSCAN (hierarchical DBSCAN), [DBSCAN](#)

He initialization, [Glorot and He Initialization](#)-[Glorot and He Initialization](#)

Heaviside step function, [The Perceptron](#)

heavy tail, feature distribution, [Feature Scaling and Transformation](#)

Hebb's rule, [The Perceptron](#)

Hebbian learning, [The Perceptron](#)

Hessians, [AdamW](#)

hidden layers

neurons per layer, [Number of Neurons per Hidden Layer](#)

number of, [Number of Hidden Layers](#)

stacked autoencoders, [Stacked Autoencoders](#)-[Training One Autoencoder at a Time](#)

hierarchical clustering, [Unsupervised learning](#)

hierarchical DBSCAN (HDBSCAN), [DBSCAN](#)

high variance, with decision trees, [Decision Trees Have a High Variance](#)

hinge loss function, Under the Hood of Linear SVM Classifiers

histogram-based gradient boosting (HGB), Histogram-Based Gradient Boosting-Histogram-Based Gradient Boosting

histograms, Take a Quick Look at the Data Structure

hold-out sets, Testing and Validating

holdout validation, Hyperparameter Tuning and Model Selection

housing dataset, Working with Real Data-Check the Assumptions

Huber loss, Regression MLPs, Custom Loss Functions, Custom Metrics, Forecasting Using a Linear Model

Hugging Face, Hugging Face's Transformers Library-Hugging Face's Transformers Library

Hungarian algorithm, Object Tracking

Hyperband tuner, Fine-Tuning Neural Network Hyperparameters

hyperbolic tangent (htan), The Multilayer Perceptron and Backpropagation, Fighting the Unstable Gradients Problem

hyperparameters, Overfitting the Training Data, Fine-Tuning Neural Network Hyperparameters-Learning Rate, Batch Size, and Other Hyperparameters

activation function, Learning Rate, Batch Size, and Other Hyperparameters

batch size, Learning Rate, Batch Size, and Other Hyperparameters

CART algorithm, The CART Training Algorithm

convolutional layers, Implementing Convolutional Layers with Keras

in custom transformations, Custom Transformers

decision tree, Gradient Boosting

dimensionality reduction, Choosing the Right Number of Dimensions

gamma (γ) value, Gaussian RBF Kernel

GAN challenges, The Difficulties of Training GANs

Keras Tuner, Hyperparameter Tuning on Vertex AI

learning rate, Gradient Descent, Learning Rate, Batch Size, and Other Hyperparameters

momentum β , Momentum

Monte Carlo samples, Monte Carlo (MC) Dropout

neurons per hidden layer, Number of Neurons per Hidden Layer

and normalization, Feature Scaling and Transformation

number of hidden layers, Number of Hidden Layers

number of iterations, Learning Rate, Batch Size, and Other Hyperparameters

optimizer, Learning Rate, Batch Size, and Other Hyperparameters

PG algorithms, Policy Gradients

preprocessor and model interaction, Grid Search

randomized search, Fine-Tuning Neural Network Hyperparameters-Fine-Tuning Neural Network Hyperparameters

saving along with model, Custom Activation Functions, Initializers, Regularizers, and Constraints

SGDClassifier, SVM Classes and Computational Complexity

subsample, Gradient Boosting

SVM classifiers with polynomial kernel, Polynomial Kernel

tolerance (ϵ), [SVM Classes](#) and Computational Complexity

tuning of, [Hyperparameter Tuning](#) and [Model Selection-Hyperparameter Tuning](#) and [Model Selection](#), [Grid Search-Grid Search](#), [Evaluate Your System on the Test Set](#), [Training](#) and evaluating the model,
[Hyperparameter Tuning on Vertex AI](#)-[Hyperparameter Tuning on Vertex AI](#)

hypothesis, [Select a Performance Measure](#)

hypothesis boosting (see [boosting](#))

hypothesis function, [Linear Regression](#)

I

identity matrix, [Ridge Regression](#)

IID (see [independent and identically distributed](#))

image generation, [Semantic Segmentation](#), [StyleGANs](#), [Diffusion Models](#)

image segmentation, [Clustering Algorithms: k-means and DBSCAN](#), [Using Clustering for Image Segmentation](#)-[Using Clustering for Image Segmentation](#)

images, classifying and generating, [Examples of Applications](#)

autoencoders (see [autoencoders](#))

CNNs (see [convolutional neural networks](#))

diffusion models, [Diffusion Models-Diffusion Models](#)

generating with GANs, [Generative Adversarial Networks-StyleGANs](#)

implementing MLPs, [Building an Image Classifier Using the Sequential API](#)-[Using the model to make predictions](#)

labels, [Classification and Localization](#)

loading and preprocessing data, [Image Preprocessing Layers](#)

representative images, [Using Clustering for Semi-Supervised Learning](#)
semantic segmentation, [Using Clustering for Image Segmentation](#)
tuning hyperparameters, [Fine-Tuning Neural Network Hyperparameters](#)
importance sampling (IS), [Prioritized Experience Replay](#)
impurity measures, [Making Predictions, Gini Impurity or Entropy?](#)
imputation, [Clean the Data](#)
incremental learning, [Online learning](#)
incremental PCA (IPCA), [Incremental PCA-Incremental PCA](#)
independent and identically distributed (IID), training instances as, [Stochastic Gradient Descent](#)
inductive bias, [Vision Transformers](#)
inertia, model, [Centroid initialization methods, Accelerated k-means and mini-batch k-means](#)
inference, [Model-based learning and a typical machine learning workflow](#)
info(), [Take a Quick Look at the Data Structure](#)
information theory, [Softmax Regression, Gini Impurity or Entropy?](#)
inliers, [Unsupervised Learning Techniques](#)
input and output sequences, RNNs, [Input and Output Sequences-Input and Output Sequences](#)
input gate, LSTM, [LSTM cells](#)
input layer, neural network, [The Perceptron, Creating the model using the sequential API, Building Complex Models Using the Functional API](#)
(see also hidden layers)

input signature, [TF Functions and Concrete Functions](#)

instance segmentation, [Using Clustering for Image Segmentation, Semantic Segmentation](#)

instance-based learning, [Instance-based learning, Model-based learning and a typical machine learning workflow](#)

inter-op thread pool, [Parallel Execution Across Multiple Devices](#)

intercept term constant, [Linear Regression](#)

interleaving lines from multiple files, [Interleaving Lines from Multiple Files-Interleaving Lines from Multiple Files](#)

interpretable ML, [Making Predictions](#)

invariance, max pooling layer, [Pooling Layers](#)

inverse_transform(), [Feature Scaling and Transformation, Custom Transformers, PCA for Compression, Other Algorithms for Anomaly and Novelty Detection](#)

IPCA (incremental PCA), [Incremental PCA-Incremental PCA](#)

iris dataset, [Decision Boundaries](#)

irreducible error, [Learning Curves](#)

IS (importance sampling), [Prioritized Experience Replay](#)

isolation forest, [Other Algorithms for Anomaly and Novelty Detection](#)

Isomap, [Other Dimensionality Reduction Techniques](#)

isotropic noise, [Diffusion Models](#)

IterativeImputer, [Clean the Data](#)

J

Jacobians, AdamW

joblib library, Launch, Monitor, and Maintain Your System-Launch, Monitor, and Maintain Your System

JSON Lines, Running Batch Prediction Jobs on Vertex AI, Running Batch Prediction Jobs on Vertex AI

Jupyter, Running the Code Examples Using Google Colab

K

k-fold cross-validation, Better Evaluation Using Cross-Validation, Measuring Accuracy Using Cross-Validation, Measuring Accuracy Using Cross-Validation

k-means algorithm, Custom Transformers, k-means-Limits of k-means

accelerated k-means, Accelerated k-means and mini-batch k-means

centroid initialization methods, Centroid initialization methods-Centroid initialization methods

finding optimal number of clusters, Finding the optimal number of clusters-Finding the optimal number of clusters

limitations of, Limits of k-means

mini-batch k-means, Accelerated k-means and mini-batch k-means

workings of, The k-means algorithm-The k-means algorithm

k-means++, Centroid initialization methods

k-nearest neighbors regression, Model-based learning and a typical machine learning workflow

Kaiming initialization, Glorot and He Initialization

Kalman Filters, Object Tracking

Keras API, [Objective and Approach](#), [A Quick Tour of TensorFlow](#)

(see also [artificial neural networks](#))

and accessing TensorFlow API directly, [Image Preprocessing Layers](#) activation function support, [Leaky ReLU](#), [ELU](#) and [SELU](#), [GELU](#), [Swish](#), and [Mish](#)

convolutional layer implementation, [Implementing Convolutional Layers with Keras](#)-[Implementing Convolutional Layers with Keras](#)

custom functions in, [TensorFlow Functions and Graphs](#)

gradient clipping, [Gradient Clipping](#)

image preprocessing layers, [Image Preprocessing Layers](#)

implementing MLPs with, [Implementing MLPs with Keras](#)-[Using TensorBoard for Visualization](#)

initialization handling, [Glorot and He Initialization](#)

initializers, [Creating the model using the sequential API](#)

layer preprocessing, [Keras Preprocessing Layers](#)-[Using Pretrained Language Model Components](#)

learning rate scheduling, [Learning Rate Scheduling](#)-[Learning Rate Scheduling](#)

loading a dataset, [Building an Image Classifier Using the Sequential API](#)

PG algorithm, [Policy Gradients](#)-[Policy Gradients](#)

pool layer implementation, [Implementing Pooling Layers with Keras](#)-[Implementing Pooling Layers with Keras](#)

pretrained CNN models, [Using Pretrained Models from Keras](#)-[Using Pretrained Models from Keras](#)

ResNet-34 CNN with, [Implementing a ResNet-34 CNN Using Keras](#)

saving models, [Exporting SavedModels, Deploying a Model to a Mobile or Embedded Device](#)

stacked encoder implementation, [Implementing a Stacked Autoencoder Using Keras](#)

tf.data API dataset, [Using the Dataset with Keras-Using the Dataset with Keras](#)

tf.keras library, [Using Keras to load the dataset](#)

tf.keras.activations.get(), [Custom Layers](#)

tf.keras.activations.relu(), [Creating the model using the sequential API](#)

tf.keras.applications module, [Using Pretrained Models from Keras](#)

tf.keras.applications.xception.preprocess_input(), [Pretrained Models for Transfer Learning](#)

tf.keras.backend module, [Tensors and Operations](#)

tf.keras.callbacks.EarlyStopping, [Using Callbacks](#)

tf.keras.callbacks.LearningRateScheduler, [Learning Rate Scheduling](#)

tf.keras.callbacks.ModelCheckpoint, [Using Callbacks](#)

tf.keras.callbacks.TensorBoard, [Using TensorBoard for Visualization, Masking](#)

tf.keras.datasets.imdb.load_data(), [Sentiment Analysis](#)

tf.keras.initializers.VarianceScaling, [Glorot and He Initialization](#)

tf.keras.layers.ActivityRegularization, [Sparse Autoencoders](#)

tf.keras.layers.AdditiveAttention, [Attention Mechanisms](#)

tf.keras.layers.Attention, [Attention Mechanisms](#)

`tf.keras.layers.AvgPool2D`, **Implementing Pooling Layers with Keras**

`tf.keras.layers.BatchNormalization`, **Batch Normalization, Implementing batch normalization with Keras-Implementing batch normalization with Keras, Fighting the Unstable Gradients Problem**

`tf.keras.layers.Bidirectional`, **Bidirectional RNNs**

`tf.keras.layers.CategoryEncoding`, **The CategoryEncoding Layer**

`tf.keras.layers.CenterCrop`, **Image Preprocessing Layers**

`tf.keras.layers.Concatenate`, **Building Complex Models Using the Functional API, Building Complex Models Using the Functional API, GoogLeNet**

`tf.keras.layers.Conv1D`, **Semantic Segmentation, Masking**

`tf.keras.layers.Conv2D`, **Implementing Convolutional Layers with Keras**

`tf.keras.layers.Conv2DTranspose`, **Semantic Segmentation**

`tf.keras.layers.Conv3D`, **Semantic Segmentation**

`tf.keras.layers.Dense`, **Building Complex Models Using the Functional API, Building Complex Models Using the Functional API, Custom Layers-Custom Layers, Encoding Categorical Features Using Embeddings, Tying Weights, Variational Autoencoders**

`tf.keras.layers.Discretization`, **The Discretization Layer**

`tf.keras.layers.Dropout`, **Dropout**

`tf.keras.layers.Embedding`, **Encoding Categorical Features Using Embeddings, Building and Training the Char-RNN Model, Sentiment Analysis, Positional encodings**

`tf.keras.layers.GlobalAvgPool2D`, **Implementing Pooling Layers with Keras**

`tf.keras.layers.GRU`, [GRU cells](#)

`tf.keras.layers.GRUCell`, [GRU cells](#)

`tf.keras.layers.Hashing`, [The Hashing Layer](#)

`tf.keras.layers.Input`, [Building Complex Models Using the Functional API](#)

`tf.keras.layers.Lambda`, [Custom Layers](#), [Building and Training the Char-RNN Model](#)

`tf.keras.layers.LayerNormalization`, [Fighting the Unstable Gradients Problem](#)

`tf.keras.layers.LeakyReLU`, [Leaky ReLU](#)

`tf.keras.layers.LSTM`, [LSTM cells](#)

`tf.keras.layers.Masking`, [Masking](#)

`tf.keras.layers.MaxPool2D`, [Implementing Pooling Layers with Keras](#)

`tf.keras.layers.MultiHeadAttention`, [Multi-head attention](#)

`tf.keras.layers.Normalization`, [Building a Regression MLP Using the Sequential API](#), [Building Complex Models Using the Functional API](#), [The Normalization Layer-The Normalization Layer](#)

`tf.keras.layers.PReLU`, [Leaky ReLU](#)

`tf.keras.layers.Rescaling`, [Image Preprocessing Layers](#)

`tf.keras.layers.Resizing`, [Image Preprocessing Layers](#), [Using Pretrained Models from Keras](#)

`tf.keras.layers.RNN`, [LSTM cells](#)

`tf.keras.layers.SeparableConv2D`, [Xception](#)

`tf.keras.layers.StringLookup`, [The StringLookup Layer](#)

`tf.keras.layers.TextVectorization`, [Text Preprocessing](#)-[Text Preprocessing](#), [Creating the Training Dataset](#), [Building and Training the Char-RNN Model](#), [Sentiment Analysis](#), [Sentiment Analysis](#)-[An Encoder–Decoder Network for Neural Machine Translation](#)

`tf.keras.layers.TimeDistributed`, [Forecasting Using a Sequence-to-Sequence Model](#)

`tf.keras.losses.Huber`, [Custom Loss Functions](#)

`tf.keras.losses.kullback_leibler_divergence()`, [Sparse Autoencoders](#)

`tf.keras.losses.Loss`, [Saving and Loading Models That Contain Custom Components](#)

`tf.keras.losses.sparse_categorical_crossentropy()`, [Compiling the model](#), [CNN Architectures](#), [Building and Training the Char-RNN Model](#), [An Encoder–Decoder Network for Neural Machine Translation](#), [Hugging Face’s Transformers Library](#)

`tf.keras.metrics.MeanIoU`, [Classification and Localization](#)

`tf.keras.metrics.Metric`, [Custom Metrics](#)

`tf.keras.metrics.Precision`, [Custom Metrics](#)

`tf.keras.Model`, [Building Complex Models Using the Functional API](#)

`tf.keras.models.clone_model()`, [Using the Subclassing API to Build Dynamic Models](#), [Transfer Learning with Keras](#)

`tf.keras.models.load_model()`, [Saving and Restoring a Model](#), [Saving and Loading Models That Contain Custom Components](#)-[Saving and Loading Models That Contain Custom Components](#), [Custom Models](#), [Training at Scale Using the Distribution Strategies API](#)

`tf.keras.optimizers.Adam`, [Building a Regression MLP Using the Sequential API](#), [AdamW](#)

`tf.keras.optimizers.Adamax`, [AdamW](#)

`tf.keras.optimizers.experimental.AdamW`, [AdamW](#)

`tf.keras.optimizers.Nadam`, [AdamW](#)

`tf.keras.optimizers.schedules`, [Learning Rate Scheduling](#)

`tf.keras.optimizers.SGD`, [Momentum](#)

`tf.keras.preprocessing.image.ImageDataGenerator`, [Pretrained Models for Transfer Learning](#)

`tf.keras.regularizers.l1_l2()`, [l1 and l2 Regularization](#)

`tf.keras.Sequential`, [Creating the model using the sequential API](#),
[Building a Regression MLP Using the Sequential API](#), [CNN Architectures](#)

`tf.keras.utils.get_file()`, [Creating the Training Dataset](#)

`tf.keras.utils.set_random_seed()`, [Creating the model using the sequential API](#)

`tf.keras.utils.timeseries_dataset_from_array()`, [Preparing the Data for Machine Learning Models](#), [Preparing the Data for Machine Learning Models](#)

`tf.keras.utils.to_categorical()`, [Compiling the model](#)

time series forecasting for RNN, [Forecasting Using a Simple RNN](#)-
[Forecasting Using a Deep RNN](#)

transfer learning with, [Transfer Learning with Keras](#)-[Transfer Learning with Keras](#)

using TF functions (or not), [Using TF Functions with Keras \(or Not\)](#)

Keras session, [Creating the model using the sequential API](#)

Keras Tuner, [Hyperparameter Tuning on Vertex AI](#)

kernel trick, **Polynomial Kernel-SVM** Classes and Computational Complexity, **Kernelized SVMs**-**Kernelized SVMs**

kernelized SVMs, **Kernelized SVMs**-**Kernelized SVMs**

kernels (convolution kernels), **Filters**, **CNN Architectures**

kernels (runtimes), **Running the Code Examples Using Google Colab**

KL (Kullback-Leibler) divergence, **Softmax Regression**, **Sparse Autoencoders**

KLDivergenceRegularizer, **Sparse Autoencoders**

KMeans, **Custom Transformers**

KNeighborsClassifier, **Multilabel Classification**, **Multioutput Classification**

KNNImputer, **Clean the Data**

Kullback-Leibler (KL) divergence, **Softmax Regression**, **Sparse Autoencoders**

L

label propagation, **Using Clustering for Semi-Supervised Learning**-**Using Clustering for Semi-Supervised Learning**

labels, **Frame the Problem**

in clustering, **k-means**

image classification, **Classification and Localization**

supervised learning, **Supervised learning**

unlabeled data issue, **Unsupervised Pretraining Using Stacked Autoencoders**

landmarks, **Similarity Features**

language models, [Generating Shakespearean Text Using a Character RNN](#)

(see also natural language processing)

large margin classification, [Linear SVM Classification](#)

Lasso, [Lasso Regression](#)

lasso regression, [Lasso Regression-Lasso Regression](#)

latent diffusion models, [Diffusion Models](#)

latent loss, [Variational Autoencoders](#)

latent representation of inputs, [Vision Transformers](#), [Autoencoders](#), [GANs](#), [and Diffusion Models](#), [Efficient Data Representations](#), [Deep Convolutional GANs](#)

latent space, [Variational Autoencoders](#)

law of large numbers, [Voting Classifiers](#)

layer normalization, [Exercises](#), [Processing Sequences Using RNNs and CNNs](#), [Fighting the Unstable Gradients Problem](#)

LDA (linear discriminant analysis), [Other Dimensionality Reduction Techniques](#)

leaf node, decision tree, [Making Predictions](#), [Estimating Class Probabilities](#), [Regularization Hyperparameters](#)

leakyReLU, [Leaky ReLU-Leaky ReLU](#)

learning curves, overfit or underfit analysis, [Learning Curves-Learning Curves](#)

learning rate, [Online learning](#), [Gradient Descent](#), [Batch Gradient Descent](#), [Stochastic Gradient Descent](#), [Learning Rate](#), [Batch Size](#), and [Other Hyperparameters](#), [Q-Learning](#)

learning rate schedules, [Learning Rate Scheduling-Learning Rate Scheduling](#)

learning schedules, [Stochastic Gradient Descent](#)

LeCun initialization, [Glorot and He Initialization](#)

LeNet-5, [The Architecture of the Visual Cortex, LeNet-5](#)

Levenshtein distance, [Gaussian RBF Kernel](#)

liblinear library, [SVM Classes and Computational Complexity](#)

libsvm library, [SVM Classes and Computational Complexity](#)

life satisfaction dataset, [Model-based learning and a typical machine learning workflow](#)

likelihood function, [Selecting the Number of Clusters-Selecting the Number of Clusters](#)

linear discriminant analysis (LDA), [Other Dimensionality Reduction Techniques](#)

linear models

forecasting time series, [Forecasting Using a Linear Model](#)

linear regression (see linear regression)

regularized, [Regularized Linear Models-Early Stopping](#)

SVM, [Linear SVM Classification-Soft Margin Classification](#)

training and running example, [Model-based learning and a typical machine learning workflow](#)

linear regression, [Model-based learning and a typical machine learning workflow-Model-based learning and a typical machine learning workflow, Linear Regression-Mini-Batch Gradient Descent](#)

comparison of algorithms, [Mini-Batch Gradient Descent](#)

computational complexity, [Computational Complexity](#)

gradient descent in, [Gradient Descent-Mini-Batch Gradient Descent](#)
learning curves in, [Learning Curves-Learning Curves](#)
Normal equation, [The Normal Equation-The Normal Equation](#)
regularizing models (see regularization)
ridge regression, [Ridge Regression-Ridge Regression, Elastic Net Regression](#)
training set evaluation, [Train and Evaluate on the Training Set](#)
using stochastic gradient descent, [Stochastic Gradient Descent](#)
linear SVM classification, [Linear SVM Classification-Soft Margin Classification, Under the Hood of Linear SVM Classifiers-Kernelized SVMs](#)
linear threshold units (LTUs), [The Perceptron](#)
linearly separable, SVM classes, [Linear SVM Classification-Linear SVM Classification, Similarity Features](#)
[LinearRegression](#), [Clean the Data, Feature Scaling and Transformation, Train and Evaluate on the Training Set, The Normal Equation, Computational Complexity, Polynomial Regression](#)
[LinearSVC](#), [Soft Margin Classification, SVM Classes and Computational Complexity, Under the Hood of Linear SVM Classifiers](#)
[LinearSVR](#), [SVM Regression-SVM Regression](#)
Lipschitz continuous, derivative as, [Gradient Descent](#)
LLE (locally linear embedding), [LLE-LLE](#)
loading and preprocessing data, [Loading and Preprocessing Data with TensorFlow-The TensorFlow Datasets Project](#)
image preprocessing layers, [Image Preprocessing Layers](#)

layer preprocessing in Keras, [Loading and Preprocessing Data with TensorFlow](#), [Keras Preprocessing Layers-Using Pretrained Language Model Components](#)

`tf.data` API, [Loading and Preprocessing Data with TensorFlow-Using the Dataset with Keras](#)

TFDS project, [The TensorFlow Datasets Project-The TensorFlow Datasets Project](#)

TFRecord format, [Loading and Preprocessing Data with TensorFlow](#), [The TFRecord Format-Handling Lists of Lists Using the SequenceExample Protobuf](#)

local outlier factor (LOF), [Other Algorithms for Anomaly and Novelty Detection](#)

local receptive field, [The Architecture of the Visual Cortex](#)

local response normalization (LRN), [AlexNet](#)

local versus global minimum, gradient descent, [Gradient Descent](#)

locality sensitive hashing (LSH), [Random Projection](#)

localization, CNNs, [Classification and Localization-Object Detection](#)

locally linear embedding (LLE), [LLE-LLE](#)

LOF (local outlier factor), [Other Algorithms for Anomaly and Novelty Detection](#)

log loss, [Training and Cost Function](#)

log-odds function, [Estimating Probabilities](#)

log-transformer, [Custom Transformers](#)

logical GPU device, [Managing the GPU RAM](#)

logistic function, [Estimating Probabilities](#)

logistic regression, Supervised learning, Feature Scaling and Transformation, Logistic Regression-Softmax Regression

decision boundaries illustration, Decision Boundaries-Decision Boundaries

estimating probabilities, Estimating Probabilities-Estimating Probabilities

softmax regression model, Softmax Regression-Softmax Regression

training and cost function, Training and Cost Function-Training and Cost Function

LogisticRegression, Decision Boundaries, Softmax Regression

logit function, Estimating Probabilities

long sequences, training RNN on, Handling Long Sequences-WaveNet

short-term memory problem, Tackling the Short-Term Memory Problem-WaveNet

unstable gradients problem, Fighting the Unstable Gradients Problem-Fighting the Unstable Gradients Problem

long short-term memory (LSTM) cell, LSTM cells-LSTM cells, Masking, An Encoder–Decoder Network for Neural Machine Translation, Bidirectional RNNs

loss functions

based on model internals, Losses and Metrics Based on Model Internals-Losses and Metrics Based on Model Internals

custom, Custom Loss Functions

versus metrics, Custom Metrics

output, Building Complex Models Using the Functional API

LRN (local response normalization), [AlexNet](#)

LSH (locality sensitive hashing), [Random Projection](#)

LSTM (long short-term memory) cell, [LSTM cells-LSTM cells](#), [Masking](#), [An Encoder–Decoder Network for Neural Machine Translation](#), [Bidirectional RNNs](#)

LTUs (linear threshold units), [The Perceptron](#)

Luong attention, [Attention Mechanisms](#)

M

machine learning (ML), [What Is Machine Learning?](#)

application/technique examples, [Examples of Applications-Examples of Applications](#)

challenges of, [Main Challenges of Machine Learning-Stepping Back](#)

notations, [Select a Performance Measure-Select a Performance Measure](#)

project checklist, [Look at the Big Picture](#)

(see also end-to-end ML project exercise)

reasons for using, [Why Use Machine Learning?-Why Use Machine Learning?](#)

resources on, [Other Resources-Other Resources](#)

spam filter example, [The Machine Learning Landscape-Why Use Machine Learning?](#)

testing and validating, [Testing and Validating-Data Mismatch](#)

types of systems, [Types of Machine Learning Systems-Model-based learning and a typical machine learning workflow](#)

MAE (mean absolute error), [Select a Performance Measure](#)

majority-vote predictions, [Exercises](#)

`make_column_selector()`, [Transformation Pipelines](#)

`make_column_transformer()`, [Transformation Pipelines](#)

`make_pipeline()`, [Transformation Pipelines](#), [Learning Curves](#)

Manhattan norm, [Select a Performance Measure](#)

manifold hypothesis, [Manifold Learning](#)

manifold learning, dimension reduction, [Manifold Learning](#)-[Manifold Learning](#)

MAP (maximum a-posteriori) estimation, [Selecting the Number of Clusters](#)

mAP (mean average precision), [You Only Look Once](#)

MAPE (mean absolute percentage error), [Forecasting a Time Series](#)

mapping network, [StyleGANs](#), [StyleGANs](#)

MapReduce, [Frame the Problem](#)

margin violations, [Soft Margin Classification](#), [Under the Hood of Linear SVM Classifiers](#)

Markov chains, [Markov Decision Processes](#)

Markov decision processes (MDPs), [Markov Decision Processes](#)-[Markov Decision Processes](#), [Exploration Policies](#)

mask R-CNN, [Semantic Segmentation](#)

mask tensor, [Masking](#)

masked language model (MLM), [An Avalanche of Transformer Models](#)

masked multi-head attention layer, [Attention Is All You Need: The Original Transformer Architecture](#)

masking, [Masking-Masking](#), Multi-head attention

Matching Engine service, Vertex AI, [Creating a Prediction Service on Vertex AI](#)

Matplotlib, [Running the Code Examples Using Google Colab](#)

max pooling layer, [Pooling Layers](#), [CNN Architectures](#)

max-norm regularization, [Max-Norm Regularization](#)

maximization step, Gaussian mixtures, [Gaussian Mixtures](#)

maximum a-posteriori (MAP) estimation, [Selecting the Number of Clusters](#)

maximum likelihood estimate (MLE), [Selecting the Number of Clusters](#)

MC (Monte Carlo) dropout regularization, [Monte Carlo \(MC\) Dropout](#)-
[Monte Carlo \(MC\) Dropout](#)

MCTS (Monte Carlo tree search), [Overview of Some Popular RL Algorithms](#)

MDPs (Markov decision processes), [Markov Decision Processes](#)-[Markov Decision Processes](#), [Exploration Policies](#)

MDS (multidimensional scaling), [Other Dimensionality Reduction Techniques](#)

mean absolute error (MAE), [Select a Performance Measure](#)

mean absolute percentage error (MAPE), [Forecasting a Time Series](#)

mean average precision (mAP), [You Only Look Once](#)

mean squared error (MSE), [Linear Regression](#), [Variational Autoencoders](#)

mean-shift, clustering algorithms, [Other Clustering Algorithms](#)

mean_squared_error(), [Train and Evaluate on the Training Set](#)

measure of similarity, [Instance-based learning](#)

memory bandwidth, GPU card, [Prefetching](#)

memory cells (cells), RNNs, [Memory Cells](#), [Tackling the Short-Term Memory Problem-WaveNet](#)

memory requirements, convolutional layers, [Memory Requirements](#)

Mercer's theorem, [Kernelized SVMs](#)

meta learner, [Stacking](#)

metagraphs, SavedModel, [Exporting SavedModels](#)

min-max scaling, [Feature Scaling and Transformation](#)

mini-batch discrimination, [The Difficulties of Training GANs](#)

mini-batch gradient descent, [Mini-Batch Gradient Descent-Mini-Batch Gradient Descent](#), [Early Stopping](#)

mini-batch k-means, [Accelerated k-means and mini-batch k-means](#)

mini-batches, [Online learning](#), [Progressive Growing of GANs](#)

MinMaxScaler, [Feature Scaling and Transformation](#)

mirrored strategy, data parallelism, [Data parallelism using the mirrored strategy](#), [Training at Scale Using the Distribution Strategies API](#), [Training a Model on a TensorFlow Cluster](#)

Mish activation function, [GELU](#), [Swish](#), and [Mish](#)

mixing regularization, StyleGAN, [StyleGANs](#)

ML (see machine learning)

ML Operations (MLOps), [Launch](#), [Monitor](#), and [Maintain Your System](#)

MLE (maximum likelihood estimate), [Selecting the Number of Clusters](#)

MLM (masked language model), [An Avalanche of Transformer Models](#)

MLPs (see multilayer perceptrons)

MNIST dataset, [MNIST-MNIST](#)

mobile device, deploying model to, [Deploying a Model to a Mobile or Embedded Device](#)-[Deploying a Model to a Mobile or Embedded Device](#)

mode collapse, [Vision Transformers](#), [The Difficulties of Training GANs](#)

model parallelism, [Model Parallelism](#)-[Model Parallelism](#)

model parameters, [Model-based learning](#) and a typical machine learning workflow

early stopping regularization, [Training and Cost Function](#)

in gradient descent, [Gradient Descent](#), [Batch Gradient Descent](#)

linear SVM classifier mechanics, [Under the Hood of Linear SVM Classifiers](#)

and variable updating, [Variables](#)

weight matrix shape, [Creating the model using the sequential API](#)

model rot, [Batch learning](#)

model selection, [Model-based learning](#) and a typical machine learning workflow, [Hyperparameter Tuning](#) and [Model Selection](#)-[Hyperparameter Tuning](#) and [Model Selection](#)

model server (see [TensorFlow Serving](#))

model warmup, [Deploying a new model version](#)

model-based learning, [Model-based learning](#) and a typical machine learning workflow-[Model-based learning](#) and a typical machine learning workflow

modes, [Feature Scaling and Transformation](#)

momentum optimization, [Momentum](#)

momentum β , Momentum

Monte Carlo (MC) dropout, Monte Carlo (MC) Dropout-Monte Carlo (MC) Dropout

Monte Carlo tree search (MCTS), Overview of Some Popular RL Algorithms

MSE (mean squared error), Linear Regression, Variational Autoencoders

multi-head attention layer, Attention Is All You Need: The Original Transformer Architecture, Multi-head attention-Multi-head attention

multi-hot encoding, The CategoryEncoding Layer

multiclass (multinomial) classification, Multiclass Classification-Multiclass Classification, Classification MLPs-Classification MLPs

multidimensional scaling (MDS), Other Dimensionality Reduction Techniques

multilabel classifiers, Multilabel Classification-Multilabel Classification

multilayer perceptrons (MLPs), Introduction to Artificial Neural Networks with Keras, The Perceptron-Using TensorBoard for Visualization

and autoencoders, Efficient Data Representations, Performing PCA with an Undercomplete Linear Autoencoder

and backpropagation, The Multilayer Perceptron and Backpropagation-The Multilayer Perceptron and Backpropagation

callbacks, Using Callbacks-Using Callbacks

classification MLPs, Classification MLPs-Classification MLPs

complex models, Building Complex Models Using the Functional API-Building Complex Models Using the Functional API

dynamic models, Using the Subclassing API to Build Dynamic Models-Using the Subclassing API to Build Dynamic Models

image classifier, [Building an Image Classifier Using the Sequential API](#)-
Using the model to make predictions

regression MLPs, [Regression MLPs-Regression MLPs](#), Building a
Regression MLP Using the Sequential API

saving and restoring a model, [Saving and Restoring a Model-Saving and Restoring a Model](#)

visualization with TensorBoard, [Using TensorBoard for Visualization-Using TensorBoard for Visualization](#)

multimodal distribution, [Feature Scaling and Transformation](#)

multimodal transformers, [Vision Transformers](#)

multinomial logistic regression, [Softmax Regression-Softmax Regression](#)

multioutput classifiers, [Multioutput Classification-Multioutput Classification](#)

multiple regression, [Frame the Problem](#)

multiplicative attention, [Attention Mechanisms](#)

multitask classification, [Building Complex Models Using the Functional API](#)

multivariate regression, [Frame the Problem](#)

multivariate time series, [Forecasting a Time Series](#), [Forecasting Multivariate Time Series-Forecasting Multivariate Time Series](#)

N

Nadam, [Nadam](#)

NAG (Nesterov accelerated gradient), [Nesterov Accelerated Gradient](#),
[Nadam](#)

naive forecasting, [Forecasting a Time Series](#)

Nash equilibrium, [The Difficulties of Training GANs](#)

natural language processing (NLP), [Natural Language Processing with RNNs](#) and [Attention-Hugging Face's Transformers Library](#)

char-RNN model to generate text, [Generating Shakespearean Text Using a Character RNN-Stateful RNN](#)

encoder–decoder network for machine translation, [An Encoder–Decoder Network for Neural Machine Translation-An Encoder–Decoder Network for Neural Machine Translation](#)

machine learning examples, [Examples of Applications](#)

sentiment analysis, [Sentiment Analysis-Reusing Pretrained Embeddings and Language Models](#)

text classification, [Sentiment Analysis-Reusing Pretrained Embeddings and Language Models](#)

text encoding, [Text Preprocessing-Using Pretrained Language Model Components](#)

text summarization, [Examples of Applications](#)

transformer models (see [transformer models](#))

word embeddings, [Encoding Categorical Features Using Embeddings](#)

natural language understanding (NLU), [Examples of Applications](#)

NCCL (Nvidia collective communications library), [Training at Scale Using the Distribution Strategies API](#)

NEAT (neuroevolution of augmenting topologies), [Policy Search](#)

negative class, [Confusion Matrices](#), [Logistic Regression](#)

nested dataset, [Preparing the Data for Machine Learning Models](#)

Nesterov accelerated gradient (NAG), [Nesterov Accelerated Gradient](#), [Nadam](#)

Nesterov momentum optimization, Nesterov Accelerated Gradient, Nadam

neural machine translation (NMT), Reusing Pretrained Embeddings and Language Models-Multi-head attention

and attention mechanisms, Attention Mechanisms-Multi-head attention with transformers, An Avalanche of Transformer Models-An Avalanche of Transformer Models

neural networks (see artificial neural networks)

neuroevolution of augmenting topologies (NEAT), Policy Search

next sentence prediction (NSP), An Avalanche of Transformer Models

NLU (natural language understanding), Examples of Applications

NMT (see neural machine translation)

No Free Lunch theorem, Data Mismatch

non-max suppression, bounding boxes, Object Detection

nonlinear dimensionality reduction (NLDR), LLE-LLE

nonlinear SVM classifiers, Nonlinear SVM Classification-SVM Classes and Computational Complexity

nonparametric models, Regularization Hyperparameters

nonrepresentative training data, Nonrepresentative Training Data

nonresponse bias, Nonrepresentative Training Data

normal distribution, The Vanishing/Exploding Gradients Problems, Glorot and He Initialization

Normal equation, The Normal Equation-The Normal Equation

normalization, Feature Scaling and Transformation, Batch Normalization-Implementing batch normalization with Keras, Fighting the Unstable

Gradients Problem

Normalization layer, [The Normalization Layer-The Normalization Layer](#)

normalized exponential (softmax function), [Softmax Regression](#)

notations, [Select a Performance Measure-Select a Performance Measure](#), [Linear Regression](#)

novelty detection, [Unsupervised learning](#), [Using Gaussian Mixtures for Anomaly Detection](#)

NP-Complete problem, CART as, [The CART Training Algorithm](#)

NSP (next sentence prediction), [An Avalanche of Transformer Models](#)

nucleus sampling, [Generating Fake Shakespearean Text](#)

null hypothesis, [Regularization Hyperparameters](#)

number of inputs, [Creating the model using the sequential API](#), Glorot and He Initialization

number of neurons per hidden layer, [Number of Neurons per Hidden Layer](#)

NumPy, [Running the Code Examples Using Google Colab](#)

NumPy arrays, [Clean the Data](#), [Custom Transformers](#), [Using TensorFlow like NumPy-Other Data Structures](#)

Nvidia collective communications library (NCCL), [Training at Scale Using the Distribution Strategies API](#)

Nvidia GPU card, [Getting Your Own GPU-Getting Your Own GPU](#)

O

OAuth 2.0, [Creating a Prediction Service on Vertex AI](#)

object detection, CNNs, [Object Detection-You Only Look Once](#)

object tracking, CNNs, Object Tracking

objectness score, Object Detection

observation space, reinforcement learning, Learning to Optimize Rewards, Neural Network Policies

observations, Introduction to OpenAI Gym-Introduction to OpenAI Gym

OCR (optical character recognition), The Machine Learning Landscape

OEL (open-ended learning), Overview of Some Popular RL Algorithms

off-policy algorithm, Q-Learning

offline learning, Batch learning

on-policy algorithm, Q-Learning

one-class SVM, Other Algorithms for Anomaly and Novelty Detection

one-hot encoding, Handling Text and Categorical Attributes-Handling Text and Categorical Attributes, The CategoryEncoding Layer, Encoding Categorical Features Using Embeddings

one-versus-all (OvA) strategy, Multiclass Classification-Multiclass Classification

one-versus-one (OvO) strategy, Multiclass Classification-Multiclass Classification

one-versus-the-rest (OvR) strategy, Multiclass Classification-Multiclass Classification

1cycle scheduling, Learning Rate Scheduling, Learning Rate Scheduling

1D convolutional layers, Semantic Segmentation, Using 1D convolutional layers to process sequences

OneHotEncoder, Handling Text and Categorical Attributes-Handling Text and Categorical Attributes, Feature Scaling and Transformation,

Transformation Pipelines

online kernelized SVMs, [Kernelized SVMs](#)

online learning, [Online learning](#)-[Online learning](#)

online model, DQN, [Fixed Q-value Targets](#)

OOB (out-of-bag) evaluation, [Out-of-Bag Evaluation](#)-[Out-of-Bag Evaluation](#)

open-ended learning (OEL), [Overview of Some Popular RL Algorithms](#)

OpenAI Gym, [Introduction to OpenAI Gym](#)-[Introduction to OpenAI Gym](#)

operations (ops) and tensors, [A Quick Tour of TensorFlow](#), [Tensors and Operations](#)-[Tensors and Operations](#)

optical character recognition (OCR), [The Machine Learning Landscape](#)

optimal state value, MDP, [Markov Decision Processes](#)

optimizers, [Faster Optimizers](#)-[AdamW](#)

AdaGrad, [AdaGrad](#)

Adam optimization, [Adam](#)

AdaMax, [AdaMax](#)

AdamW, [AdamW](#)

hyperparameters, [Learning Rate](#), [Batch Size](#), and [Other Hyperparameters](#)

momentum optimization, [Momentum](#)

Nadam, [Nadam](#)

Nesterov accelerated gradient, [Nesterov Accelerated Gradient](#)

output layer, [An Encoder–Decoder Network for Neural Machine Translation](#)

RMSProp, [RMSProp](#)

oracle, [Fine-Tuning Neural Network Hyperparameters](#)

order of integration (d) hyperparameter, [The ARMA Model Family](#)

OrdinalEncoder, [Handling Text and Categorical Attributes](#)

orthogonal matrix, [An Encoder–Decoder Network for Neural Machine Translation](#)

out-of-bag (OOB) evaluation, [Out-of-Bag Evaluation-Out-of-Bag Evaluation](#)

out-of-core learning, [Online learning](#)

out-of-sample error, [Testing and Validating](#)

outlier detection (see anomaly detection)

outliers, [Unsupervised Learning Techniques](#)

output gate, LSTM, [LSTM cells](#)

output layer, neural network, [An Encoder–Decoder Network for Neural Machine Translation](#)

OvA (one-versus-all) strategy, [Multiclass Classification-Multiclass Classification](#)

overcomplete autoencoder, [Convolutional Autoencoders](#)

overfitting of data, [Overfitting the Training Data-Overfitting the Training Data, Create a Test Set](#)

avoiding through regularization, [Avoiding Overfitting Through Regularization-Max-Norm Regularization](#)

and decision trees, [Regularization Hyperparameters, Regression](#)

and dropout regularization, [Dropout](#)

gamma (γ) hyperparameter to adjust, [Gaussian RBF Kernel](#)

image classification, [Training and evaluating the model](#)
learning curves to assess, [Learning Curves-Learning Curves](#)
number of neurons per hidden layer, [Number of Neurons per Hidden Layer](#)
polynomial regression, [Training Models](#)
SVM model, [Soft Margin Classification](#)
OvO (one-versus-one) strategy, [Multiclass Classification-Multiclass Classification](#)
OvR (one-versus-the-rest) strategy, [Multiclass Classification-Multiclass Classification](#)

P

p-value, [Regularization Hyperparameters](#)
PACF (partial autocorrelation function), [The ARMA Model Family](#)
padding options, convolutional layer, [Implementing Convolutional Layers with Keras-Implementing Convolutional Layers with Keras](#)
PaLM (Pathways language model), [An Avalanche of Transformer Models](#)
Pandas, [Running the Code Examples Using Google Colab](#), [Look for Correlations-Look for Correlations](#), [Handling Text and Categorical Attributes](#), [Feature Scaling and Transformation](#)
parallelism, training models across devices, [Training Models Across Multiple Devices-Hyperparameter Tuning on Vertex AI](#)
data parallelism, [Data Parallelism-Bandwidth saturation](#)
distribution strategies API, [Training at Scale Using the Distribution Strategies API](#)

with GPU, [Parallel Execution Across Multiple Devices](#)-[Parallel Execution Across Multiple Devices](#)

hyperparameter tuning, [Hyperparameter Tuning on Vertex AI](#)-
[Hyperparameter Tuning on Vertex AI](#)

model parallelism, [Model Parallelism](#)-[Model Parallelism](#)

on TensorFlow cluster, [Training a Model on a TensorFlow Cluster](#)-
[Training a Model on a TensorFlow Cluster](#)

Vertex AI for running large jobs, [Running Large Training Jobs on Vertex AI](#)-
[Running Large Training Jobs on Vertex AI](#)

parameter efficiency, [Number of Hidden Layers](#)

parameter matrix, [Softmax Regression](#)

parameter servers, [Data parallelism with centralized parameters](#)

parameter space, [Gradient Descent](#)

parameter vector, [Linear Regression](#), [Gradient Descent](#), [Training and Cost Function](#), [Softmax Regression](#)

parametric leaky ReLU (PReLU), [Leaky ReLU](#)

parametric models, [Regularization](#) [Hyperparameters](#)

partial autocorrelation function (PACF), [The ARMA Model Family](#)

partial derivative, [Batch Gradient Descent](#)

partial_fit(), [Stochastic Gradient Descent](#)

Pathways language model (PaLM), [An Avalanche of Transformer Models](#)

PCA (see principal component analysis)

PDF (probability density function), [Unsupervised Learning Techniques](#),
[Selecting the Number of Clusters](#)

Pearson's r, [Look for Correlations](#)

penalties, reinforcement learning, [Reinforcement learning](#)

PER (prioritized experience replay), [Prioritized Experience Replay](#)

Perceiver, [Vision Transformers](#)

percentiles, [Take a Quick Look at the Data Structure](#)

perceptrons, [Introduction to Artificial Neural Networks with Keras, The Perceptron-Classification MLPs](#)

(see also multilayer perceptrons)

performance measures, [Performance Measures-The ROC Curve](#)

confusion matrix, [Confusion Matrices-Confusion Matrices](#)

cross-validation to measure accuracy, [Measuring Accuracy Using Cross-Validation-Measuring Accuracy Using Cross-Validation](#)

precision and recall, [Precision and Recall-The Precision/Recall Trade-off](#)

ROC curve, [The ROC Curve-The ROC Curve](#)

selecting, [Select a Performance Measure-Select a Performance Measure](#)

performance scheduling, [Learning Rate Scheduling, Learning Rate Scheduling](#)

permutation(), [Create a Test Set](#)

PG (policy gradients) algorithm, [Policy Search, Policy Gradients-Policy Gradients](#)

piecewise constant scheduling, [Learning Rate Scheduling, Learning Rate Scheduling](#)

PipeDream, [Bandwidth saturation](#)

Pipeline class, Transformation Pipelines

Pipeline constructor, Transformation Pipelines-Transformation Pipelines

pipelines, Frame the Problem, Transformation Pipelines-Transformation Pipelines, Learning Curves, Soft Margin Classification

pixelwise normalization layer, Progressive Growing of GANs

placeholders, function definitions, Exploring Function Definitions and Graphs

POET algorithm, Overview of Some Popular RL Algorithms

policy gradients (PG) algorithm, Policy Search, Policy Gradients-Policy Gradients

policy parameters, Policy Search

policy space, Policy Search

policy, reinforcement learning, Reinforcement learning, Policy Search, Neural Network Policies

polynomial features, SVM classifiers, Polynomial Kernel

polynomial kernel, Polynomial Kernel, Kernelized SVMs

polynomial regression, Training Models, Polynomial Regression-Polynomial Regression

polynomial time, The CART Training Algorithm

PolynomialFeatures, Polynomial Regression, Nonlinear SVM Classification

pooling kernel, Pooling Layers

pooling layers, Pooling Layers-Implementing Pooling Layers with Keras

positional encodings, Positional encodings-Positional encodings

positive class, Confusion Matrices, Logistic Regression

post-training quantization, [Deploying a Model to a Mobile or Embedded Device](#)

posterior distribution, [Variational Autoencoders](#)

power law distribution, [Feature Scaling and Transformation](#)

power scheduling, [Learning Rate Scheduling](#)

PPO (proximal policy optimization), [Overview of Some Popular RL Algorithms](#)

precision and recall, classifier metrics, [Precision and Recall-The Precision/Recall Trade-off](#)

precision/recall curve (PR), [The Precision/Recall Trade-off](#), [The ROC Curve](#)

precision/recall trade-off, [The Precision/Recall Trade-off-The Precision/Recall Trade-off](#)

`predict()`, [Clean the Data](#), [Custom Transformers](#), [Transformation Pipelines](#)

predicted class, [Confusion Matrices](#)

prediction service, on Vertex AI, [Creating a Prediction Service on Vertex AI-Running Batch Prediction Jobs on Vertex AI](#)

predictions

backpropagation, [The Multilayer Perceptron and Backpropagation](#)

confusion matrix, [Confusion Matrices](#)-[Confusion Matrices](#)

cross-validation to measure accuracy, [Measuring Accuracy Using Cross-Validation](#)-[Measuring Accuracy Using Cross-Validation](#)

decision trees, [Making Predictions-The CART Training Algorithm](#)

with linear SVM classifier, [Under the Hood of Linear SVM Classifiers](#)

predictors, [Supervised learning](#)

(see also ensemble learning)

`predict_log_proba()`, Soft Margin Classification

`predict_proba()`, Soft Margin Classification

prefetching of data, Prefetching-Prefetching

PReLU (parametric leaky ReLU), Leaky ReLU

preprocessed attributes, Take a Quick Look at the Data Structure

preprocessing data (see loading and preprocessing data)

preprocessing mismatch, The Normalization Layer

pretraining and pretrained layers

on auxiliary task, Pretraining on an Auxiliary Task

CNNs, Using Pretrained Models from Keras-Pretrained Models for Transfer Learning

greedy layer-wise pretraining, Unsupervised Pretraining, Training One Autoencoder at a Time, Progressive Growing of GANs

language model components, Using Pretrained Language Model Components

reusing embeddings, Reusing Pretrained Embeddings and Language Models-Reusing Pretrained Embeddings and Language Models

reusing layers, Reusing Pretrained Layers-Pretraining on an Auxiliary Task

in unsupervised learning, Unsupervised Pretraining, Reusing Pretrained Embeddings and Language Models, An Avalanche of Transformer Models, Unsupervised Pretraining Using Stacked Autoencoders

primal problem, The Dual Problem

principal component (PC), [Principal Components-Principal Components](#)

principal component analysis (PCA), [PCA-Incremental PCA](#)

choosing number of dimensions, [Choosing the Right Number of Dimensions](#)-[Choosing the Right Number of Dimensions](#)

for compression, [PCA for Compression](#)-[PCA for Compression](#)

explained variance ratio, [Explained Variance Ratio](#)

finding principal components, [Principal Components](#)

incremental PCA, [Incremental PCA](#)-[Incremental PCA](#)

preserving variance, [Preserving the Variance](#)

projecting down to d dimensions, [Projecting Down to d Dimensions](#)

randomized PCA, [Randomized PCA](#)

for scaling data in decision trees, [Sensitivity to Axis Orientation](#)

with undercomplete linear autoencoder, [Performing PCA with an Undercomplete Linear Autoencoder](#)-[Performing PCA with an Undercomplete Linear Autoencoder](#)

using Scikit_Learn for, [Using Scikit-Learn](#)

prior distribution, [Variational Autoencoders](#)

prioritized experience replay (PER), [Prioritized Experience Replay](#)

probabilistic autoencoders, [Variational Autoencoders](#)

probabilities, estimating, [Estimating Probabilities](#)-[Estimating Probabilities](#), [Estimating Class Probabilities](#), [Voting Classifiers](#)

probability density function (PDF), [Unsupervised Learning Techniques](#), [Selecting the Number of Clusters](#)

probability versus likelihood, [Selecting the Number of Clusters](#)-[Selecting the](#)

Number of Clusters

profiling the network, with TensorBoard, [Using TensorBoard for Visualization](#)

progressive web app (PWA), [Running a Model in a Web Page](#)

projection, dimensionality reduction, [Projection-Projection](#)

propositional logic, [From Biological to Artificial Neurons](#)

protocol buffers (protobuf), [A Brief Introduction to Protocol Buffers](#)-
[TensorFlow Protobufs](#), [Handling Lists of Lists Using the SequenceExample](#)
[Protobuf](#), [Querying TF Serving through the gRPC API](#)

proximal policy optimization (PPO), [Overview of Some Popular RL Algorithms](#)

pruning of decision tree nodes, [Regularization Hyperparameters](#)

pseudoinverse, [The Normal Equation](#)

PWA (progressive web app), [Running a Model in a Web Page](#)

Python API, [Get the Data](#)

Q

Q-learning algorithm, [Q-Learning-Dueling DQN](#)

approximate Q-learning, [Approximate Q-Learning and Deep Q-Learning](#)

exploration policies, [Exploration Policies](#)

implementing deep Q-learning, [Implementing Deep Q-Learning](#)-
[Implementing Deep Q-Learning](#)

variants in deep Q-learning, [Deep Q-Learning Variants-Dueling DQN](#)

Q-value iteration algorithm, [Markov Decision Processes](#)-[Markov Decision](#)

Processes

Q-values, [Markov Decision Processes](#)-[Markov Decision Processes](#)

quadratic equation, [Polynomial Regression](#)

quadratic programming (QP) problems, [Under the Hood of Linear SVM Classifiers](#)

quantization-aware training, [Deploying a Model to a Mobile or Embedded Device](#)

quartiles, [Take a Quick Look at the Data Structure](#)

queries per second (QPS), [Training and Deploying TensorFlow Models at Scale](#), [Creating a Prediction Service on Vertex AI](#)

question-answering modules, [Examples of Applications](#)

queues, [Other Data Structures](#), [Queues](#)

R

radial basis function (RBF), [Feature Scaling and Transformation](#)

ragged dimensions, [Other Data Structures](#)

Rainbow agent, [Dueling DQN](#)

random forests, [Better Evaluation Using Cross-Validation](#), [Ensemble Learning and Random Forests](#), [Random Forests](#)-[Feature Importance](#)

analysis of models and their errors, [Analyzing the Best Models and Their Errors](#)

decision trees (see decision trees)

extra-trees, [Extra-Trees](#)

feature importance measurement, [Feature Importance](#)

random initialization, [Gradient Descent](#), [Gradient Descent](#)
random patches, [Random Patches and Random Subspaces](#)
random projection algorithm, [Random Projection](#)-[Random Projection](#)
random subspaces, [Random Patches and Random Subspaces](#)
`RandomForestClassifier`, [The ROC Curve](#)-[The ROC Curve](#)
`RandomForestRegressor`, [Better Evaluation Using Cross-Validation](#)
randomized leaky ReLU (RReLU), [Leaky ReLU](#)
randomized PCA, [Randomized PCA](#)
randomized search, [Randomized Search](#)-[Randomized Search](#), [Fine-Tuning Neural Network Hyperparameters](#)-[Fine-Tuning Neural Network Hyperparameters](#)
RBF (radial basis function), [Feature Scaling and Transformation](#)
`rbf_kernel()`, [Feature Scaling and Transformation](#), [Custom Transformers](#)
recall metric, [Precision and Recall](#)
receiver operating characteristic (ROC) curve, [The ROC Curve](#)-[The ROC Curve](#)
recognition network, [Efficient Data Representations](#)
(see also autoencoders)
recommender system, [Examples of Applications](#)
reconstruction error, [PCA for Compression](#), [Tying Weights](#)
reconstruction loss, [Losses and Metrics Based on Model Internals](#), [Efficient Data Representations](#)
rectified linear units (ReLU) (see ReLU)

recurrent dropout, [Processing Sequences Using RNNs and CNNs](#)

recurrent layer normalization, [Processing Sequences Using RNNs and CNNs](#),
[Fighting the Unstable Gradients Problem](#)

recurrent neural networks (RNNs), [Processing Sequences Using RNNs and CNNs](#)-[Beam Search](#)

bidirectional, [Bidirectional RNNs](#)

deep RNN, [Forecasting Using a Deep RNN](#)

forecasting time series (see time series data)

gradient clipping, [Gradient Clipping](#)

handling long sequences, [Handling Long Sequences-WaveNet](#)

input and output sequences, [Input and Output Sequences](#)-[Input and Output Sequences](#)

memory cells, [Memory Cells](#), [Tackling the Short-Term Memory Problem-WaveNet](#)

NLP (see natural language processing)

and Perceiver, [Vision Transformers](#)

splitting across devices, [Model Parallelism](#)

stateful, [Natural Language Processing with RNNs and Attention](#),
[Stateful RNN](#)-[Stateful RNN](#)

stateless, [Natural Language Processing with RNNs and Attention](#),
[Stateful RNN](#)

training, [Training RNNs](#)

and vision transformers, [Vision Transformers](#)

recurrent neurons, [Recurrent Neurons and Layers](#)

reduce operation, [Data parallelism using the mirrored strategy](#)
region proposal network (RPN), [You Only Look Once](#)
regression MLPs, [Regression MLPs](#)-[Regression MLPs](#)
regression models
and classification, [Supervised learning](#), [Multioutput Classification](#)
decision tree tasks, [Regression](#)-[Regression](#)
forecasting example, [Examples of Applications](#)
lasso regression, [Lasso Regression](#)-[Lasso Regression](#)
linear regression (see linear regression)
logistic regression (see logistic regression)
multiple regression, [Frame the Problem](#)
multivariate regression, [Frame the Problem](#)
polynomial regression, [Training Models](#), [Polynomial Regression](#)-[Polynomial Regression](#)
regression MLPs, [Building a Regression MLP Using the Sequential API](#)
ridge regression, [Ridge Regression](#)-[Ridge Regression](#), [Elastic Net](#)
[Regression](#)
softmax regression, [Softmax Regression](#)-[Softmax Regression](#)
SVM, [SVM Regression](#)-[SVM Regression](#)
univariate regression, [Frame the Problem](#)
regression to the mean, [Supervised learning](#)
regularization, [Overfitting the Training Data](#), [Avoiding Overfitting Through](#)
[Regularization](#)-[Max-Norm Regularization](#)

custom regularizers, [Custom Activation Functions, Initializers, Regularizers, and Constraints](#)
decision trees, [Regularization Hyperparameters](#)
dropout, [Dropout-Dropout](#)
early stopping, [Early Stopping-Early Stopping](#), [Gradient Boosting](#),
[Forecasting Using a Linear Model](#)
elastic net, [Elastic Net Regression](#)
hyperparameters, [Regularization Hyperparameters-Regularization Hyperparameters](#)
lasso regression, [Lasso Regression-Lasso Regression](#)
linear models, [Regularized Linear Models-Early Stopping](#)
max-norm, [Max-Norm Regularization](#)
MC dropout, [Monte Carlo \(MC\) Dropout-Monte Carlo \(MC\) Dropout](#)
ridge, [Ridge Regression](#)
shrinkage, [Gradient Boosting](#)
subword, [Sentiment Analysis](#)
Tikhonov, [Ridge Regression-Ridge Regression](#)
weight decay, [AdamW](#)
 ℓ_1 and ℓ_2 regularization, [\$\ell_1\$ and \$\ell_2\$ Regularization](#)
REINFORCE algorithms, [Policy Gradients](#)
reinforcement learning (RL), [Reinforcement learning](#), [Reinforcement Learning-Overview of Some Popular RL Algorithms](#)
actions, [Evaluating Actions: The Credit Assignment Problem-Evaluating Actions: The Credit Assignment Problem](#)

credit assignment problem, [Evaluating Actions: The Credit Assignment Problem](#)-[Evaluating Actions: The Credit Assignment Problem](#)

examples of, [Examples of Applications](#), [Learning to Optimize Rewards](#)
learning in order to optimizing rewards, [Learning to Optimize Rewards](#)

Markov decision processes, [Markov Decision Processes](#)-[Markov Decision Processes](#)

neural network policies, [Neural Network Policies](#)

OpenAI Gym, [Introduction to OpenAI Gym](#)-[Introduction to OpenAI Gym](#)

PG algorithms, [Policy Gradients](#)-[Policy Gradients](#)

policy search, [Policy Search](#)

Q-learning, [Q-Learning](#)-[Dueling DQN](#)

TD learning, [Temporal Difference Learning](#)

ReLU (rectified linear units)

and backpropagation, [The Multilayer Perceptron and Backpropagation](#)

in CNN architectures, [CNN Architectures](#)

as default for simple tasks, [GELU](#), [Swish](#), and [Mish](#)

leakyReLU, [Leaky ReLU](#)-[Leaky ReLU](#)

and MLPs, [Regression MLPs](#)

RNN unstable gradients problem, [Fighting the Unstable Gradients Problem](#)

RReLU, [Leaky ReLU](#)

tuning hyperparameters, [Fine-Tuning Neural Network Hyperparameters](#)

replay buffer, [Implementing Deep Q-Learning](#)
representation learning, [Handling Text and Categorical Attributes](#)
(see also autoencoders)

residual block, [Custom Models-Custom Models](#)

residual errors, [Gradient Boosting-Gradient Boosting](#)

residual learning, [ResNet](#)

residual network (ResNet), [ResNet-ResNet](#)

residual units, [ResNet](#)

ResNet-152, [ResNet](#)

ResNet-34, [ResNet](#), [Implementing a ResNet-34 CNN Using Keras](#)

ResNet-50, [Using Pretrained Models from Keras](#)

ResNeXt, [Other Noteworthy Architectures](#)

REST API, querying through, [Querying TF Serving through the REST API](#)

return, in reinforcement learning, [Policy Gradients](#)

reverse process, diffusion model, [Diffusion Models-Diffusion Models](#)

reverse-mode autodiff, [The Multilayer Perceptron and Backpropagation](#),
[Computing Gradients Using Autodiff](#)

rewards, reinforcement learning, [Reinforcement learning](#), [Learning to Optimize Rewards](#)

Ridge, [Ridge Regression](#)

ridge regression, [Ridge Regression-Ridge Regression](#), [Elastic Net Regression](#)

ridge regularization, [Ridge Regression](#)

RidgeCV, [Ridge Regression](#)

RL (see reinforcement learning)

RMSProp, [RMSProp](#)

ROC (receiver operating characteristic) curve, [The ROC Curve-The ROC Curve](#)

root mean square error (RMSE), [Select a Performance Measure-Select a Performance Measure](#), [Train and Evaluate on the Training Set](#), [Linear Regression](#), [Early Stopping](#)

root node, decision tree, [Making Predictions](#), [Making Predictions](#)

RPN (region proposal network), [You Only Look Once](#)

RReLU (randomized leaky ReLU), [Leaky ReLU](#)

S

SAC (soft actor-critic), [Overview of Some Popular RL Algorithms](#)

“same” padding, computer vision, [Implementing Convolutional Layers with Keras](#)

SAMME, [AdaBoost](#)

sample inefficiency, [Policy Gradients](#)

sampled softmax, [An Encoder–Decoder Network for Neural Machine Translation](#)

sampling bias, [Nonrepresentative Training Data](#), [Create a Test Set](#)

sampling noise, [Nonrepresentative Training Data](#)

SARIMA model, [The ARMA Model Family-The ARMA Model Family](#)

SavedModel, [Exporting SavedModels-Exporting SavedModels](#)

saving, loading, and restoring models, [Saving and Restoring a Model-Saving and Restoring a Model](#), [Saving and Loading Models That Contain Custom](#)

Components-Saving and Loading Models That Contain Custom Components

scaled dot-product attention layer, Multi-head attention

scatter matrix, Look for Correlations-Look for Correlations

Scikit-Learn, Objective and Approach

bagging and pasting in, Bagging and Pasting in Scikit-Learn

CART algorithm, The CART Training Algorithm, Regression

cross-validation, Better Evaluation Using Cross-Validation-Better Evaluation Using Cross-Validation

design principles, Clean the Data-Clean the Data

PCA implementation, Using Scikit-Learn

Pipeline constructor, Transformation Pipelines-Transformation Pipelines

sklearn.base.BaseEstimator, Custom Transformers

sklearn.base.clone(), Early Stopping

sklearn.base.TransformerMixin, Custom Transformers

sklearn.cluster.DBSCAN, DBSCAN

sklearn.cluster.KMeans, Custom Transformers, k-means

sklearn.cluster.MiniBatchKMeans, Accelerated k-means and mini-batch k-means

sklearn.compose.ColumnTransformer, Transformation Pipelines

sklearn.compose.TransformedTargetRegressor, Feature Scaling and Transformation

sklearn.datasets.load_iris(), Decision Boundaries

sklearn.datasets.make_moons(), Nonlinear SVM Classification

`sklearn.decomposition.IncrementalPCA`, [Incremental PCA](#)

`sklearn.decomposition.PCA`, [Using Scikit-Learn](#)

`sklearn.ensemble.AdaBoostClassifier`, [AdaBoost](#)

`sklearn.ensemble.BaggingClassifier`, [Bagging and Pasting in Scikit-Learn](#)

`sklearn.ensemble.GradientBoostingRegressor`, [Gradient Boosting-Gradient Boosting](#)

`sklearn.ensemble.HistGradientBoostingClassifier`, [Histogram-Based Gradient Boosting](#)

`sklearn.ensemble.HistGradientBoostingRegressor`, [Histogram-Based Gradient Boosting](#)

`sklearn.ensemble.RandomForestClassifier`, [The ROC Curve-The ROC Curve, Voting Classifiers, Random Forests, Feature Importance, Choosing the Right Number of Dimensions](#)

`sklearn.ensemble.RandomForestRegressor`, [Better Evaluation Using Cross-Validation, Random Forests](#)

`sklearn.ensemble.StackingClassifier`, [Stacking](#)

`sklearn.ensemble.StackingRegressor`, [Stacking](#)

`sklearn.ensemble.VotingClassifier`, [Voting Classifiers](#)

`sklearn.externals.joblib`, [Launch, Monitor, and Maintain Your System-Launch, Monitor, and Maintain Your System](#)

`sklearn.feature_selection.SelectFromModel`, [Analyzing the Best Models and Their Errors](#)

`sklearn.impute.IterativeImputer`, [Clean the Data](#)

`sklearn.impute.KNNImputer`, [Clean the Data](#)

sklearn.impute.SimpleImputer, [Clean the Data](#)

sklearn.linear_model.ElasticNet, [Elastic Net Regression](#)

sklearn.linear_model.Lasso, [Lasso Regression](#)

sklearn.linear_model.LinearRegression, [Model-based learning and a typical machine learning workflow](#), [Clean the Data](#), [Feature Scaling and Transformation](#), [The Normal Equation](#), [Computational Complexity](#), [Polynomial Regression](#)

sklearn.linear_model.LogisticRegression, [Decision Boundaries](#), [Softmax Regression](#), [Using Clustering for Semi-Supervised Learning](#)

sklearn.linear_model.Perceptron, [The Perceptron](#)

sklearn.linear_model.Ridge, [Ridge Regression](#)

sklearn.linear_model.RidgeCV, [Ridge Regression](#)

sklearn.linear_model.SGDClassifier, [Training a Binary Classifier](#), [The Precision/Recall Trade-off](#), [The Precision/Recall Trade-off](#), [The ROC Curve-The ROC Curve](#), [Multiclass Classification](#), [SVM Classes and Computational Complexity](#), [Under the Hood of Linear SVM Classifiers](#), [The Perceptron](#)

sklearn.linear_model.SGDRegressor, [Stochastic Gradient Descent](#), [Early Stopping](#)

sklearn.manifold.LocallyLinearEmbedding, [LLE](#)

sklearn.metrics.ConfusionMatrixDisplay, [Error Analysis](#)

sklearn.metrics.confusion_matrix(), [Confusion Matrices](#), [Error Analysis](#)

sklearn.metrics.f1_score(), [Precision and Recall](#), [Multilabel Classification](#)

sklearn.metrics.mean_squared_error(), [Train and Evaluate on the Training Set](#)

`sklearn.metrics.precision_recall_curve()`, [The Precision/Recall Trade-off](#), [The ROC Curve](#)

`sklearn.metrics.precision_score()`, [Precision and Recall](#)

`sklearn.metrics.recall_score()`, [Precision and Recall](#)

`sklearn.metrics.roc_auc_score()`, [The ROC Curve](#)

`sklearn.metrics.roc_curve()`, [The ROC Curve](#)

`sklearn.metrics.silhouette_score()`, [Finding the optimal number of clusters](#)

`sklearn.mixture.BayesianGaussianMixture`, [Bayesian Gaussian Mixture Models](#)

`sklearn.mixture.GaussianMixture`, [Gaussian Mixtures](#)

`sklearn.model_selection.cross_val_predict()`, [Confusion Matrices](#), [The Precision/Recall Trade-off](#), [The ROC Curve](#), [Error Analysis](#), [Stacking](#)

`sklearn.model_selection.cross_val_score()`, [Better Evaluation Using Cross-Validation](#), [Measuring Accuracy Using Cross-Validation](#)

`sklearn.model_selection.GridSearchCV`, [Grid Search-Grid Search](#)

`sklearn.model_selection.learning_curve()`, [Learning Curves](#)

`sklearn.model_selection.RandomizedSearchCV`, [Choosing the Right Number of Dimensions](#)

`sklearn.model_selection.StratifiedKFold`, [Measuring Accuracy Using Cross-Validation](#)

`sklearn.model_selection.StratifiedShuffleSplit`, [Create a Test Set](#)

`sklearn.model_selection.train_test_split()`, [Create a Test Set](#), [Create a Test Set](#), [Better Evaluation Using Cross-Validation](#)

`sklearn.multiclass.OneVsOneClassifier`, [Multiclass Classification](#)

sklearn.multiclass.OneVsRestClassifier, [Multiclass Classification](#)

sklearn.multioutput.ChainClassifier, [Multilabel Classification](#)

sklearn.neighbors.KNeighborsClassifier, [Multilabel Classification](#),
[Multioutput Classification](#), [DBSCAN](#)

sklearn.neighbors.KNeighborsRegressor, [Model-based learning and a typical machine learning workflow](#)

sklearn.neural_network.MLPClassifier, [Classification MLPs](#)

sklearn.neural_network.MLPRegressor, [Regression MLPs](#)

sklearn.pipeline.make_pipeline(), [Learning Curves](#)

sklearn.pipeline.Pipeline, [Transformation Pipelines](#)

sklearn.preprocessing.FunctionTransformer, [Custom Transformers](#)

sklearn.preprocessing.MinMaxScaler, [Feature Scaling and Transformation](#)

sklearn.preprocessing.OneHotEncoder, [Handling Text and Categorical Attributes](#)-[Handling Text and Categorical Attributes](#), [Feature Scaling and Transformation](#), [Transformation Pipelines](#)

sklearn.preprocessing.OrdinalEncoder, [Handling Text and Categorical Attributes](#), [Histogram-Based Gradient Boosting](#)

sklearn.preprocessing.PolynomialFeatures, [Polynomial Regression](#), [Nonlinear SVM Classification](#)

sklearn.preprocessing.StandardScaler, [Feature Scaling and Transformation](#), [Gradient Descent](#), [Ridge Regression](#), [Nonlinear SVM Classification](#)

sklearn.random_projection.GaussianRandomProjection, [Random Projection](#)

`sklearn.random_projection.SparseRandomProjection`, **Random Projection**

`sklearn.semi_supervised.LabelPropagation`, **Using Clustering for Semi-Supervised Learning**

`sklearn.semi_supervised.LabelSpreading`, **Using Clustering for Semi-Supervised Learning**

`sklearn.semi_supervised.SelfTrainingClassifier`, **Using Clustering for Semi-Supervised Learning**

`sklearn.svm.LinearSVC`, **Soft Margin Classification, SVM Classes and Computational Complexity, Under the Hood of Linear SVM Classifiers**

`sklearn.svm.SVC`, **Multiclass Classification, Polynomial Kernel, SVM Classes and Computational Complexity, Under the Hood of Linear SVM Classifiers**

`sklearn.svm.SVR`, **SVM Regression**

`sklearn.tree.DecisionTreeClassifier`, **Training and Visualizing a Decision Tree, Gini Impurity or Entropy?, Regularization Hyperparameters, Sensitivity to Axis Orientation, Random Forests**

`sklearn.tree.DecisionTreeRegressor`, **Train and Evaluate on the Training Set, Decision Trees, Regression, Gradient Boosting**

`sklearn.tree.export_graphviz()`, **Training and Visualizing a Decision Tree**

`sklearn.tree.ExtraTreesClassifier`, **Extra-Trees**

`sklearn.utils.estimator_checks`, **Custom Transformers**

`sklearn.utils.validation module`, **Custom Transformers**

SVM classification classes, SVM Classes and Computational Complexity

`score()`, **Clean the Data**

search engines, clustering for, [Clustering Algorithms: k-means and DBSCAN](#)

search space, [Randomized Search](#)

seasonality, time series modeling, [Forecasting a Time Series](#)

second moment (variance of gradient), [Adam](#)

second-order partial derivatives (Hessians), [AdamW](#)

segment embedding, [An Avalanche of Transformer Models](#)

`SelectFromModel`, [Analyzing the Best Models and Their Errors](#)

self-attention layers, [Attention Is All You Need: The Original Transformer Architecture](#)

Multi-head attention, [Vision Transformers](#)

self-distillation, [Vision Transformers](#)

self-normalization, [ELU and SELU](#), [Dropout](#), [Summary and Practical Guidelines](#)

self-supervised learning, [Self-supervised learning](#)-[Self-supervised learning](#)

`SELU` (scaled ELU) activation function, [ELU and SELU](#), [Dropout](#)

semantic interpolation, [Generating Fashion MNIST Images](#)

semantic segmentation, [Examples of Applications](#), [Using Clustering for Image Segmentation](#), [Semantic Segmentation](#)-[Semantic Segmentation](#)

semi-supervised learning, [Semi-supervised learning](#), [Clustering Algorithms: k-means and DBSCAN](#), [Using Clustering for Semi-Supervised Learning](#)-[Using Clustering for Semi-Supervised Learning](#)

`SENet`, [SENet-SENet](#)

sensitivity (recall), [ROC curve](#), [The ROC Curve](#)

sensitivity metric, [Confusion Matrices](#)

sensors, [Learning to Optimize Rewards](#)

sentence encoder, [Using Pretrained Language Model Components, Reusing Pretrained Embeddings and Language Models](#)

SentencePiece project, [Sentiment Analysis](#)

sentiment analysis, [Sentiment Analysis-Reusing Pretrained Embeddings and Language Models](#)

sentiment neuron, [Stateful RNN](#)

separable convolution layer, [Xception](#)

sequence length, [Using 1D convolutional layers to process sequences, Positional encodings](#)

sequence-to-sequence (seq2seq) network, [Input and Output Sequences, Forecasting Using a Sequence-to-Sequence Model-Forecasting Using a Sequence-to-Sequence Model](#)

sequence-to-vector network, [Input and Output Sequences](#)

SequenceExample protobuf, [Handling Lists of Lists Using the SequenceExample Protobuf](#)

sequential API, image classifier with, [Building an Image Classifier Using the Sequential API-Using the model to make predictions](#)

service account, GCP, [Creating a Prediction Service on Vertex AI](#)

service worker, [Running a Model in a Web Page](#)

sets, [Sets](#)

set_params(), [Custom Transformers](#)

SGD (see stochastic gradient descent)

SGDClassifier, [Training a Binary Classifier, The Precision/Recall Trade-off, The Precision/Recall Trade-off, The ROC Curve-The ROC Curve, Multiclass Classification, SVM Classes and Computational Complexity, Under the](#)

Hood of Linear SVM Classifiers, Under the Hood of Linear SVM Classifiers

SGDRegressor, Stochastic Gradient Descent, Early Stopping

sharpening, NLP transformers, Vision Transformers

shrinkage, Gradient Boosting

shuffle_and_split_data(), Create a Test Set, Create a Test Set

shuffling data, Stochastic Gradient Descent, Shuffling the Data-Shuffling the Data

Siamese neural network, Parallel Execution Across Multiple Devices

sigmoid activation function, Estimating Probabilities, The Multilayer Perceptron and Backpropagation, The Vanishing/Exploding Gradients Problems, Sparse Autoencoders

signals, Biological Neurons

silhouette coefficient, Finding the optimal number of clusters

silhouette diagram, Finding the optimal number of clusters

SiLU activation function, GELU, Swish, and Mish

similarity features, SVM, Similarity Features

SimpleImputer, Clean the Data

simulated annealing, Stochastic Gradient Descent

simulated environments, Introduction to OpenAI Gym-Introduction to OpenAI Gym

single program, multiple data (SPMD), Data Parallelism-Bandwidth saturation

single-shot learning, Semantic Segmentation

singular value decomposition (SVD), The Normal Equation, Principal

Components, Randomized PCA

skewed datasets, Measuring Accuracy Using Cross-Validation

skewed left/right, Take a Quick Look at the Data Structure

skip connections, ELU and SELU, ResNet

slack variable, Under the Hood of Linear SVM Classifiers

smoothing terms, Batch Normalization, AdaGrad

SMOTE (synthetic minority oversampling technique), AlexNet

soft actor-critic (SAC), Overview of Some Popular RL Algorithms

soft clustering, k-means

soft margin classification, Soft Margin Classification-Soft Margin Classification, Under the Hood of Linear SVM Classifiers

soft voting, Voting Classifiers

softmax activation function, Softmax Regression, Classification MLPs, Creating the model using the sequential API, An Encoder–Decoder Network for Neural Machine Translation

softmax regression, Softmax Regression-Softmax Regression

softplus activation function, Regression MLPs

spam filters, The Machine Learning Landscape-Why Use Machine Learning?, Types of Machine Learning Systems-Supervised learning

sparse autoencoders, Sparse Autoencoders-Sparse Autoencoders

sparse features, SVC class, SVM Classes and Computational Complexity

sparse matrix, Handling Text and Categorical Attributes, Feature Scaling and Transformation, Transformation Pipelines

sparse models, Lasso Regression, AdamW

sparse tensors, [Sparse Tensors](#)
sparsity loss, [Sparse Autoencoders](#)
specificity, ROC curve, [The ROC Curve](#)
spectral clustering, [Other Clustering Algorithms](#)
speech recognition, [Why Use Machine Learning?](#), [Examples of Applications](#)
split node, decision tree, [Making Predictions](#)
SPMD (single program, multiple data), [Data Parallelism-Bandwidth saturation](#)
squared hinge loss, [Under the Hood of Linear SVM Classifiers](#)
Stable Diffusion, [Diffusion Models](#)
stacked autoencoders, [Stacked Autoencoders-Training One Autoencoder at a Time](#)
stacking (stacked generalization), [Stacking-Stacking](#)
stale gradients, [Asynchronous updates](#)
standard correlation coefficient, [Look for Correlations](#)
standard deviation, [Take a Quick Look at the Data Structure](#)
standardization, [Feature Scaling and Transformation](#)
StandardScaler, [Feature Scaling and Transformation](#), [Gradient Descent](#),
[Ridge Regression](#), [Nonlinear SVM Classification](#)
state-action values (Q-Values), [Markov Decision Processes-Markov Decision Processes](#)
stateful metric, [Custom Metrics](#)
stateful RNN, [Natural Language Processing with RNNs and Attention](#),
[Stateful RNN-Stateful RNN](#)

stateless RNN, Natural Language Processing with RNNs and Attention,
Stateful RNN, Stateful RNN

stationary time series, Forecasting a Time Series

statistical mode, Bagging and Pasting

statistical significance, Regularization Hyperparameters

statsmodels library, The ARMA Model Family

stemming, Exercises

step functions, TLU, The Perceptron

stochastic gradient boosting, Gradient Boosting

stochastic gradient descent (SGD), Stochastic Gradient Descent-Stochastic Gradient Descent, SVM Classes and Computational Complexity

early stopping, Early Stopping

image classification, Compiling the model

and perceptron learning algorithm, The Perceptron

ridge regularization, Ridge Regression

and TD learning, Temporal Difference Learning

training binary classifier, Training a Binary Classifier

stratified sampling, Create a Test Set-Create a Test Set, Measuring Accuracy Using Cross-Validation

strat_train_set, Prepare the Data for Machine Learning Algorithms

streaming metric, Custom Metrics

strides, Convolutional Layers, Semantic Segmentation, Using 1D convolutional layers to process sequences

string kernels, Gaussian RBF Kernel

string tensors, Other Data Structures

StringLookup layer, The StringLookup Layer

strings, Strings-Strings

strong learners, Voting Classifiers

style mixing, StyleGAN, StyleGANs

style transfer, GANs, StyleGANs

StyleGANs, StyleGANs-StyleGANs

subclassing API, Using the Subclassing API to Build Dynamic Models-Using the Subclassing API to Build Dynamic Models

subgradient vector, Lasso Regression

subsampling, pooling layer, Pooling Layers

subword regularization, Sentiment Analysis

super-convergence, Learning Rate Scheduling

super-resolution, Semantic Segmentation

supervised learning, Supervised learning

support vector machines (SVMs), Support Vector Machines-Kernelized SVMs

decision function, Under the Hood of Linear SVM Classifiers-Under the Hood of Linear SVM Classifiers

dual problem, The Dual Problem-Kernelized SVMs

kernelized SVMs, Kernelized SVMs-Kernelized SVMs

linear classification, Linear SVM Classification-Soft Margin

Classification

mechanics of, [Under the Hood of Linear SVM Classifiers-Kernelized SVMs](#)

in multiclass classification, [Multiclass Classification](#)

nonlinear classifiers, [Nonlinear SVM Classification](#)

one-class SVM, [Other Algorithms for Anomaly and Novelty Detection](#)

SVM regression, [SVM Regression-SVM Regression](#)

support vectors, [Linear SVM Classification](#)

SVC class, [Polynomial Kernel, SVM Classes and Computational Complexity](#)

SVD (singular value decomposition), [The Normal Equation, Principal Components, Randomized PCA](#)

SVMs (see support vector machines)

SVR class, [SVM Regression](#)

Swish activation function, [GELU, Swish, and Mish](#)

Swiss roll dataset, [Manifold Learning-Manifold Learning](#)

symbolic differentiation, [Forward-Mode Autodiff](#)

symbolic tensors, [AutoGraph and Tracing, TF Functions and Concrete Functions](#)

synchronous updates, with centralized parameters, [Synchronous updates](#)

synthetic minority oversampling technique (SMOTE), [AlexNet](#)

T

t-distributed stochastic neighbor embedding (t-SNE), [Other Dimensionality Reduction Techniques, Visualizing the Fashion MNIST Dataset](#)

target attributes, [Take a Quick Look at the Data Structure](#)

target distribution, transforming, [Feature Scaling and Transformation](#)

target model, DQN, [Fixed Q-value Targets](#)

TD error, [Temporal Difference Learning](#)

TD target, [Temporal Difference Learning](#)

TD-Gammon, [Reinforcement Learning](#)

teacher forcing, [An Encoder–Decoder Network for Neural Machine Translation](#)

temperature, Char-RNN model, [Generating Fake Shakespearean Text](#)

temporal difference (TD) learning, [Temporal Difference Learning](#), [Prioritized Experience Replay](#)

tensor arrays, [Tensor Arrays](#)

tensor processing units (TPUs), [A Quick Tour of TensorFlow](#), [Deploying a new model version](#), [Deploying a Model to a Mobile or Embedded Device](#), [Training a Model on a TensorFlow Cluster](#)

[TensorBoard](#), [Using TensorBoard for Visualization](#)-[Using TensorBoard for Visualization](#), [A Quick Tour of TensorFlow](#), [Masking](#), [Running Large Training Jobs on Vertex AI](#)

[TensorFlow](#), [Objective and Approach](#), [Objective and Approach](#), [Custom Models and Training with TensorFlow](#)-[The TensorFlow Datasets Project](#), [Training and Deploying TensorFlow Models at Scale](#)-[Hyperparameter Tuning on Vertex AI](#)

(see also [Keras API](#))

architecture, [A Quick Tour of TensorFlow](#)

creating training function, [Using the Dataset with Keras](#)

custom models (see custom models and training algorithms)

deploying model to a mobile device, [Deploying a Model to a Mobile or Embedded Device](#)-[Deploying a Model to a Mobile or Embedded Device](#)

functions and graphs, [TensorFlow Graphs](#)-[Using TF Functions with Keras \(or Not\)](#)

GPU management with, [Managing the GPU RAM](#)-[Managing the GPU RAM](#), [Parallel Execution Across Multiple Devices](#)-[Parallel Execution Across Multiple Devices](#)

graphs and functions, [A Quick Tour of TensorFlow](#), [TensorFlow Functions and Graphs](#)-[TF Function Rules](#), [TensorFlow Graphs](#)-[Using TF Functions with Keras \(or Not\)](#)

`hub.KerasLayer`, [Using Pretrained Language Model Components](#)

math operations, [Tensors and Operations](#)

with NumPy, [Using TensorFlow like NumPy](#)-[Other Data Structures](#)

operations (ops) and tensors, [A Quick Tour of TensorFlow](#), [Tensors and Operations](#)-[Tensors and NumPy](#)

parallelism to train models (see parallelism)

platforms and APIs available, [A Quick Tour of TensorFlow](#)

serving a model (see [TensorFlow Serving](#))

special data structures, [Special Data Structures](#)-[Queues](#)

`tf.add()`, [Tensors and Operations](#)

`tf.autograph.to_code()`, [AutoGraph and Tracing](#)

`tf.cast()`, [Type Conversions](#)

`tf.config.set_soft_device_placement`, [Placing Operations and Variables on Devices](#)

`tf.config.threading.set_inter_op_parallelism_threads()`, [Parallel Execution Across Multiple Devices](#)

`tf.config.threading.set_intra_op_parallelism_threads()`, [Parallel Execution Across Multiple Devices](#)

`tf.constant()`, [Tensors and Operations](#)

`tf.data` API (see [tf.data API](#))

`tf.device()`, [Placing Operations and Variables on Devices](#)

`tf.distribute.experimental.CentralStorageStrategy`, [Training at Scale Using the Distribution Strategies API](#)

`tf.distribute.experimental.TPUStrategy`, [Training a Model on a TensorFlow Cluster](#)

`tf.distribute.MirroredStrategy`, [Training at Scale Using the Distribution Strategies API](#), [Hyperparameter Tuning on Vertex AI](#)

`tf.distribute.MultiWorkerMirroredStrategy`, [Training a Model on a TensorFlow Cluster](#)

`tf.float32`, [Tensors and NumPy](#)

`tf.float32` data type, [Custom Metrics](#), [Preprocessing the Data](#)

`tf.function()`, [TensorFlow Functions and Graphs](#), [TF Function Rules](#)

`tf.int32` data type, [Other Data Structures](#)

`tf.io.decode_base64()`, [Running Batch Prediction Jobs on Vertex AI](#)

`tf.io.decode_csv()`, [Preprocessing the Data](#)

`tf.io.decode_image()`, [Running Batch Prediction Jobs on Vertex AI](#)

`tf.io.decode_png()`, [Running Batch Prediction Jobs on Vertex AI](#)

`tf.io.decode_proto()`, [A Brief Introduction to Protocol Buffers](#)

`tf.io.FixedLenFeature`, [Loading and Parsing Examples](#)

`tf.io.parse_example()`, [Loading and Parsing Examples](#)

`tf.io.parse_sequence_example()`, [Handling Lists of Lists Using the SequenceExample Protobuf](#)

`tf.io.parse_single_example()`, [Loading and Parsing Examples](#)

`tf.io.parse_single_sequence_example()`, [Handling Lists of Lists Using the SequenceExample Protobuf](#)

`tf.io.parse_tensor()`, [Loading and Parsing Examples](#)

`tf.io.serialize_tensor()`, [Loading and Parsing Examples](#)

`tf.io.TFRecordOptions`, [Compressed TFRecord Files](#)

`tf.io.TFRecordWriter`, [The TFRecord Format](#)

`tf.io.VarLenFeature`, [Loading and Parsing Examples](#)

`tf.linalg.band_part()`, [Multi-head attention](#)

`tf.lite.TFLiteConverter.from_keras_model()`, [Deploying a Model to a Mobile or Embedded Device](#)

`tf.make_tensor_proto()`, [Querying TF Serving through the gRPC API](#)

`tf.matmul()`, [Tensors and Operations](#), [Multi-head attention](#)

`tf.nn.conv2d()`, [Implementing Convolutional Layers with Keras](#)

`tf.nn.embedding_lookup()`, [Image Preprocessing Layers](#)

`tf.nn.local_response_normalization()`, [AlexNet](#)

`tf.nn.moments()`, [Image Preprocessing Layers](#)

`tf.nn.sampled_softmax_loss()`, [An Encoder–Decoder Network for Neural Machine Translation](#)

`tf.py_function()`, [TF Function Rules](#), [Exporting SavedModels](#)

`tf.queue` module, [Other Data Structures](#)

`tf.queue.FIFOQueue`, [Queues](#)

`tf.RaggedTensor`, [Other Data Structures](#)

`tf.random.categorical()`, [Generating Fake Shakespearean Text](#)

`tf.reduce_max()`, [Implementing Pooling Layers with Keras](#)

`tf.reduce_mean()`, [Custom Training Loops](#)

`tf.reduce_sum()`, [TensorFlow Functions and Graphs](#)

`tf.saved_model_cli` command, [Exporting SavedModels](#)

`tf.sets` module, [Other Data Structures](#)

`tf.sort()`, [TF Function Rules](#)

`tf.SparseTensor`, [Other Data Structures](#)

`tf.stack()`, [Preprocessing the Data](#), [Stateful RNN](#)

`tf.string` data type, [Other Data Structures](#)

`tf.strings` module, [Other Data Structures](#)

`tf.Tensor`, [Tensors and Operations](#), [Variables](#)

`tf.TensorArray`, [Other Data Structures](#)

`tf.transpose()`, [Tensors and Operations](#)

`tf.Variable`, [Variables](#)

`tf.Variable.assign()`, [Variables](#)

type conversions, [Type Conversions](#)

variables, [Variables](#)

web page, running a model in, [Running a Model in a Web Page](#)

TensorFlow cluster, [Training a Model on a TensorFlow Cluster](#)-[Training a Model on a TensorFlow Cluster](#)

TensorFlow Datasets (TFDS) project, [The TensorFlow Datasets Project](#)-[The TensorFlow Datasets Project](#)

TensorFlow Extended (TFX), [A Quick Tour of TensorFlow](#)

TensorFlow Hub, [A Quick Tour of TensorFlow](#), [Using Pretrained Language Model Components](#), [Reusing Pretrained Embeddings and Language Models](#)

TensorFlow Lite, [A Quick Tour of TensorFlow](#)

TensorFlow playground, [Classification MLPs](#)

TensorFlow Serving (TF Serving), [Serving a TensorFlow Model](#)-[Running Batch Prediction Jobs on Vertex AI](#)

batch prediction jobs on Vertex AI, [Running Batch Prediction Jobs on Vertex AI](#)

creating prediction service, [Creating a Prediction Service on Vertex AI](#)-[Creating a Prediction Service on Vertex AI](#)

deploying new model version, [Deploying a new model version](#)-[Deploying a new model version](#)

Docker container, [Installing and starting TensorFlow Serving](#)

exporting SavedModels, [Exporting SavedModels](#)-[Exporting SavedModels](#)

gRPC API, querying through, [Querying TF Serving through the gRPC API](#)

installing and starting up, [Installing and starting TensorFlow Serving](#)-[Installing and starting TensorFlow Serving](#)

REST API, querying through, [Querying TF Serving through the REST API](#)

TensorFlow Text, [Text Preprocessing](#), [Sentiment Analysis](#)

TensorFlow.js (TFJS) JavaScript library, [Running a Model in a Web Page](#)

tensors, [Tensors and Operations-Tensors and NumPy](#)

term-frequency x inverse-document-frequency (TF-IDF), [Text Preprocessing](#)

terminal state, Markov chain, [Markov Decision Processes](#)

test set, [Testing and Validating](#), [Create a Test Set](#)-[Create a Test Set](#)

text attributes, [Handling Text and Categorical Attributes](#)

text processing (see natural language processing)

TF Serving (see TensorFlow Serving)

tf.data API, [The tf.data API-Using the Dataset with Keras](#)

chaining transformations, [Chaining Transformations-Chaining Transformations](#)

interleaving lines from multiple files, [Interleaving Lines from Multiple Files-Interleaving Lines from Multiple Files](#)

and Keras preprocessing layers, [The Normalization Layer](#)

prefetching, [Prefetching-Prefetching](#)

preprocessing the data, [Preprocessing the Data-Preprocessing the Data](#)

shuffling data, [Shuffling the Data-Shuffling the Data](#)

tf.data.AUTOTUNE, [Chaining Transformations](#)

tf.data.Dataset.from_tensor_slices(), [The tf.data API-Chaining Transformations](#)

`tf.data.TFRecordDataset`, [The TFRecord Format](#), [The TFRecord Format](#), [Loading and Parsing Examples](#)

using dataset with Keras, [Using the Dataset with Keras](#)-[Using the Dataset with Keras](#)

TFDS (TensorFlow Datasets) project, [The TensorFlow Datasets Project](#)-[The TensorFlow Datasets Project](#)

TFJS (TensorFlow.js) JavaScript library, [Running a Model in a Web Page](#)

TFLite, [Deploying a Model to a Mobile or Embedded Device](#)-[Deploying a Model to a Mobile or Embedded Device](#)

TFRecord format, [Loading and Preprocessing Data with TensorFlow](#), [The TFRecord Format](#)-[Handling Lists of Lists Using the SequenceExample](#) [Protobuf](#)

TFX (TensorFlow Extended), [A Quick Tour of TensorFlow](#)

theoretical information criterion, [Selecting the Number of Clusters](#)

3D convolutional layers, [Semantic Segmentation](#)

threshold logic units (TLUs), [The Perceptron](#), [The Multilayer Perceptron](#) and [Backpropagation](#)

Tikhonov regularization, [Ridge Regression](#)-[Ridge Regression](#), [Elastic Net](#) [Regression](#)

time series data, forecasting, [Processing Sequences Using RNNs and CNNs](#), [Forecasting a Time Series](#)-[Forecasting Using a Sequence-to-Sequence Model](#)

ARMA model family, [The ARMA Model Family](#)-[The ARMA Model Family](#)

data preparation for ML models, [Preparing the Data for Machine Learning Models](#)-[Preparing the Data for Machine Learning Models](#)

with deep RNN, [Forecasting Using a Deep RNN](#)

with linear model, [Forecasting Using a Linear Model](#)

multivariate time series, [Forecasting a Time Series](#), [Forecasting Multivariate Time Series](#)-[Multivariate Time Series-Forecasting Multivariate Time Series](#)

with sequence-to-sequence model, [Input and Output Sequences](#), [Forecasting Using a Sequence-to-Sequence Model](#)-[Forecasting Using a Sequence-to-Sequence Model](#)

several time steps ahead, [Forecasting Several Time Steps Ahead](#)-[Forecasting Several Time Steps Ahead](#)

with simple RNN, [Forecasting Using a Simple RNN](#)-[Forecasting Using a Simple RNN](#)

TLUs (threshold logic units), [The Perceptron](#), [The Multilayer Perceptron](#) and [Backpropagation](#)

TNR (true negative rate), [The ROC Curve](#)

tokenizers library, [Sentiment Analysis](#)

tolerance (ϵ), [Batch Gradient Descent](#), [SVM Classes and Computational Complexity](#)

TPR (true positive rate), [Confusion Matrices](#), [The ROC Curve](#)

TPUs (tensor processing units), [A Quick Tour of TensorFlow](#), [Deploying a new model version](#), [Deploying a Model to a Mobile or Embedded Device](#), [Training a Model on a TensorFlow Cluster](#)

train-dev set, [Data Mismatch](#)

training, [Model-based learning](#) and a typical machine learning workflow

training instance, [What Is Machine Learning?](#)

training loops, [Custom Training Loops](#)-[Custom Training Loops](#), [Using the Dataset with Keras](#)

training models, [Training Models-Softmax Regression](#)

learning curves in, [Learning Curves-Learning Curves](#)

linear regression, [Training Models](#), [Linear Regression-Mini-Batch Gradient Descent](#)

logistic regression, [Logistic Regression-Softmax Regression](#)

perceptrons, [The Perceptron-The Perceptron](#)

polynomial regression, [Training Models](#), [Polynomial Regression-Polynomial Regression](#)

training set, [What Is Machine Learning?](#), [Testing and Validating](#)

cost function of, [Training and Cost Function-Training and Cost Function](#)

insufficient quantities, [Insufficient Quantity of Training Data](#)

irrelevant features, [Irrelevant Features](#)

min-max scaling, [Feature Scaling and Transformation](#)

nonrepresentative, [Nonrepresentative Training Data](#)

overfitting, [Overfitting the Training Data-Overfitting the Training Data](#)

preparing for ML algorithms, [Prepare the Data for Machine Learning Algorithms](#)

training and evaluating on, [Train and Evaluate on the Training Set-Train and Evaluate on the Training Set](#)

transforming data, [Feature Scaling and Transformation](#)

underfitting, [Underfitting the Training Data](#)

visualizing data, [Explore and Visualize the Data to Gain Insights](#)

training set expansion, [Exercises](#), [AlexNet](#), [Pretrained Models for Transfer Learning](#)

training/serving skew, Loading and Preprocessing Data with TensorFlow

train_test_split(), Create a Test Set, Better Evaluation Using Cross-Validation

transfer learning, Self-supervised learning, Reusing Pretrained Layers, Transfer Learning with Keras-Transfer Learning with Keras, Pretrained Models for Transfer Learning-Pretrained Models for Transfer Learning

transform(), Clean the Data, Handling Text and Categorical Attributes, Feature Scaling and Transformation, Custom Transformers

transformation of data

custom transformers, Custom Transformers-Custom Transformers

estimator transformers, Clean the Data

and feature scaling, Feature Scaling and Transformation-Feature Scaling and Transformation

transformer models (see transformer models)

transformation pipelines, Transformation Pipelines-Transformation Pipelines

TransformedTargetRegressor, Feature Scaling and Transformation

transformer, Attention Is All You Need: The Original Transformer Architecture

transformer models

attention mechanisms, Attention Is All You Need: The Original Transformer Architecture-Multi-head attention, Vision Transformers

BERT, An Avalanche of Transformer Models

DistilBERT, An Avalanche of Transformer Models, Hugging Face's Transformers Library-Hugging Face's Transformers Library

Hugging Face library, Hugging Face's Transformers Library-Hugging Face's Transformers Library

Pathways language model, [An Avalanche of Transformer Models](#)
vision transformers, [Vision Transformers](#)-[Vision Transformers](#)
TransformerMixin, [Custom Transformers](#)
transformers library, [Hugging Face's Transformers Library](#)-[Hugging Face's Transformers Library](#)
translation, with RNNs, [Natural Language Processing with RNNs and Attention](#), [An Encoder–Decoder Network for Neural Machine Translation–Beam Search](#)
(see also transformer models)
transpose operator, [Select a Performance Measure](#)
transposed convolutional layer, [Semantic Segmentation](#)-[Semantic Segmentation](#)
true negative rate (TNR), [The ROC Curve](#)
true negatives, confusion matrix, [Confusion Matrices](#)
true positive rate (TPR), [Confusion Matrices](#), [The ROC Curve](#)
true positives, confusion matrix, [Confusion Matrices](#)
trust region policy optimization (TRPO), [Overview of Some Popular RL Algorithms](#)
2D convolutional layers, [Implementing Convolutional Layers with Keras](#)
tying weights, [Tying Weights](#)
type I errors, confusion matrix, [Confusion Matrices](#)
type II errors, confusion matrix, [Confusion Matrices](#)

U

uncertainty sampling, [Using Clustering for Semi-Supervised Learning](#)

undercomplete, autoencoder as, [Efficient Data Representations-Performing PCA with an Undercomplete Linear Autoencoder](#)

underfitting of data, [Underfitting the Training Data, Train and Evaluate on the Training Set, Learning Curves-Learning Curves, Gaussian RBF Kernel](#)

univariate regression, [Frame the Problem](#)

univariate time series, [Forecasting a Time Series](#)

Universal Sentence Encoder, [Reusing Pretrained Embeddings and Language Models-Reusing Pretrained Embeddings and Language Models](#)

unreasonable effectiveness of data, [Insufficient Quantity of Training Data](#)

unrolling the network through time, [Recurrent Neurons and Layers](#)

unstable gradients problem, [The Vanishing/Exploding Gradients Problems, Fighting the Unstable Gradients Problem](#)

(see also vanishing and exploding gradients)

unsupervised learning, [Unsupervised learning-Unsupervised learning, Unsupervised Learning Techniques-Other Algorithms for Anomaly and Novelty Detection](#)

anomaly detection, [Unsupervised learning](#)

association rule learning, [Unsupervised learning](#)

autoencoders (see autoencoders)

clustering (see clustering algorithms)

density estimation, [Unsupervised Learning Techniques](#)

diffusion models, [Diffusion Models-Diffusion Models](#)

dimensionality reduction (see dimensionality reduction)

GANs (see generative adversarial networks)

GMM, Gaussian Mixtures-Other Algorithms for Anomaly and Novelty Detection

k-means (see k-means algorithm)

novelty detection, Unsupervised learning

pretraining, Unsupervised Pretraining, Reusing Pretrained Embeddings and Language Models, An Avalanche of Transformer Models, Unsupervised Pretraining Using Stacked Autoencoders

stacked autoencoders, Unsupervised Pretraining Using Stacked Autoencoders

transformer models, An Avalanche of Transformer Models

visualization algorithms, Unsupervised learning-Unsupervised learning

upsampling layer, Semantic Segmentation

utility function, Model-based learning and a typical machine learning workflow

V

VAEs (variational autoencoders), Variational Autoencoders-Variational Autoencoders

“valid” padding, computer vision, Implementing Convolutional Layers with Keras

validation set, Hyperparameter Tuning and Model Selection, Training and evaluating the model-Training and evaluating the model

value_counts(), Take a Quick Look at the Data Structure

vanishing and exploding gradients, The Vanishing/Exploding Gradients Problems-Gradient Clipping

activation function improvements, [Better Activation Functions-GELU, Swish, and Mish](#)

batch normalization, [Batch Normalization-Implementing batch normalization with Keras](#)

Glorot and He initialization, [Glorot and He Initialization-Glorot and He Initialization](#)

gradient clipping, [Gradient Clipping](#)

unstable gradients problem, [Fighting the Unstable Gradients Problem-Fighting the Unstable Gradients Problem](#)

variables

handling in TF functions, [Handling Variables and Other Resources in TF Functions-Handling Variables and Other Resources in TF Functions](#)

persistence of, [Custom Metrics](#)

placing on GPUs, [Placing Operations and Variables on Devices](#)

in TensorFlow, [Variables](#)

variance

bias/variance trade-off, [Learning Curves](#)

explained, [Explained Variance Ratio-Choosing the Right Number of Dimensions](#)

high variance with decision trees, [Decision Trees Have a High Variance](#)

preserving, [Preserving the Variance](#)

variational autoencoders (VAEs), [Variational Autoencoders-Variational Autoencoders](#)

vector-to-sequence network, [Input and Output Sequences](#)

vectors, norms for measuring distance, [Select a Performance Measure](#)

[Vertex AI](#), [Launch, Monitor, and Maintain Your System](#), [Creating a Prediction Service on Vertex AI](#)-[Creating a Prediction Service on Vertex AI](#), [Running Large Training Jobs on Vertex AI](#)-[Running Large Training Jobs on Vertex AI](#)

[VGGNet](#), [VGGNet](#)

virtual GPU device, [Managing the GPU RAM](#)

vision transformers (ViTs), [Vision Transformers](#)-[Vision Transformers](#)

visual cortex architecture, [The Architecture of the Visual Cortex](#)

visualization of data, [Examples of Applications](#), [Unsupervised learning](#)-[Unsupervised learning](#), [Explore and Visualize the Data to Gain Insights](#)-[Experiment with Attribute Combinations](#)

decision trees, [Training and Visualizing a Decision Tree](#)-[Training and Visualizing a Decision Tree](#)

dimensionality reduction, [Dimensionality Reduction](#), [Choosing the Right Number of Dimensions](#)

end-to-end exercise, [Explore and Visualize the Data to Gain Insights](#)-[Experiment with Attribute Combinations](#)

MLPs with TensorBoard, [Using TensorBoard for Visualization](#)-[Using TensorBoard for Visualization](#)

stacked autoencoders, [Visualizing the Reconstructions](#)-[Visualizing the Reconstructions](#)

t-SNE, [Other Dimensionality Reduction Techniques](#)

ViTs (vision transformers), [Vision Transformers](#)-[Vision Transformers](#)

voting classifiers, [Voting Classifiers](#)-[Voting Classifiers](#)

W

wall time, [Batch Normalization](#)

warmup phase, asynchronous model updates, [Asynchronous updates](#)

WaveNet, [Processing Sequences Using RNNs and CNNs](#), WaveNet-WaveNet

weak learners, [Voting Classifiers](#)

web page, running a model in, [Running a Model in a Web Page](#)

weight decay, [AdamW](#)

weight stashing, [Bandwidth saturation](#)

weight-tying, [Tying Weights](#)

weights

boosting, [AdaBoost-AdaBoost](#)

convolutional layers, [Implementing Convolutional Layers with Keras](#)

freezing reused layers, [Reusing Pretrained Layers](#)

of hidden layers, [Creating the model using the sequential API](#)

in prioritized experience replay, [Prioritized Experience Replay](#)

saving instead of whole model, [Saving and Restoring a Model](#)

white box models, [Making Predictions](#)

Wide & Deep neural network, [Building Complex Models Using the Functional API-Building Complex Models Using the Functional API](#)

window length, [Creating the Training Dataset](#)

wisdom of the crowd, [Ensemble Learning and Random Forests](#)

word embeddings, [Encoding Categorical Features Using Embeddings](#)

neural machine translation, [An Encoder–Decoder Network for Neural Machine Translation-Beam Search](#)

sentiment analysis, [Sentiment Analysis-Reusing Pretrained Embeddings and Language Models](#)

worker task type, [Training a Model on a TensorFlow Cluster](#)

workers, [Data parallelism with centralized parameters](#)

X

Xavier initialization, [Glorot and He Initialization](#)

Xception (Extreme Inception), [Xception-Xception, Pretrained Models for Transfer Learning-Pretrained Models for Transfer Learning](#)

XLA (accelerated linear algebra), [TensorFlow Functions and Graphs](#)

XOR (exclusive or) problem, [The Perceptron](#)

Y

You Only Look Once (YOLO), [You Only Look Once-You Only Look Once](#)

Z

zero padding, [Convolutional Layers, Implementing Convolutional Layers with Keras](#)

zero-shot learning (ZSL), [An Avalanche of Transformer Models, Vision Transformers](#)

ZFNet, [AlexNet](#)

About the Author

Aurélien Géron is a machine learning consultant and lecturer. A former Googler, he led YouTube's video classification team from 2013 to 2016. He's been a founder of and CTO at a few different companies: Wifirst, a leading wireless ISP in France; Polyconseil, a consulting firm focused on telecoms, media, and strategy; and Kiwisoft, a consulting firm focused on machine learning and data privacy.

Before all that Aurélien worked as an engineer in a variety of domains: finance (JP Morgan and Société Générale), defense (Canada's DOD), and healthcare (blood transfusion). He also published a few technical books (on C++, WiFi, and internet architectures) and lectured about computer science at a French engineering school.

A few fun facts: he taught his three children to count in binary with their fingers (up to 1,023), he studied microbiology and evolutionary genetics before going into software engineering, and his parachute didn't open on the second jump.

Colophon

The animal on the cover of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* is the fire salamander (*Salamandra salamandra*), an amphibian found across most of Europe. Its black, glossy skin features large yellow spots on the head and back, signaling the presence of alkaloid toxins. This is a possible source of this amphibian's common name: contact with these toxins (which they can also spray short distances) causes convulsions and hyperventilation. Either the painful poisons or the moistness of the salamander's skin (or both) led to a misguided belief that these creatures not only could survive being placed in fire but could extinguish it as well.

Fire salamanders live in shaded forests, hiding in moist crevices and under logs near the pools or other freshwater bodies that facilitate their breeding. Though they spend most of their lives on land, they give birth to their young in water. They subsist mostly on a diet of insects, spiders, slugs, and worms. Fire salamanders can grow up to a foot in length, and in captivity may live as long as 50 years.

The fire salamander's numbers have been reduced by destruction of their forest habitat and capture for the pet trade, but the greatest threat they face is the susceptibility of their moisture-permeable skin to pollutants and microbes. Since 2014, they have become extinct in parts of the Netherlands and Belgium due to an introduced fungus.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. The cover illustration is by Karen Montgomery, based on an engraving from *Wood's Illustrated Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.