we will present the most popular ones: Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and finally Adam and Nadam optimization.

Momentum Optimization

Imagine a bowling ball rolling down a gentle slope on a smooth surface: it will start out slowly, but it will quickly pick up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance). This is the very simple idea behind *Momentum optimization*, proposed by Boris Polyak in 1964.[12] In contrast, regular Gradient Descent will simply take small regular steps down the slope, so it will take much more time to reach the bottom.

Recall that Gradient Descent simply updates the weights $\boldsymbol{\theta}$ by directly subtracting the gradient of the cost function $J(\boldsymbol{\theta})$ with regards to the weights ($\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$) multiplied by the learning rate η. The equation is: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.

Momentum optimization cares a great deal about what previous gradients were: at each iteration, it subtracts the local gradient from the *momentum vector* $\mathbf{m}$ (multiplied by the learning rate η), and it updates the weights by simply adding this momentum vector (see Equation 11-4). In other words, the gradient is used for acceleration, not for speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter β, simply called the *momentum*, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

Equation 11-4. Momentum algorithm

1. $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$

You can easily verify that if the gradient remains constant, the terminal velocity (i.e., the maximum size of the weight updates) is equal to that gradient multiplied by the learning rate η multiplied by $\frac{1}{1-\beta}$ (ignoring the sign). For example, if $\beta = 0.9$, then the terminal velocity is equal to 10 times the gradient times the learning rate, so Momentum optimization ends up going 10 times faster than Gradient Descent! This allows Momentum optimization to escape from plateaus much faster than Gradient Descent. In particular, we saw in Chapter 4 that when the inputs have very different scales the cost function will look like an elongated bowl (see Figure 4-7). Gradient Descent goes down the steep slope quite fast, but then it takes a very long time to go down the val-

12 "Some methods of speeding up the convergence of iteration methods," B. Polyak (1964).

ley. In contrast, Momentum optimization will roll down the valley faster and faster until it reaches the bottom (the optimum). In deep neural networks that don't use Batch Normalization, the upper layers will often end up having inputs with very different scales, so using Momentum optimization helps a lot. It can also help roll past local optima.

Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum. This is one of the reasons why it is good to have a bit of friction in the system: it gets rid of these oscillations and thus speeds up convergence.

Implementing Momentum optimization in Keras is a no-brainer: just use the `SGD` optimizer and set its `momentum` hyperparameter, then lie back and profit!

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

The one drawback of Momentum optimization is that it adds yet another hyperparameter to tune. However, the momentum value of 0.9 usually works well in practice and almost always goes faster than regular Gradient Descent.

Nesterov Accelerated Gradient

One small variant to Momentum optimization, proposed by Yurii Nesterov in 1983,[13] is almost always faster than vanilla Momentum optimization. The idea of *Nesterov Momentum optimization*, or *Nesterov Accelerated Gradient* (NAG), is to measure the gradient of the cost function not at the local position but slightly ahead in the direction of the momentum (see Equation 11-5). The only difference from vanilla Momentum optimization is that the gradient is measured at $\boldsymbol{\theta} + \beta\mathbf{m}$ rather than at $\boldsymbol{\theta}$.

Equation 11-5. Nesterov Accelerated Gradient algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \beta\mathbf{m})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$

This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than using the gradient at the original position, as you can see in Figure 11-6 (where ∇_1 represents the gradient of the cost function measured at the starting point $\boldsymbol{\theta}$, and ∇_2 represents the

13 "A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence $O(1/k^2)$," Yurii Nesterov (1983).

gradient at the point located at $\boldsymbol{\theta} + \beta\mathbf{m}$). As you can see, the Nesterov update ends up slightly closer to the optimum. After a while, these small improvements add up and NAG ends up being significantly faster than regular Momentum optimization. Moreover, note that when the momentum pushes the weights across a valley, ∇_1 continues to push further across the valley, while ∇_2 pushes back toward the bottom of the valley. This helps reduce oscillations and thus converges faster.

NAG will almost always speed up training compared to regular Momentum optimization. To use it, simply set `nesterov=True` when creating the `SGD` optimizer:

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

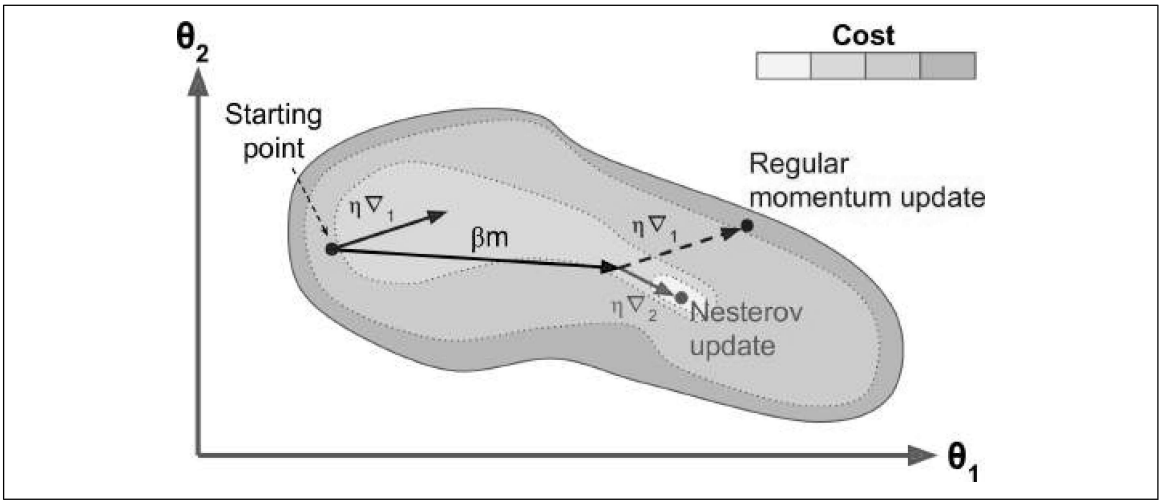

Figure 11-6. Regular versus Nesterov Momentum optimization

AdaGrad

Consider the elongated bowl problem again: Gradient Descent starts by quickly going down the steepest slope, then slowly goes down the bottom of the valley. It would be nice if the algorithm could detect this early on and correct its direction to point a bit more toward the global optimum.

The *AdaGrad* algorithm[14] achieves this by scaling down the gradient vector along the steepest dimensions (see Equation 11-6):

Equation 11-6. AdaGrad algorithm

1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \epsilon}$

14 "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," J. Duchi et al. (2011).

The first step accumulates the square of the gradients into the vector **s** (recall that the $\otimes$ symbol represents the element-wise multiplication). This vectorized form is equivalent to computing $s_i \leftarrow s_i + (\partial J(\mathbf{\theta}) / \partial \theta_i)^2$ for each element s_i of the vector **s**; in other words, each s_i accumulates the squares of the partial derivative of the cost function with regards to parameter θ_i. If the cost function is steep along the ith dimension, then s_i will get larger and larger at each iteration.

The second step is almost identical to Gradient Descent, but with one big difference: the gradient vector is scaled down by a factor of $\sqrt{\mathbf{s} + \epsilon}$ (the $\oslash$ symbol represents the element-wise division, and ϵ is a smoothing term to avoid division by zero, typically set to 10^{-10}). This vectorized form is equivalent to computing $\theta_i \leftarrow \theta_i - \eta \, \partial J(\mathbf{\theta}) / \partial \theta_i / \sqrt{s_i + \epsilon}$ for all parameters θ_i (simultaneously).

In short, this algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an *adaptive learning rate*. It helps point the resulting updates more directly toward the global optimum (see Figure 11-7). One additional benefit is that it requires much less tuning of the learning rate hyperparameter η.

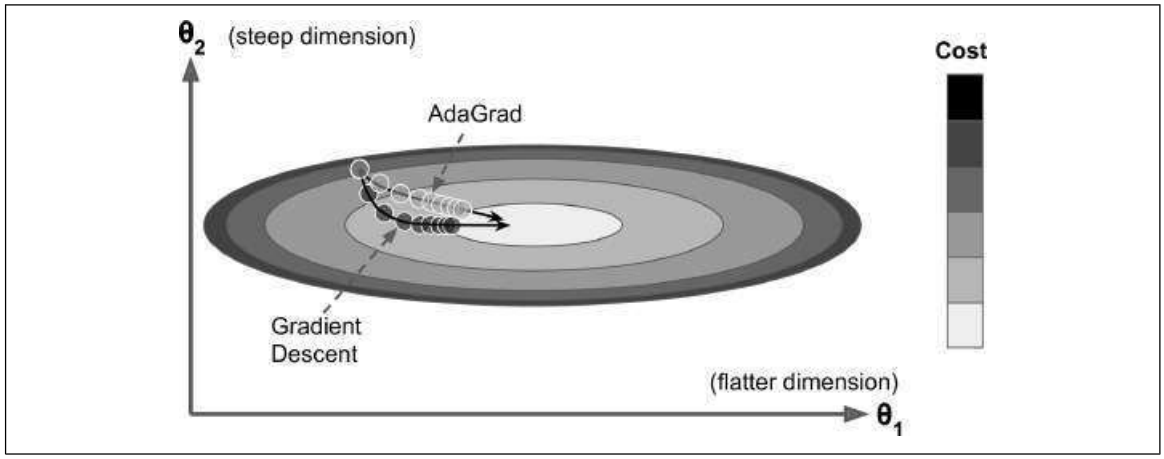

Figure 11-7. AdaGrad versus Gradient Descent

AdaGrad often performs well for simple quadratic problems, but unfortunately it often stops too early when training neural networks. The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum. So even though Keras has an `Adagrad` optimizer, you should not use it to train deep neural networks (it may be efficient for simpler tasks such as Linear Regression, though). However, understanding Adagrad is helpful to grasp the other adaptive learning rate optimizers.

RMSProp

Although AdaGrad slows down a bit too fast and ends up never converging to the global optimum, the *RMSProp* algorithm[15] fixes this by accumulating only the gradients from the most recent iterations (as opposed to all the gradients since the beginning of training). It does so by using exponential decay in the first step (see Equation 11-7).

Equation 11-7. RMSProp algorithm

1. $\mathbf{s} \leftarrow \beta\mathbf{s} + (1 - \beta)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \epsilon}$

The decay rate β is typically set to 0.9. Yes, it is once again a new hyperparameter, but this default value often works well, so you may not need to tune it at all.

As you might expect, Keras has an `RMSProp` optimizer:

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Except on very simple problems, this optimizer almost always performs much better than AdaGrad. In fact, it was the preferred optimization algorithm of many researchers until Adam optimization came around.

Adam and Nadam Optimization

Adam,[16] which stands for *adaptive moment estimation*, combines the ideas of Momentum optimization and RMSProp: just like Momentum optimization it keeps track of an exponentially decaying average of past gradients, and just like RMSProp it keeps track of an exponentially decaying average of past squared gradients (see Equation 11-8).[17]

15 This algorithm was created by Geoffrey Hinton and Tijmen Tieleman in 2012, and presented by Geoffrey Hinton in his Coursera class on neural networks (slides: *https://homl.info/57*; video: *https://homl.info/58*). Amusingly, since the authors did not write a paper to describe it, researchers often cite "slide 29 in lecture 6" in their papers.

16 "Adam: A Method for Stochastic Optimization," D. Kingma, J. Ba (2015).

17 These are estimations of the mean and (uncentered) variance of the gradients. The mean is often called the *first moment*, while the variance is often called the *second moment*, hence the name of the algorithm.

Equation 11-8. Adam algorithm

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - \left(1 - \beta_1\right) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + \left(1 - \beta_2\right) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3. $\widehat{\mathbf{m}} \leftarrow \dfrac{\mathbf{m}}{1 - \beta_1^{\,t}}$
4. $\widehat{\mathbf{s}} \leftarrow \dfrac{\mathbf{s}}{1 - \beta_2^{\,t}}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \, \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \epsilon}$

- t represents the iteration number (starting at 1).

If you just look at steps 1, 2, and 5, you will notice Adam's close similarity to both Momentum optimization and RMSProp. The only difference is that step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just $1 - \beta_1$ times the decaying sum). Steps 3 and 4 are somewhat of a technical detail: since **m** and **s** are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost **m** and **s** at the beginning of training.

The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999. As earlier, the smoothing term ϵ is usually initialized to a tiny number such as 10^{-7}. These are the default values for the `Adam` class (to be precise, `epsilon` defaults to `None`, which tells Keras to use `keras.backend.epsilon()`, which defaults to 10^{-7}; you can change it using `keras.backend.set_epsilon()`).

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

Since Adam is an adaptive learning rate algorithm (like AdaGrad and RMSProp), it requires less tuning of the learning rate hyperparameter η. You can often use the default value $\eta = 0.001$, making Adam even easier to use than Gradient Descent.

If you are starting to feel overwhelmed by all these different techniques, and wondering how to choose the right ones for your task, don't worry: some practical guidelines are provided at the end of this chapter.

Finally, two variants of Adam are worth mentioning:

- Adamax, introduced in the same paper as Adam: notice that in step 2 of Equation 11-8, Adam accumulates the squares of the gradients in **s** (with a greater weight for more recent weights). In step 5, if we ignore ϵ and steps 3 and 4 (which are technical details anyway), Adam just scales down the parameter updates by the square root of **s**. In short, Adam scales down the parameter updates by the ℓ_2 norm of the time-decayed gradients (recall that the ℓ_2 norm is the square root of the sum of squares). Adamax just replaces the ℓ_2 norm with the ℓ_∞ norm (a fancy way of saying the max). Specifically, it replaces step 2 in Equation 11-8 with $\mathbf{s} \leftarrow \max\left(\beta_2 \mathbf{s}, \nabla_\theta J(\theta)\right)$, it drops step 4, and in step 5 it scales down the gradient updates by a factor of **s**, which is just the max of the time-decayed gradients. In practice, this can make Adamax more stable than Adam, but this really depends on the dataset, and in general Adam actually performs better. So it's just one more optimizer you can try if you experience problems with Adam on some task.
- Nadam optimization[18] is more important: it is simply Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam. In his report, Timothy Dozat compares many different optimizers on various tasks, and finds that Nadam generally outperforms Adam, but is sometimes outperformed by RMSProp.

Adaptive optimization methods (including RMSProp, Adam and Nadam optimization) are often great, converging fast to a good solution. However, a 2017 paper[19] by Ashia C. Wilson et al. showed that they can lead to solutions that generalize poorly on some datasets. So when you are disappointed by your model's performance, try using plain Nesterov Accelerated Gradient instead: your dataset may just be allergic to adaptive gradients. Also check out the latest research, it is moving fast (e.g., AdaBound).

All the optimization techniques discussed so far only rely on the *first-order partial derivatives* (*Jacobians*). The optimization literature contains amazing algorithms based on the *second-order partial derivatives* (the *Hessians*, which are the partial derivatives of the Jacobians). Unfortunately, these algorithms are very hard to apply to deep neural networks because there are n^2 Hessians per output (where n is the number of parameters), as opposed to just n Jacobians per output. Since DNNs typically have tens of thousands of parameters, the second-order optimization algorithms

18 "Incorporating Nesterov Momentum into Adam," Timothy Dozat (2015).

19 "The Marginal Value of Adaptive Gradient Methods in Machine Learning," A. C. Wilson et al. (2017).

often don't even fit in memory, and even when they do, computing the Hessians is just too slow.

Training Sparse Models

All the optimization algorithms just presented produce dense models, meaning that most parameters will be nonzero. If you need a blazingly fast model at runtime, or if you need it to take up less memory, you may prefer to end up with a sparse model instead.

One trivial way to achieve this is to train the model as usual, then get rid of the tiny weights (set them to 0). However, this will typically not lead to a very sparse model, and it may degrade the model's performance.

A better option is to apply strong ℓ_1 regularization during training, as it pushes the optimizer to zero out as many weights as it can (as discussed in Chapter 4 about Lasso Regression).

However, in some cases these techniques may remain insufficient. One last option is to apply *Dual Averaging*, often called *Follow The Regularized Leader* (FTRL), a technique proposed by Yurii Nesterov.[20] When used with ℓ_1 regularization, this technique often leads to very sparse models. Keras implements a variant of FTRL called *FTRL-Proximal*[21] in the `FTRL` optimizer.

Learning Rate Scheduling

Finding a good learning rate can be tricky. If you set it way too high, training may actually diverge (as we discussed in Chapter 4). If you set it too low, training will eventually converge to the optimum, but it will take a very long time. If you set it slightly too high, it will make progress very quickly at first, but it will end up dancing around the optimum, never really settling down. If you have a limited computing budget, you may have to interrupt training before it has converged properly, yielding a suboptimal solution (see Figure 11-8).

20 "Primal-Dual Subgradient Methods for Convex Problems," Yurii Nesterov (2005).

21 "Ad Click Prediction: a View from the Trenches," H. McMahan et al. (2013).

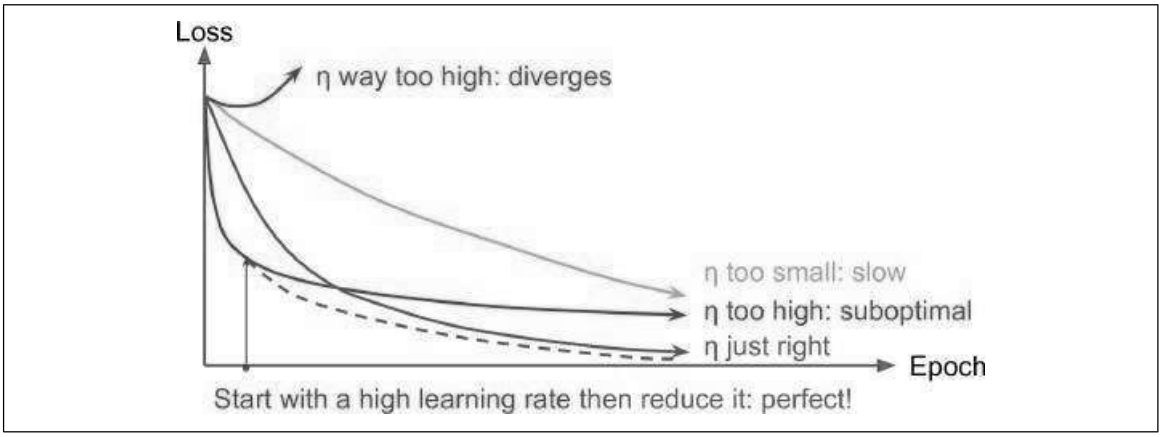

Figure 11-8. Learning curves for various learning rates η

As we discussed in Chapter 10, one approach is to start with a large learning rate, and divide it by 3 until the training algorithm stops diverging. You will not be too far from the optimal learning rate, which will learn quickly and converge to good solution.

However, you can do better than a constant learning rate: if you start with a high learning rate and then reduce it once it stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate. There are many different strategies to reduce the learning rate during training. These strategies are called *learning schedules* (we briefly introduced this concept in Chapter 4), the most common of which are:

Power scheduling

Set the learning rate to a function of the iteration number t: $\eta(t) = \eta_0 / (1 + t/k)^c$. The initial learning rate η_0, the power c (typically set to 1) and the steps s are hyperparameters. The learning rate drops at each step, and after s steps it is down to $\eta_0 / 2$. After s more steps, it is down to $\eta_0 / 3$. Then down to $\eta_0 / 4$, then $\eta_0 / 5$, and so on. As you can see, this schedule first drops quickly, then more and more slowly. Of course, this requires tuning η_0, s (and possibly c).

Exponential scheduling

Set the learning rate to: $\eta(t) = \eta_0 \, 0.1^{t/s}$. The learning rate will gradually drop by a factor of 10 every s steps. While power scheduling reduces the learning rate more and more slowly, exponential scheduling keeps slashing it by a factor of *10* every s steps.

Piecewise constant scheduling

Use a constant learning rate for a number of epochs (e.g., $\eta_0 = 0.1$ for 5 epochs), then a smaller learning rate for another number of epochs (e.g., $\eta_1 = 0.001$ for 50 epochs), and so on. Although this solution can work very well, it requires fid-

dling around to figure out the right sequence of learning rates, and how long to use each of them.

Performance scheduling
: Measure the validation error every N steps (just like for early stopping) and reduce the learning rate by a factor of λ when the error stops dropping.

A 2013 paper[22] by Andrew Senior et al. compared the performance of some of the most popular learning schedules when training deep neural networks for speech recognition using Momentum optimization. The authors concluded that, in this setting, both performance scheduling and exponential scheduling performed well. They favored exponential scheduling because it was easy to tune and it converged slightly faster to the optimal solution (they also mentioned that it was easier to implement than performance scheduling, but in Keras both options are easy).

Implementing power scheduling in Keras is the easiest option: just set the `decay` hyperparameter when creating an optimizer. The `decay` is the inverse of s (the number of steps it takes to divide the learning rate by one more unit), and Keras assumes that c is equal to 1. For example:

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

Exponential scheduling and piecewise scheduling are quite simple too. You first need to define a function that takes the current epoch and returns the learning rate. For example, let's implement exponential scheduling:

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1**(epoch / 20)
```

If you do not want to hard-code η_0 and s, you can create a function that returns a configured function:

```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1**(epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

Next, just create a `LearningRateScheduler` callback, giving it the schedule function, and pass this callback to the `fit()` method:

```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, [...], callbacks=[lr_scheduler])
```

22 "An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition," A. Senior et al. (2013).

The `LearningRateScheduler` will update the optimizer's `learning_rate` attribute at the beginning of each epoch. Updating the learning rate just once per epoch is usually enough, but if you want it to be updated more often, for example at every step, you need to write your own callback (see the notebook for an example). This can make sense if there are many steps per epoch.

The schedule function can optionally take the current learning rate as a second argument. For example, the following schedule function just multiplies the previous learning rate by $0.1^{1/20}$, which results in the same exponential decay (except the decay now starts at the beginning of epoch 0 instead of 1). This implementation relies on the optimizer's initial learning rate (contrary to the previous implementation), so make sure to set it appropriately.

```
def exponential_decay_fn(epoch, lr):
    return lr * 0.1**(1 / 20)
```

When you save a model, the optimizer and its learning rate get saved along with it. This means that with this new schedule function, you could just load a trained model and continue training where it left off, no problem. However, things are not so simple if your schedule function uses the `epoch` argument: indeed, the epoch does not get saved, and it gets reset to 0 every time you call the `fit()` method. This could lead to a very large learning rate when you continue training a model where it left off, which would likely damage your model's weights. One solution is to manually set the `fit()` method's `initial_epoch` argument so the `epoch` starts at the right value.

For piecewise constant scheduling, you can use a schedule function like the following one (as earlier, you can define a more general function if you want, see the notebook for an example), then create a `LearningRateScheduler` callback with this function and pass it to the `fit()` method, just like we did for exponential scheduling:

```
def piecewise_constant_fn(epoch):
    if epoch < 5:
        return 0.01
    elif epoch < 15:
        return 0.005
    else:
        return 0.001
```

For performance scheduling, simply use the `ReduceLROnPlateau` callback. For example, if you pass the following callback to the `fit()` method, it will multiply the learning rate by 0.5 whenever the best validation loss does not improve for 5 consecutive epochs (other options are available, please check the documentation for more details):

```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)
```

Lastly, tf.keras offers an alternative way to implement learning rate scheduling: just define the learning rate using one of the schedules available in `keras.optimiz`

ers.schedules, then pass this learning rate to any optimizer. This approach updates the learning rate at each step rather than at each epoch. For example, here is how to implement the same exponential schedule as earlier:

```
s = 20 * len(X_train) // 32 # number of steps in 20 epochs (batch size = 32)
learning_rate = keras.optimizers.schedules.ExponentialDecay(0.01, s, 0.1)
optimizer = keras.optimizers.SGD(learning_rate)
```

This is nice and simple, plus when you save the model, the learning rate and its schedule (including its state) get saved as well. However, this approach is not part of the Keras API, it is specific to tf.keras.

To sum up, exponential decay or performance scheduling can considerably speed up convergence, so give them a try!

Avoiding Overfitting Through Regularization

> With four parameters I can fit an elephant and with five I can make him wiggle his trunk.
>
> —John von Neumann, *cited by Enrico Fermi in Nature 427*

With thousands of parameters you can fit the whole zoo. Deep neural networks typically have tens of thousands of parameters, sometimes even millions. With so many parameters, the network has an incredible amount of freedom and can fit a huge variety of complex datasets. But this great flexibility also means that it is prone to overfitting the training set. We need regularization.

We already implemented one of the best regularization techniques in Chapter 10: early stopping. Moreover, even though Batch Normalization was designed to solve the vanishing/exploding gradients problems, is also acts like a pretty good regularizer. In this section we will present other popular regularization techniques for neural networks: ℓ_1 and ℓ_2 regularization, dropout and max-norm regularization.

ℓ_1 and ℓ_2 Regularization

Just like you did in Chapter 4 for simple linear models, you can use ℓ_1 and ℓ_2 regularization to constrain a neural network's connection weights (but typically not its biases). Here is how to apply ℓ_2 regularization to a Keras layer's connection weights, using a regularization factor of 0.01:

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

The `l2()` function returns a regularizer that will be called to compute the regularization loss, at each step during training. This regularization loss is then added to the final loss. As you might expect, you can just use `keras.regularizers.l1()` if you

want ℓ_1 regularization, and if you want both ℓ_1 and ℓ_2 regularization, use `keras.regularizers.l1_l2()` (specifying both regularization factors).

Since you will typically want to apply the same regularizer to all layers in your network, as well as the same activation function and the same initialization strategy in all hidden layers, you may find yourself repeating the same arguments over and over. This makes it ugly and error-prone. To avoid this, you can try refactoring your code to use loops. Another option is to use Python's `functools.partial()` function: it lets you create a thin wrapper for any callable, with some default argument values. For example:

```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
])
```

Dropout

Dropout is one of the most popular regularization techniques for deep neural networks. It was proposed[23] by Geoffrey Hinton in 2012 and further detailed in a paper[24] by Nitish Srivastava et al., and it has proven to be highly successful: even the state-of-the-art neural networks got a 1–2% accuracy boost simply by adding dropout. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons, but always excluding the output neurons) has a probability p of being temporarily "dropped out," meaning it will be entirely ignored during this training step, but it may be active during the next step (see Figure 11-9). The hyperparameter p is called the *dropout rate*, and it is typically set to 50%. After training, neurons don't get dropped anymore. And that's all (except for a technical detail we will discuss momentarily).

23 "Improving neural networks by preventing co-adaptation of feature detectors," G. Hinton et al. (2012).

24 "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," N. Srivastava et al. (2014).

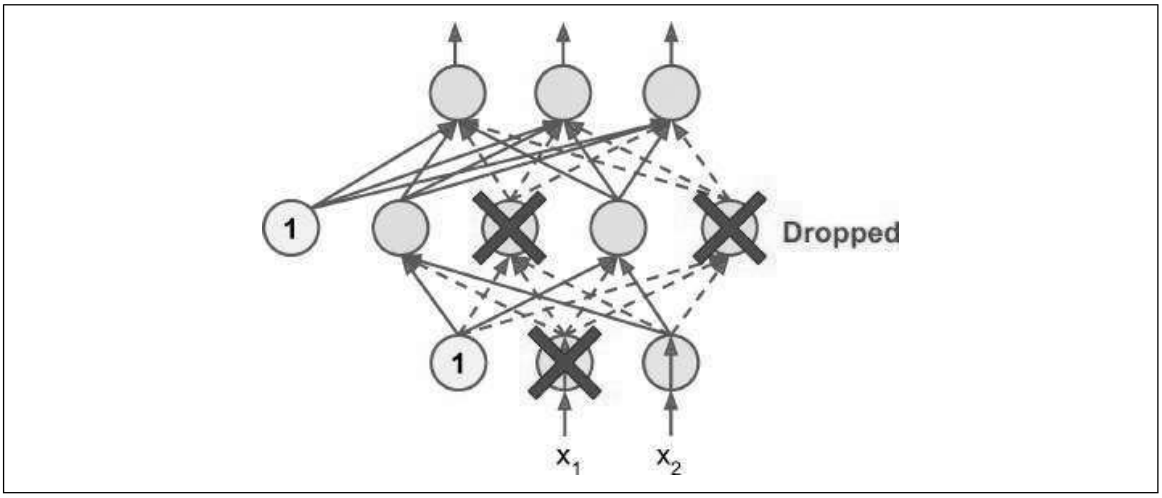

Figure 11-9. Dropout regularization

It is quite surprising at first that this rather brutal technique works at all. Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work? Well, who knows; perhaps it would! The company would obviously be forced to adapt its organization; it could not rely on any single person to fill in the coffee machine or perform any other critical tasks, so this expertise would have to be spread across several people. Employees would have to learn to cooperate with many of their coworkers, not just a handful of them. The company would become much more resilient. If one person quit, it wouldn't make much of a difference. It's unclear whether this idea would actually work for companies, but it certainly does for neural networks. Neurons trained with dropout cannot co-adapt with their neighboring neurons; they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons; they must pay attention to each of their input neurons. They end up being less sensitive to slight changes in the inputs. In the end you get a more robust network that generalizes better.

Another way to understand the power of dropout is to realize that a unique neural network is generated at each training step. Since each neuron can be either present or absent, there is a total of 2^N possible networks (where N is the total number of droppable neurons). This is such a huge number that it is virtually impossible for the same neural network to be sampled twice. Once you have run a 10,000 training steps, you have essentially trained 10,000 different neural networks (each with just one training instance). These neural networks are obviously not independent since they share many of their weights, but they are nevertheless all different. The resulting neural network can be seen as an averaging ensemble of all these smaller neural networks.

There is one small but important technical detail. Suppose $p = 50\%$, in which case during testing a neuron will be connected to twice as many input neurons as it was (on average) during training. To compensate for this fact, we need to multiply each

neuron's input connection weights by 0.5 after training. If we don't, each neuron will get a total input signal roughly twice as large as what the network was trained on, and it is unlikely to perform well. More generally, we need to multiply each input connection weight by the *keep probability* ($1 - p$) after training. Alternatively, we can divide each neuron's output by the keep probability during training (these alternatives are not perfectly equivalent, but they work equally well).

To implement dropout using Keras, you can use the `keras.layers.Dropout` layer. During training, it randomly drops some inputs (setting them to 0) and divides the remaining inputs by the keep probability. After training, it does nothing at all, it just passes the inputs to the next layer. For example, the following code applies dropout regularization before every `Dense` layer, using a dropout rate of 0.2:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

Since dropout is only active during training, the training loss is penalized compared to the validation loss, so comparing the two can be misleading. In particular, a model may be overfitting the training set and yet have similar training and validation losses. So make sure to evaluate the training loss without dropout (e.g., after training). Alternatively, you can call the `fit()` method inside a `with keras.backend.learning_phase_scope(1)` block: this will force dropout to be active during both training and validation.[25]

If you observe that the model is overfitting, you can increase the dropout rate. Conversely, you should try decreasing the dropout rate if the model underfits the training set. It can also help to increase the dropout rate for large layers, and reduce it for small ones. Moreover, many state-of-the-art architectures only use dropout after the last hidden layer, so you may want to try this if full dropout is too strong.

Dropout does tend to significantly slow down convergence, but it usually results in a much better model when tuned properly. So, it is generally well worth the extra time and effort.

25 This is specific to tf.keras, so you may prefer to use `keras.backend.set_learning_phase(1)` before calling the `fit()` method (and set it back to 0 right after).

If you want to regularize a self-normalizing network based on the SELU activation function (as discussed earlier), you should use `AlphaDropout`: this is a variant of dropout that preserves the mean and standard deviation of its inputs (it was introduced in the same paper as SELU, as regular dropout would break self-normalization).

Monte-Carlo (MC) Dropout

In 2016, a paper[26] by Yarin Gal and Zoubin Ghahramani added more good reasons to use dropout:

- First, the paper establishes a profound connection between dropout networks (i.e., neural networks containing a dropout layer before every weight layer) and approximate Bayesian inference[27], giving dropout a solid mathematical justification.
- Second, they introduce a powerful technique called *MC Dropout*, which can boost the performance of any trained dropout model, without having to retrain it or even modify it at all!
- Moreover, MC Dropout also provides a much better measure of the model's uncertainty.
- Finally, it is also amazingly simple to implement. If this all sounds like a "one weird trick" advertisement, then take a look at the following code. It is the full implementation of *MC Dropout*, boosting the dropout model we trained earlier, without retraining it:

```
with keras.backend.learning_phase_scope(1): # force training mode = dropout on
    y_probas = np.stack([model.predict(X_test_scaled)
                         for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

We first force training mode on, using a `learning_phase_scope(1)` context. This turns dropout on within the `with` block. Then we make 100 predictions over the test set, and we stack them. Since dropout is on, all predictions will be different. Recall that `predict()` returns a matrix with one row per instance, and one column per class. Since there are 10,000 instances in the test set, and 10 classes, this is a matrix of shape [10000, 10]. We stack 100 such matrices, so `y_probas` is an array of shape [100, 10000, 10]. Once we average over the first dimension (`axis=0`), we get `y_proba`, an array of shape [10000, 10], like we would get with a single prediction. That's all! Averaging

26 "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning," Y. Gal and Z. Ghahramani (2016).

27 Specifically, they show that training a dropout network is mathematically equivalent to approximate Bayesian inference in a specific type of probabilistic model called a *deep Gaussian Process*.

over multiple predictions with dropout on gives us a Monte Carlo estimate that is generally more reliable than the result of a single prediction with dropout off. For example, let's look at the model's prediction for the first instance in the test set, with dropout off:

```
>>> np.round(model.predict(X_test_scaled[:1]), 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.99]],
      dtype=float32)
```

The model seems almost certain that this image belongs to class 9 (ankle boot). Should you trust it? Is there really so little room for doubt? Compare this with the predictions made when dropout is activated:

```
>>> np.round(y_probas[:, :1], 2)
array([[[0.  , 0.  , 0.  , 0.  , 0.  , 0.14, 0.  , 0.17, 0.  , 0.68]],
       [[0.  , 0.  , 0.  , 0.  , 0.  , 0.16, 0.  , 0.2 , 0.  , 0.64]],
       [[0.  , 0.  , 0.  , 0.  , 0.  , 0.02, 0.  , 0.01, 0.  , 0.97]],
       [...]
```

This tells a very different story: apparently, when we activate dropout, the model is not sure anymore. It still seems to prefer class 9, but sometimes it hesitates with classes 5 (sandal) and 7 (sneaker), which makes sense given they're all footwear. Once we average over the first dimension, we get the following MC dropout predictions:

```
>>> np.round(y_proba[:1], 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.22, 0.  , 0.16, 0.  , 0.62]],
      dtype=float32)
```

The model still thinks this image belongs to class 9, but only with a 62% confidence, which seems much more reasonable than 99%. Plus it's useful to know exactly which other classes it thinks are likely. And you can also take a look at the standard deviation of the probability estimates:

```
>>> y_std = y_probas.std(axis=0)
>>> np.round(y_std[:1], 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.28, 0.  , 0.21, 0.02, 0.32]],
      dtype=float32)
```

Apparently there's quite a lot of variance in the probability estimates: if you were building a risk-sensitive system (e.g., a medical or financial system), you should probably treat such an uncertain prediction with extreme caution. You definitely would not treat it like a 99% confident prediction. Moreover, the model's accuracy got a small boost from 86.8 to 86.9:

```
>>> accuracy = np.sum(y_pred == y_test) / len(y_test)
>>> accuracy
0.8694
```

The number of Monte Carlo samples you use (100 in this example) is a hyperparameter you can tweak. The higher it is, the more accurate the predictions and their uncertainty estimates will be. However, it you double it, inference time will also be doubled. Moreover, above a certain number of samples, you will notice little improvement. So your job is to find the right tradeoff between latency and accuracy, depending on your application.

If your model contains other layers that behave in a special way during training (such as Batch Normalization layers), then you should not force training mode like we just did. Instead, you should replace the `Dropout` layers with the following `MCDropout` class:

```
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

We just sublass the `Dropout` layer and override the `call()` method to force its `training` argument to `True` (see Chapter 12). Similarly, you could define an `MCAlphaDropout` class by subclassing `AlphaDropout` instead. If you are creating a model from scratch, it's just a matter of using `MCDropout` rather than `Dropout`. But if you have a model that was already trained using `Dropout`, you need to create a new model, identical to the existing model except replacing the `Dropout` layers with `MCDropout`, then copy the existing model's weights to your new model.

In short, MC Dropout is a fantastic technique that boosts dropout models and provides better uncertainty estimates. And of course, since it is just regular dropout during training, it also acts like a regularizer.

Max-Norm Regularization

Another regularization technique that is quite popular for neural networks is called *max-norm regularization*: for each neuron, it constrains the weights $\mathbf{w}$ of the incoming connections such that $\| *w* \|_2 \leq \_r\_$, where r is the max-norm hyperparameter and $\| \cdot \|_2$ is the ℓ_2 norm.

Max-norm regularization does not add a regularization loss term to the overall loss function. Instead, it is typically implemented by computing $\|\mathbf{w}\|_2$ after each training step and clipping $\mathbf{w}$ if needed ($\mathbf{w} \leftarrow \mathbf{w}\frac{r}{\| \mathbf{w} \|_2}$).

Reducing r increases the amount of regularization and helps reduce overfitting. Max-norm regularization can also help alleviate the vanishing/exploding gradients problems (if you are not using Batch Normalization).

To implement max-norm regularization in Keras, just set every hidden layer's `kernel_constraint` argument to a `max_norm()` constraint, with the appropriate max value, for example:

```
keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal",
                   kernel_constraint=keras.constraints.max_norm(1.))
```

After each training iteration, the model's `fit()` method will call the object returned by `max_norm()`, passing it the layer's weights and getting clipped weights in return, which then replace the layer's weights. As we will see in Chapter 12, you can define your own custom constraint function if you ever need to, and use it as the `kernel_constraint`. You can also constrain the bias terms by setting the `bias_constraint` argument.

The `max_norm()` function has an `axis` argument that defaults to 0. A `Dense` layer usually has weights of shape [number of inputs, number of neurons], so using `axis=0` means that the max norm constraint will apply independently to each neuron's weight vector. If you want to use max-norm with convolutional layers (see Chapter 14), make sure to set the `max_norm()` constraint's `axis` argument appropriately (usually `axis=[0, 1, 2]`).

Summary and Practical Guidelines

In this chapter, we have covered a wide range of techniques and you may be wondering which ones you should use. The configuration in Table 11-2 will work fine in most cases, without requiring much hyperparameter tuning.

Table 11-2. Default DNN configuration

Hyperparameter	Default value
Kernel initializer:	LeCun initialization
Activation function:	SELU
Normalization:	None (self-normalization)
Regularization:	Early stopping
Optimizer:	Nadam
Learning rate schedule:	Performance scheduling

Don't forget to standardize the input features! Of course, you should also try to reuse parts of a pretrained neural network if you can find one that solves a similar problem, or use unsupervised pretraining if you have a lot of unlabeled data, or pretraining on an auxiliary task if you have a lot of labeled data for a similar task.

The default configuration in Table 11-2 may need to be tweaked:

- If your model self-normalizes:
  - If it overfits the training set, then you should add alpha dropout (and always use early stopping as well). Do not use other regularization methods, or else they would break self-normalization.
- If your model cannot self-normalize (e.g., it is a recurrent net or it contains skip connections):
  - You can try using ELU (or another activation function) instead of SELU, it may perform better. Make sure to change the initialization method accordingly (e.g., He init for ELU or ReLU).
  - If it is a deep network, you should use Batch Normalization after every hidden layer. If it overfits the training set, you can also try using max-norm or ℓ_2 regularization.
- If you need a sparse model, you can use ℓ_1 regularization (and optionally zero out the tiny weights after training). If you need an even sparser model, you can try using FTRL instead of Nadam optimization, along with ℓ_1 regularization. In any case, this will break self-normalization, so you will need to switch to BN if your model is deep.
- If you need a low-latency model (one that performs lightning-fast predictions), you may need to use less layers, avoid Batch Normalization, and possibly replace the SELU activation function with the leaky ReLU. Having a sparse model will also help. You may also want to reduce the float precision from 32-bits to 16-bit (or even 8-bits) (see ???).
- If you are building a risk-sensitive application, or inference latency is not very important in your application, you can use MC Dropout to boost performance and get more reliable probability estimates, along with uncertainty estimates.

With these guidelines, you are now ready to train very deep nets! I hope you are now convinced that you can go a very long way using just Keras. However, there may come a time when you need to have even more control, for example to write a custom loss function or to tweak the training algorithm. For such cases, you will need to use TensorFlow's lower-level API, as we will see in the next chapter.

Exercises

1. Is it okay to initialize all the weights to the same value as long as that value is selected randomly using He initialization?
2. Is it okay to initialize the bias terms to 0?
3. Name three advantages of the SELU activation function over ReLU.

4. In which cases would you want to use each of the following activation functions: SELU, leaky ReLU (and its variants), ReLU, tanh, logistic, and softmax?
5. What may happen if you set the `momentum` hyperparameter too close to 1 (e.g., 0.99999) when using an `SGD` optimizer?
6. Name three ways you can produce a sparse model.
7. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What are about MC dropout?
8. Deep Learning.
    a. Build a DNN with five hidden layers of 100 neurons each, He initialization, and the ELU activation function.
    b. Using Adam optimization and early stopping, try training it on MNIST but only on digits 0 to 4, as we will use transfer learning for digits 5 to 9 in the next exercise. You will need a softmax output layer with five neurons, and as always make sure to save checkpoints at regular intervals and save the final model so you can reuse it later.
    c. Tune the hyperparameters using cross-validation and see what precision you can achieve.
    d. Now try adding Batch Normalization and compare the learning curves: is it converging faster than before? Does it produce a better model?
    e. Is the model overfitting the training set? Try adding dropout to every layer and try again. Does it help?
9. Transfer learning.
    a. Create a new DNN that reuses all the pretrained hidden layers of the previous model, freezes them, and replaces the softmax output layer with a new one.
    b. Train this new DNN on digits 5 to 9, using only 100 images per digit, and time how long it takes. Despite this small number of examples, can you achieve high precision?
    c. Try caching the frozen layers, and train the model again: how much faster is it now?
    d. Try again reusing just four hidden layers instead of five. Can you achieve a higher precision?
    e. Now unfreeze the top two hidden layers and continue training: can you get the model to perform even better?
10. Pretraining on an auxiliary task.
    a. In this exercise you will build a DNN that compares two MNIST digit images and predicts whether they represent the same digit or not. Then you will reuse the lower layers of this network to train an MNIST classifier using very little

training data. Start by building two DNNs (let's call them DNN A and B), both similar to the one you built earlier but without the output layer: each DNN should have five hidden layers of 100 neurons each, He initialization, and ELU activation. Next, add one more hidden layer with 10 units on top of both DNNs. To do this, you should use a `keras.layers.Concatenate` layer to concatenate the outputs of both DNNs for each instance, then feed the result to the hidden layer. Finally, add an output layer with a single neuron using the logistic activation function.

b. Split the MNIST training set in two sets: split #1 should containing 55,000 images, and split #2 should contain contain 5,000 images. Create a function that generates a training batch where each instance is a pair of MNIST images picked from split #1. Half of the training instances should be pairs of images that belong to the same class, while the other half should be images from different classes. For each pair, the training label should be 0 if the images are from the same class, or 1 if they are from different classes.

c. Train the DNN on this training set. For each image pair, you can simultaneously feed the first image to DNN A and the second image to DNN B. The whole network will gradually learn to tell whether two images belong to the same class or not.

d. Now create a new DNN by reusing and freezing the hidden layers of DNN A and adding a softmax output layer on top with 10 neurons. Train this network on split #2 and see if you can achieve high performance despite having only 500 images per class.

Solutions to these exercises are available in ???.

CHAPTER 12

Custom Models and Training with TensorFlow

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 12 in the final release of the book.

So far we have used only TensorFlow's high level API, tf.keras, but it already got us pretty far: we built various neural network architectures, including regression and classification nets, wide & deep nets and self-normalizing nets, using all sorts of techniques, such as Batch Normalization, dropout, learning rate schedules, and more. In fact, 95% of the use cases you will encounter will not require anything else than tf.keras (and tf.data, see Chapter 13). But now it's time to dive deeper into TensorFlow and take a look at its lower-level Python API. This will be useful when you need extra control, to write custom loss functions, custom metrics, layers, models, initializers, regularizers, weight constraints and more. You may even need to fully control the training loop itself, for example to apply special transformations or constraints to the gradients (beyond just clipping them), or to use multiple optimizers for different parts of the network. We will cover all these cases in this chapter, then we will also look at how you can boost your custom models and training algorithms using TensorFlow's automatic graph generation feature. But first, let's take a quick tour of TensorFlow.

TensorFlow 2.0 was released in March 2019, making TensorFlow much easier to use. The first edition of this book used TF 1, while this edition uses TF 2.

A Quick Tour of TensorFlow

As you know, *TensorFlow* is a powerful library for numerical computation, particularly well suited and fine-tuned for large-scale Machine Learning (but you could use it for anything else that requires heavy computations). It was developed by the Google Brain team and it powers many of Google's large-scale services, such as Google Cloud Speech, Google Photos, and Google Search. It was open sourced in November 2015, and it is now the most popular deep learning library (in terms of citations in papers, adoption in companies, stars on github, etc.): countless projects use TensorFlow for all sorts of Machine Learning tasks, such as image classification, natural language processing (NLP), recommender systems, time series forecasting, and much more.

So what does TensorFlow actually offer? Here's a summary:

- Its core is very similar to NumPy, but with GPU support.
- It also supports distributed computing (across multiple devices and servers).
- It includes a kind of just-in-time (JIT) compiler that allows it to optimize computations for speed and memory usage: it works by extracting the *computation graph* from a Python function, then optimizing it (e.g., by pruning unused nodes) and finally running it efficiently (e.g., by automatically running independent operations in parallel).
- Computation graphs can be exported to a portable format, so you can train a TensorFlow model in one environment (e.g., using Python on Linux), and run it in another (e.g., using Java on an Android device).
- It implements autodiff (see Chapter 10 and ???), and provides some excellent optimizers, such as RMSProp, Nadam and FTRL (see Chapter 11), so you can easily minimize all sorts of loss functions.
- TensorFlow offers many more features, built on top of these core features: the most important is of course tf.keras[1], but it also has data loading & preprocessing ops (tf.data, tf.io, etc.), image processing ops (tf.image), signal processing ops (tf.signal), and more (see Figure 12-1 for an overview of TensorFlow's Python API).

1 TensorFlow also includes another Deep Learning API called the *Estimators API*, but it is now recommended to use tf.keras instead.

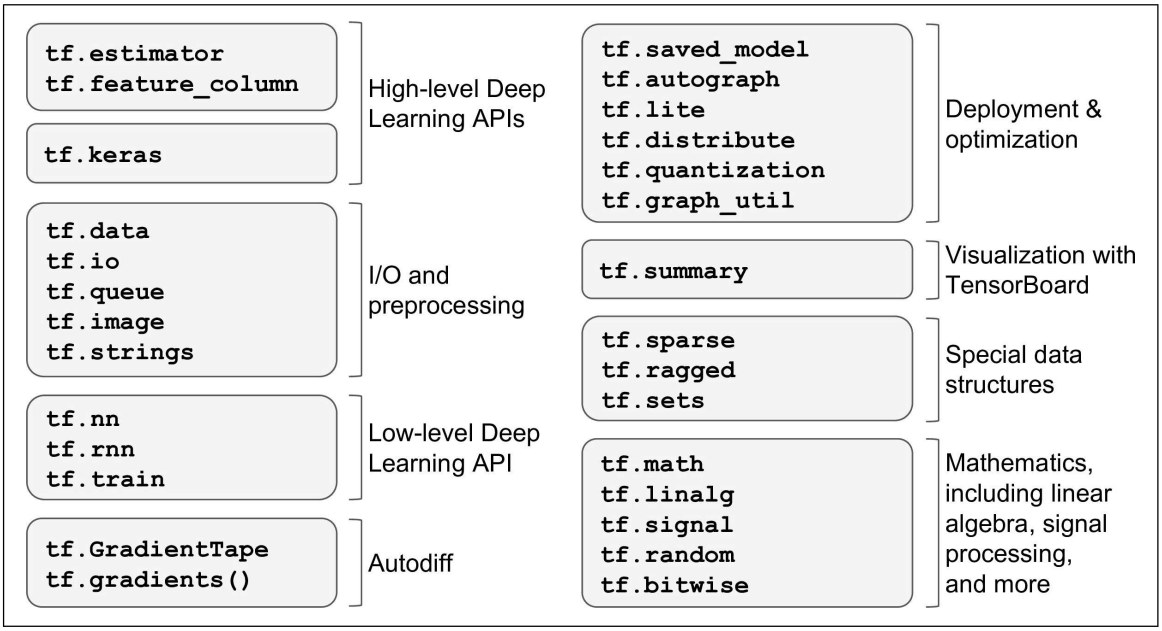

Figure 12-1. TensorFlow's Python API

We will cover many of the packages and functions of the TensorFlow API, but it's impossible to cover them all so you should really take some time to browse through the API: you will find that it is quite rich and well documented.

At the lowest level, each TensorFlow operation is implemented using highly efficient C++ code[2]. Many operations (or *ops* for short) have multiple implementations, called *kernels*: each kernel is dedicated to a specific device type, such as CPUs, GPUs, or even TPUs (*Tensor Processing Units*). As you may know, GPUs can dramatically speed up computations by splitting computations into many smaller chunks and running them in parallel across many GPU threads. TPUs are even faster. You can purchase your own GPU devices (for now, TensorFlow only supports Nvidia cards with CUDA Compute Capability 3.5+), but TPUs are only available on *Google Cloud Machine Learning Engine* (see ???).[3]

TensorFlow's architecture is shown in Figure 12-2: most of the time your code will use the high level APIs (especially tf.keras and tf.data), but when you need more flexibility you will use the lower level Python API, handling tensors directly. Note that APIs for other languages are also available. In any case, TensorFlow's execution

2 If you ever need to (but you probably won't), you can write your own operations using the C++ API.

3 If you are a researcher, you may be eligible to use these TPUs for free, see *https://tensorflow.org/tfrc/* for more details.

engine will take care of running the operations efficiently, even across multiple devices and machines if you tell it to.

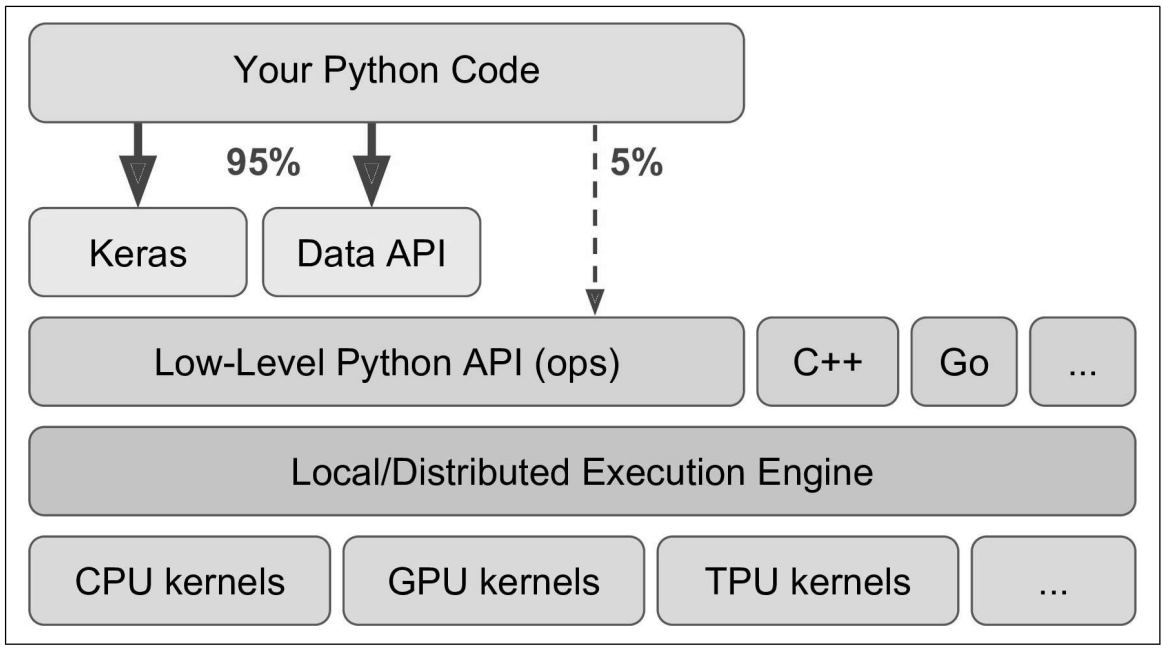

Figure 12-2. TensorFlow's architecture

TensorFlow runs not only on Windows, Linux, and MacOS, but also on mobile devices (using *TensorFlow Lite*), including both iOS and Android (see ???). If you do not want to use the Python API, there are also C++, Java, Go and Swift APIs. There is even a Javascript implementation called *TensorFlow.js* that makes it possible to run your models directly in your browser.

There's more to TensorFlow than just the library. TensorFlow is at the center of an extensive ecosystem of libraries. First, there's TensorBoard for visualization (see Chapter 10). Next, there's TensorFlow Extended (TFX), which is a set of libraries built by Google to productionize TensorFlow projects: it includes tools for data validation, preprocessing, model analysis and serving (with TF Serving, see ???). Google also launched *TensorFlow Hub*, a way to easily download and reuse pretrained neural networks. You can also get many neural network architectures, some of them pretrained, in TensorFlow's model garden. Check out the TensorFlow Resources, or *https://github.com/jtoy/awesome-tensorflow* for more TensorFlow-based projects. You will find hundreds of TensorFlow projects on GitHub, so it is often easy to find existing code for whatever you are trying to do.

More and more ML papers are released along with their implementation, and sometimes even with pretrained models. Check out *https://paperswithcode.com/* to easily find them.

Last but not least, TensorFlow has a dedicated team of passionate and helpful developers, and a large community contributing to improving it. To ask technical questions, you should use *http://stackoverflow.com/* and tag your question with *tensorflow* and *python*. You can file bugs and feature requests through GitHub. For general discussions, join the Google group.

Okay, it's time to start coding!

Using TensorFlow like NumPy

TensorFlow's API revolves around *tensors*, hence the name Tensor-Flow. A tensor is usually a multidimensional array (exactly like a NumPy `ndarray`), but it can also hold a scalar (a simple value, such as 42). These tensors will be important when we create custom cost functions, custom metrics, custom layers and more, so let's see how to create and manipulate them.

Tensors and Operations

You can easily create a tensor, using `tf.constant()`. For example, here is a tensor representing a matrix with two rows and three columns of floats:

```
>>> tf.constant([[1., 2., 3.], [4., 5., 6.]]) # matrix
<tf.Tensor: id=0, shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
>>> tf.constant(42) # scalar
<tf.Tensor: id=1, shape=(), dtype=int32, numpy=42>
```

Just like an `ndarray`, a `tf.Tensor` has a shape and a data type (`dtype`):

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> t.shape
TensorShape([2, 3])
>>> t.dtype
tf.float32
```

Indexing works much like in NumPy:

```
>>> t[:, 1:]
<tf.Tensor: id=5, shape=(2, 2), dtype=float32, numpy=
array([[2., 3.],
       [5., 6.]], dtype=float32)>
>>> t[..., 1, tf.newaxis]
<tf.Tensor: id=15, shape=(2, 1), dtype=float32, numpy=
array([[2.],
       [5.]], dtype=float32)>
```

Most importantly, all sorts of tensor operations are available:

```
>>> t + 10
<tf.Tensor: id=18, shape=(2, 3), dtype=float32, numpy=
```

```
array([[11., 12., 13.],
       [14., 15., 16.]], dtype=float32)>
>>> tf.square(t)
<tf.Tensor: id=20, shape=(2, 3), dtype=float32, numpy=
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)>
>>> t @ tf.transpose(t)
<tf.Tensor: id=24, shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

Note that writing `t + 10` is equivalent to calling `tf.add(t, 10)` (indeed, Python calls the magic method `t.__add__(10)`, which just calls `tf.add(t, 10)`). Other operators (like `-`, `*`, etc.) are also supported. The `@` operator was added in Python 3.5, for matrix multiplication: it is equivalent to calling the `tf.matmul()` function.

You will find all the basic math operations you need (e.g., `tf.add()`, `tf.multiply()`, `tf.square()`, `tf.exp()`, `tf.sqrt()`…), and more generally most operations that you can find in NumPy (e.g., `tf.reshape()`, `tf.squeeze()`, `tf.tile()`), but sometimes with a different name (e.g., `tf.reduce_mean()`, `tf.reduce_sum()`, `tf.reduce_max()`, `tf.math.log()` are the equivalent of `np.mean()`, `np.sum()`, `np.max()` and `np.log()`). When the name differs, there is often a good reason for it: for example, in TensorFlow you must write `tf.transpose(t)`, you cannot just write `t.T` like in NumPy. The reason is that it does not do exactly the same thing: in TensorFlow, a new tensor is created with its own copy of the transposed data, while in NumPy, `t.T` is just a transposed view on the same data. Similarly, the `tf.reduce_sum()` operation is named this way because its GPU kernel (i.e., GPU implementation) uses a reduce algorithm that does not guarantee the order in which the elements are added: because 32-bit floats have limited precision, this means that the result may change ever so slightly every time you call this operation. The same is true of `tf.reduce_mean()` (but of course `tf.reduce_max()` is deterministic).

Many functions and classes have aliases. For example, `tf.add()` and `tf.math.add()` are the same function. This allows TensorFlow to have concise names for the most common operations[4], while preserving well organized packages.

Keras' Low-Level API

The Keras API actually has its own low-level API, located in `keras.backend`. It includes functions like `square()`, `exp()`, `sqrt()` and so on. In tf.keras, these functions generally just call the corresponding TensorFlow operations. If you want to write code that will be portable to other Keras implementations, you should use these Keras functions. However, they only cover a subset of all functions available in TensorFlow, so in this book we will use the TensorFlow operations directly. Here is as simple example using `keras.backend`, which is commonly named `K` for short:

```
>>> from tensorflow import keras
>>> K = keras.backend
>>> K.square(K.transpose(t)) + 10
<tf.Tensor: id=39, shape=(3, 2), dtype=float32, numpy=
array([[11., 26.],
       [14., 35.],
       [19., 46.]], dtype=float32)>
```

Tensors and NumPy

Tensors play nice with NumPy: you can create a tensor from a NumPy array, and vice versa, and you can even apply TensorFlow operations to NumPy arrays and NumPy operations to tensors:

```
>>> a = np.array([2., 4., 5.])
>>> tf.constant(a)
<tf.Tensor: id=111, shape=(3,), dtype=float64, numpy=array([2., 4., 5.])>
>>> t.numpy() # or np.array(t)
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
>>> tf.square(a)
<tf.Tensor: id=116, shape=(3,), dtype=float64, numpy=array([4., 16., 25.])>
>>> np.square(t)
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)
```

4 A notable exception is `tf.math.log()` which is commonly used but there is no `tf.log()` alias (as it might be confused with logging).

Notice that NumPy uses 64-bit precision by default, while TensorFlow uses 32-bit. This is because 32-bit precision is generally more than enough for neural networks, plus it runs faster and uses less RAM. So when you create a tensor from a NumPy array, make sure to set `dtype=tf.float32`.

Type Conversions

Type conversions can significantly hurt performance, and they can easily go unnoticed when they are done automatically. To avoid this, TensorFlow does not perform any type conversions automatically: it just raises an exception if you try to execute an operation on tensors with incompatible types. For example, you cannot add a float tensor and an integer tensor, and you cannot even add a 32-bit float and a 64-bit float:

```
>>> tf.constant(2.) + tf.constant(40)
Traceback[...]InvalidArgumentError[...]expected to be a float[...]
>>> tf.constant(2.) + tf.constant(40., dtype=tf.float64)
Traceback[...]InvalidArgumentError[...]expected to be a double[...]
```

This may be a bit annoying at first, but remember that it's for a good cause! And of course you can use `tf.cast()` when you really need to convert types:

```
>>> t2 = tf.constant(40., dtype=tf.float64)
>>> tf.constant(2.0) + tf.cast(t2, tf.float32)
<tf.Tensor: id=136, shape=(), dtype=float32, numpy=42.0>
```

Variables

So far, we have used constant tensors: as their name suggests, you cannot modify them. However, the weights in a neural network need to be tweaked by backpropagation, and other parameters may also need to change over time (e.g., a momentum optimizer keeps track of past gradients). What we need is a `tf.Variable`:

```
>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
>>> v
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

A `tf.Variable` acts much like a constant tensor: you can perform the same operations with it, it plays nicely with NumPy as well, and it is just as picky with types. But it can also be modified in place using the `assign()` method (or `assign_add()` or `assign_sub()` which increment or decrement the variable by the given value). You can also modify individual cells (or slices), using the cell's (or slice's) `assign()` method (direct item assignment will not work), or using the `scatter_update()` or `scatter_nd_update()` methods:

```
v.assign(2 * v)          # => [[2., 4., 6.], [8., 10., 12.]]
v[0, 1].assign(42)       # => [[2., 42., 6.], [8., 10., 12.]]
```

```
v[:, 2].assign([0., 1.])  # => [[2., 42., 0.], [8., 10., 1.]]
v.scatter_nd_update(indices=[[0, 0], [1, 2]], updates=[100., 200.])
                          # => [[100., 42., 0.], [8., 10., 200.]]
```

In practice you will rarely have to create variables manually, since Keras provides an `add_weight()` method that will take care of it for you, as we will see. Moreover, model parameters will generally be updated directly by the optimizers, so you will rarely need to update variables manually.

Other Data Structures

TensorFlow supports several other data structures, including the following (please see the notebook or ??? for more details):

- *Sparse tensors* (`tf.SparseTensor`) efficiently represent tensors containing mostly 0s. The `tf.sparse` package contains operations for sparse tensors.
- *Tensor arrays* (`tf.TensorArray`) are lists of tensors. They have a fixed size by default, but can optionally be made dynamic. All tensors they contain must have the same shape and data type.
- *Ragged tensors* (`tf.RaggedTensor`) represent static lists of lists of tensors, where every tensor has the same shape and data type. The `tf.ragged` package contains operations for ragged tensors.
- *String tensors* are regular tensors of type `tf.string`. These actually represent byte strings, not Unicode strings, so if you create a string tensor using a Unicode string (e.g., a regular Python 3 string like `"café"`), then it will get encoded to UTF-8 automatically (e.g., `b"caf\xc3\xa9"`). Alternatively, you can represent Unicode strings using tensors of type `tf.int32`, where each item represents a Unicode codepoint (e.g., `[99, 97, 102, 233]`). The `tf.strings` package (with an `s`) contains ops for byte strings and Unicode strings (and to convert one into the other).
- *Sets* are just represented as regular tensors (or sparse tensors) containing one or more sets, and you can manipulate them using operations from the `tf.sets` package.
- *Queues*, including First In, First Out (FIFO) queues (`FIFOQueue`), queues that can prioritize some items (`PriorityQueue`), queues that shuffle their items (`RandomShuffleQueue`), and queues that can batch items of different shapes by padding (`PaddingFIFOQueue`). These classes are all in the `tf.queue` package.

With tensors, operations, variables and various data structures at your disposal, you are now ready to customize your models and training algorithms!

Customizing Models and Training Algorithms

Let's start by creating a custom loss function, which is a simple and common use case.

Custom Loss Functions

Suppose you want to train a regression model, but your training set is a bit noisy. Of course, you start by trying to clean up your dataset by removing or fixing the outliers, but it turns out to be insufficient, the dataset is still noisy. Which loss function should you use? The mean squared error might penalize large errors too much, so your model will end up being imprecise. The mean absolute error would not penalize outliers as much, but training might take a while to converge and the trained model might not be very precise. This is probably a good time to use the Huber loss (introduced in Chapter 10) instead of the good old MSE. The Huber loss is not currently part of the official Keras API, but it is available in tf.keras (just use an instance of the `keras.losses.Huber` class). But let's pretend it's not there: implementing it is easy as pie! Just create a function that takes the labels and predictions as arguments, and use TensorFlow operations to compute every instance's loss:

```
def huber_fn(y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) < 1
    squared_loss = tf.square(error) / 2
    linear_loss  = tf.abs(error) - 0.5
    return tf.where(is_small_error, squared_loss, linear_loss)
```

For better performance, you should use a vectorized implementation, as in this example. Moreover, if you want to benefit from TensorFlow's graph features, you should use only TensorFlow operations.

It is also preferable to return a tensor containing one loss per instance, rather than returning the mean loss. This way, Keras can apply class weights or sample weights when requested (see Chapter 10).

Next, you can just use this loss when you compile the Keras model, then train your model:

```
model.compile(loss=huber_fn, optimizer="nadam")
model.fit(X_train, y_train, [...])
```

And that's it! For each batch during training, Keras will call the `huber_fn()` function to compute the loss, and use it to perform a Gradient Descent step. Moreover, it will keep track of the total loss since the beginning of the epoch, and it will display the mean loss.

But what happens to this custom loss when we save the model?

Saving and Loading Models That Contain Custom Components

Saving a model containing a custom loss function actually works fine, as Keras just saves the name of the function. However, whenever you load it, you need to provide a dictionary that maps the function name to the actual function. More generally, when you load a model containing custom objects, you need to map the names to the objects:

```
model = keras.models.load_model("my_model_with_a_custom_loss.h5",
                                custom_objects={"huber_fn": huber_fn})
```

With the current implementation, any error between -1 and 1 is considered "small". But what if we want a different threshold? One solution is to create a function that creates a configured loss function:

```
def create_huber(threshold=1.0):
    def huber_fn(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < threshold
        squared_loss = tf.square(error) / 2
        linear_loss  = threshold * tf.abs(error) - threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    return huber_fn

model.compile(loss=create_huber(2.0), optimizer="nadam")
```

Unfortunately, when you save the model, the `threshold` will not be saved. This means that you will have to specify the `threshold` value when loading the model (note that the name to use is `"huber_fn"`, which is the name of the function we gave Keras, not the name of the function that created it):

```
model = keras.models.load_model("my_model_with_a_custom_loss_threshold_2.h5",
                                custom_objects={"huber_fn": create_huber(2.0)})
```

You can solve this by creating a subclass of the `keras.losses.Loss` class, and implement its `get_config()` method:

```
class HuberLoss(keras.losses.Loss):
    def __init__(self, threshold=1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)
    def call(self, y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < self.threshold
        squared_loss = tf.square(error) / 2
        linear_loss  = self.threshold * tf.abs(error) - self.threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

The Keras API only specifies how to use subclassing to define layers, models, callbacks, and regularizers. If you build other components (such as losses, metrics, initializers or constraints) using subclassing, they may not be portable to other Keras implementations.

Let's walk through this code:

- The constructor accepts `**kwargs` and passes them to the parent constructor, which handles standard hyperparameters: the `name` of the loss and the `reduction` algorithm to use to aggregate the individual instance losses. By default, it is `"sum_over_batch_size"`, which means that the loss will be the sum of the instance losses, possibly weighted by the sample weights, if any, and then divide the result by the batch size (not by the sum of weights, so this is *not* the weighted mean).[5]. Other possible values are `"sum"` and `None`.
- The `call()` method takes the labels and predictions, computes all the instance losses, and returns them.
- The `get_config()` method returns a dictionary mapping each hyperparameter name to its value. It first calls the parent class's `get_config()` method, then adds the new hyperparameters to this dictionary (note that the convenient `{**x}` syntax was added in Python 3.5).

You can then use any instance of this class when you compile the model:

```
model.compile(loss=HuberLoss(2.), optimizer="nadam")
```

When you save the model, the threshold will be saved along with it, and when you load the model you just need to map the class name to the class itself:

```
model = keras.models.load_model("my_model_with_a_custom_loss_class.h5",
                                custom_objects={"HuberLoss": HuberLoss})
```

When you save a model, Keras calls the loss instance's `get_config()` method and saves the config as JSON in the HDF5 file. When you load the model, it calls the `from_config()` class method on the `HuberLoss` class: this method is implemented by the base class (`Loss`) and just creates an instance of the class, passing `**config` to the constructor.

That's it for losses! It was not too hard, was it? Well it's just as simple for custom activation functions, initializers, regularizers, and constraints. Let's look at these now.

5 It would not be a good idea to use a weighted mean: if we did, then two instances with the same weight but in different batches would have a different impact on training, depending on the total weight of each batch.

Custom Activation Functions, Initializers, Regularizers, and Constraints

Most Keras functionalities, such as losses, regularizers, constraints, initializers, metrics, activation functions, layers and even full models can be customized in very much the same way. Most of the time, you will just need to write a simple function, with the appropriate inputs and outputs. For example, here are examples of a custom activation function (equivalent to `keras.activations.softplus` or `tf.nn.softplus`), a custom Glorot initializer (equivalent to `keras.initializers.glorot_normal`), a custom ℓ_1 regularizer (equivalent to `keras.regularizers.l1(0.01)`) and a custom constraint that ensures weights are all positive (equivalent to `keras.constraints.nonneg()` or `tf.nn.relu`):

```
def my_softplus(z): # return value is just tf.nn.softplus(z)
    return tf.math.log(tf.exp(z) + 1.0)

def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)

def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))

def my_positive_weights(weights): # return value is just tf.nn.relu(weights)
    return tf.where(weights < 0., tf.zeros_like(weights), weights)
```

As you can see, the arguments depend on the type of custom function. These custom functions can then be used normally, for example:

```
layer = keras.layers.Dense(30, activation=my_softplus,
                           kernel_initializer=my_glorot_initializer,
                           kernel_regularizer=my_l1_regularizer,
                           kernel_constraint=my_positive_weights)
```

The activation function will be applied to the output of this `Dense` layer, and its result will be passed on to the next layer. The layer's weights will be initialized using the value returned by the initializer. At each training step the weights will be passed to the regularization function to compute the regularization loss, which will be added to the main loss to get the final loss used for training. Finally, the constraint function will be called after each training step, and the layer's weights will be replaced by the constrained weights.

If a function has some hyperparameters that need to be saved along with the model, then you will want to subclass the appropriate class, such as `keras.regularizers.Regularizer`, `keras.constraints.Constraint`, `keras.initializers.Initializer` or `keras.layers.Layer` (for any layer, including activation functions). For example, much like we did for the custom loss, here is a simple class for ℓ_1 regulariza-

tion, that saves its `factor` hyperparameter (this time we do not need to call the parent constructor or the `get_config()` method, as they are not defined by the parent class):

```
class MyL1Regularizer(keras.regularizers.Regularizer):
    def __init__(self, factor):
        self.factor = factor
    def __call__(self, weights):
        return tf.reduce_sum(tf.abs(self.factor * weights))
    def get_config(self):
        return {"factor": self.factor}
```

Note that you must implement the `call()` method for losses, layers (including activation functions) and models, or the `__call__()` method for regularizers, initializers and constraints. For metrics, things are a bit different, as we will see now.

Custom Metrics

Losses and metrics are conceptually not the same thing: losses are used by Gradient Descent to *train* a model, so they must be differentiable (at least where they are evaluated) and their gradients should not be 0 everywhere. Plus, it's okay if they are not easily interpretable by humans (e.g. cross-entropy). In contrast, metrics are used to *evaluate* a model, they must be more easily interpretable, and they can be non-differentiable or have 0 gradients everywhere (e.g., accuracy).

That said, in most cases, defining a custom metric function is exactly the same as defining a custom loss function. In fact, we could even use the Huber loss function we created earlier as a metric[6], it would work just fine (and persistence would also work the same way, in this case only saving the name of the function, `"huber_fn"`):

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])
```

For each batch during training, Keras will compute this metric and keep track of its mean since the beginning of the epoch. Most of the time, this is exactly what you want. But not always! Consider a binary classifier's precision, for example. As we saw in Chapter 3, precision is the number of true positives divided by the number of positive predictions (including both true positives and false positives). Suppose the model made 5 positive predictions in the first batch, 4 of which were correct: that's 80% precision. Then suppose the model made 3 positive predictions in the second batch, but they were all incorrect: that's 0% precision for the second batch. If you just compute the mean of these two precisions, you get 40%. But wait a second, this is *not* the model's precision over these two batches! Indeed, there were a total of 4 true positives (4 + 0) out of 8 positive predictions (5 + 3), so the overall precision is 50%, not 40%. What we need is an object that can keep track of the number of true positives and the num-

6 However, the Huber loss is seldom used as a metric (the MAE or MSE are preferred).

ber of false positives, and compute their ratio when requested. This is precisely what the `keras.metrics.Precision` class does:

```
>>> precision = keras.metrics.Precision()
>>> precision([0, 1, 1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1])
<tf.Tensor: id=581729, shape=(), dtype=float32, numpy=0.8>
>>> precision([0, 1, 0, 0, 1, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0])
<tf.Tensor: id=581780, shape=(), dtype=float32, numpy=0.5>
```

In this example, we created a `Precision` object, then we used it like a function, passing it the labels and predictions for the first batch, then for the second batch (note that we could also have passed sample weights). We used the same number of true and false positives as in the example we just discussed. After the first batch, it returns the precision of 80%, then after the second batch it returns 50% (which is the overall precision so far, not the second batch's precision). This is called a *streaming metric* (or *stateful metric*), as it is gradually updated, batch after batch.

At any point, we can call the `result()` method to get the current value of the metric. We can also look at its variables (tracking the number of true and false positives) using the `variables` attribute, and reset these variables using the `reset_states()` method:

```
>>> p.result()
<tf.Tensor: id=581794, shape=(), dtype=float32, numpy=0.5>
>>> p.variables
[<tf.Variable 'true_positives:0' [...] numpy=array([4.], dtype=float32)>,
 <tf.Variable 'false_positives:0' [...] numpy=array([4.], dtype=float32)>]
>>> p.reset_states() # both variables get reset to 0.0
```

If you need to create such a streaming metric, you can just create a subclass of the `keras.metrics.Metric` class. Here is a simple example that keeps track of the total Huber loss and the number of instances seen so far. When asked for the result, it returns the ratio, which is simply the mean Huber loss:

```
class HuberMetric(keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs) # handles base args (e.g., dtype)
        self.threshold = threshold
        self.huber_fn = create_huber(threshold)
        self.total = self.add_weight("total", initializer="zeros")
        self.count = self.add_weight("count", initializer="zeros")
    def update_state(self, y_true, y_pred, sample_weight=None):
        metric = self.huber_fn(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(metric))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))
    def result(self):
        return self.total / self.count
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

Let's walk through this code:[7]:

- The constructor uses the `add_weight()` method to create the variables needed to keep track of the metric's state over multiple batches, in this case the sum of all Huber losses (`total`) and the number of instances seen so far (`count`). You could just create variables manually if you preferred. Keras tracks any `tf.Variable` that is set as an attribute (and more generally, any "trackable" object, such as layers or models).
- The `update_state()` method is called when you use an instance of this class as a function (as we did with the `Precision` object). It updates the variables given the labels and predictions for one batch (and sample weights, but in this case we just ignore them).
- The `result()` method computes and returns the final result, in this case just the mean Huber metric over all instances. When you use the metric as a function, the `update_state()` method gets called first, then the `result()` method is called, and its output is returned.
- We also implement the `get_config()` method to ensure the `threshold` gets saved along with the model.
- The default implementation of the `reset_states()` method just resets all variables to 0.0 (but you can override it if needed).

Keras will take care of variable persistence seamlessly, no action is required.

When you define a metric using a simple function, Keras automatically calls it for each batch, and it keeps track of the mean during each epoch, just like we did manually. So the only benefit of our `HuberMetric` class is that the `threshold` will be saved. But of course, some metrics, like precision, cannot simply be averaged over batches: in thoses cases, there's no other option than to implement a streaming metric.

Now that we have built a streaming metric, building a custom layer will seem like a walk in the park!

7 This class is for illustration purposes only. A simpler and better implementation would just subclass the `keras.metrics.Mean` class, see the notebook for an example.

Custom Layers

You may occasionally want to build an architecture that contains an exotic layer for which TensorFlow does not provide a default implementation. In this case, you will need to create a custom layer. Or sometimes you may simply want to build a very repetitive architecture, containing identical blocks of layers repeated many times, and it would be convenient to treat each block of layers as a single layer. For example, if the model is a sequence of layers A, B, C, A, B, C, A, B, C, then you might want to define a custom layer D containing layers A, B, C, and your model would then simply be D, D, D. Let's see how to build custom layers.

First, some layers have no weights, such as `keras.layers.Flatten` or `keras.layers.ReLU`. If you want to create a custom layer without any weights, the simplest option is to write a function and wrap it in a `keras.layers.Lambda` layer. For example, the following layer will apply the exponential function to its inputs:

```
exponential_layer = keras.layers.Lambda(lambda x: tf.exp(x))
```

This custom layer can then be used like any other layer, using the sequential API, the functional API, or the subclassing API. You can also use it as an activation function (or you could just use `activation=tf.exp`, or `activation=keras.activations.exponential`, or simply `activation="exponential"`). The exponential layer is sometimes used in the output layer of a regression model when the values to predict have very different scales (e.g., 0.001, 10., 1000.).

As you probably guessed by now, to build a custom stateful layer (i.e., a layer with weights), you need to create a subclass of the `keras.layers.Layer` class. For example, the following class implements a simplified version of the `Dense` layer:

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

    def build(self, batch_input_shape):
        self.kernel = self.add_weight(
            name="kernel", shape=[batch_input_shape[-1], self.units],
            initializer="glorot_normal")
        self.bias = self.add_weight(
            name="bias", shape=[self.units], initializer="zeros")
        super().build(batch_input_shape) # must be at the end

    def call(self, X):
        return self.activation(X @ self.kernel + self.bias)

    def compute_output_shape(self, batch_input_shape):
        return tf.TensorShape(batch_input_shape.as_list()[:-1] + [self.units])
```

```
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "units": self.units,
                "activation": keras.activations.serialize(self.activation)}
```

Let's walk through this code:

- The constructor takes all the hyperparameters as arguments (in this example just `units` and `activation`), and importantly it also takes a `**kwargs` argument. It calls the parent constructor, passing it the `kwargs`: this takes care of standard arguments such as `input_shape`, `trainable`, `name`, and so on. Then it saves the hyperparameters as attributes, converting the `activation` argument to the appropriate activation function using the `keras.activations.get()` function (it accepts functions, standard strings like `"relu"` or `"selu"`, or simply `None`)[8].
- The `build()` method's role is to create the layer's variables, by calling the `add_weight()` method for each weight. The `build()` method is called the first time the layer is used. At that point, Keras will know the shape of this layer's inputs, and it will pass it to the `build()` method[9], which is often necessary to create some of the weights. For example, we need to know the number of neurons in the previous layer in order to create the connection weights matrix (i.e., the `"kernel"`): this corresponds to the size of the last dimension of the inputs. At the end of the `build()` method (and only at the end), you must call the parent's `build()` method: this tells Keras that the layer is built (it just sets `self.built = True`).
- The `call()` method actually performs the desired operations. In this case, we compute the matrix multiplication of the inputs `X` and the layer's kernel, we add the bias vector, we apply the activation function to the result, and this gives us the output of the layer.
- The `compute_output_shape()` method simply returns the shape of this layer's outputs. In this case, it is the same shape as the inputs, except the last dimension is replaced with the number of neurons in the layer. Note that in tf.keras, shapes are instances of the `tf.TensorShape` class, which you can convert to Python lists using `as_list()`.
- The `get_config()` method is just like earlier. Note that we save the activation function's full configuration by calling `keras.activations.serialize()`.

You can now use a `MyDense` layer just like any other layer!

8 This function is specific to tf.keras. You could use `keras.activations.Activation` instead.

9 The Keras API calls this argument `input_shape`, but since it also includes the batch dimension, I prefer to call it `batch_input_shape`. Same for `compute_output_shape()`.

You can generally omit the `compute_output_shape()` method, as tf.keras automatically infers the output shape, except when the layer is dynamic (as we will see shortly). In other Keras implementations, this method is either required or by default it assumes the output shape is the same as the input shape.

To create a layer with multiple inputs (e.g., `Concatenate`), the argument to the `call()` method should be a tuple containing all the inputs, and similarly the argument to the `compute_output_shape()` method should be a tuple containing each input's batch shape. To create a layer with multiple outputs, the `call()` method should return the list of outputs, and the `compute_output_shape()` should return the list of batch output shapes (one per output). For example, the following toy layer takes two inputs and returns three outputs:

```
class MyMultiLayer(keras.layers.Layer):
    def call(self, X):
        X1, X2 = X
        return [X1 + X2, X1 * X2, X1 / X2]

    def compute_output_shape(self, batch_input_shape):
        b1, b2 = batch_input_shape
        return [b1, b1, b1] # should probably handle broadcasting rules
```

This layer may now be used like any other layer, but of course only using the functional and subclassing APIs, not the sequential API (which only accepts layers with one input and one output).

If your layer needs to have a different behavior during training and during testing (e.g., if it uses `Dropout` or `BatchNormalization` layers), then you must add a `training` argument to the `call()` method and use this argument to decide what to do. For example, let's create a layer that adds Gaussian noise during training (for regularization), but does nothing during testing (Keras actually has a layer that does the same thing: `keras.layers.GaussianNoise`):

```
class MyGaussianNoise(keras.layers.Layer):
    def __init__(self, stddev, **kwargs):
        super().__init__(**kwargs)
        self.stddev = stddev

    def call(self, X, training=None):
        if training:
            noise = tf.random.normal(tf.shape(X), stddev=self.stddev)
            return X + noise
        else:
            return X

    def compute_output_shape(self, batch_input_shape):
        return batch_input_shape
```

With that, you can now build any custom layer you need! Now let's create custom models.

Custom Models

We already looked at custom model classes in Chapter 10 when we discussed the subclassing API.[10] It is actually quite straightforward, just subclass the `keras.models.Model` class, create layers and variables in the constructor, and implement the `call()` method to do whatever you want the model to do. For example, suppose you want to build the model represented in Figure 12-3:

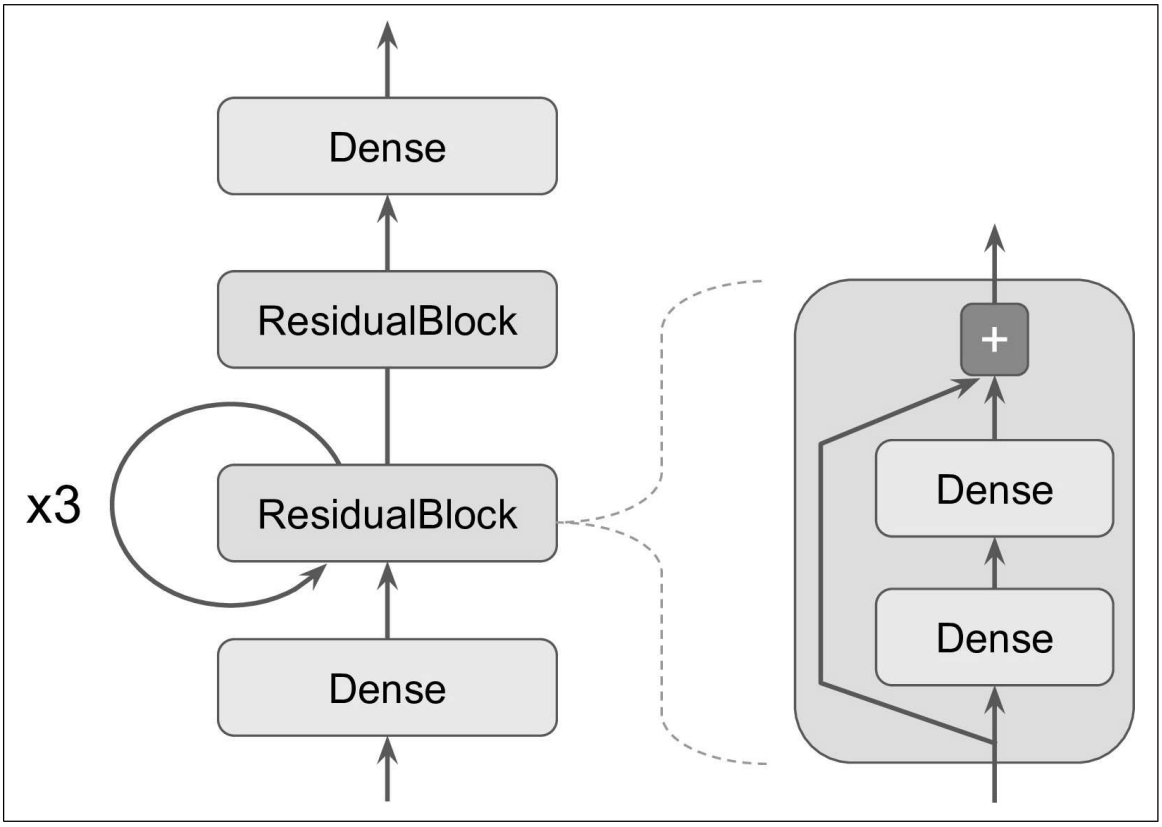

Figure 12-3. Custom Model Example

The inputs go through a first dense layer, then through a *residual block* composed of two dense layers and an addition operation (as we will see in Chapter 14, a residual block adds its inputs to its outputs), then through this same residual block 3 more times, then through a second residual block, and the final result goes through a dense output layer. Note that this model does not make much sense, it's just an example to illustrate the fact that you can easily build any kind of model you want, even contain-

10 The name "subclassing API" usually refers only to the creation of custom models by subclassing, although many other things can be created by subclassing, as we saw in this chapter.

ing loops and skip connections. To implement this model, it is best to first create a `ResidualBlock` layer, since we are going to create a couple identical blocks (and we might want to reuse it in another model):

```
class ResidualBlock(keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(n_neurons, activation="elu",
                                          kernel_initializer="he_normal")
                       for _ in range(n_layers)]

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        return inputs + Z
```

This layer is a bit special since it contains other layers. This is handled transparently by Keras: it automatically detects that the `hidden` attribute contains trackable objects (layers in this case), so their variables are automatically added to this layer's list of variables. The rest of this class is self-explanatory. Next, let's use the subclassing API to define the model itself:

```
class ResidualRegressor(keras.models.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.Dense(30, activation="elu",
                                          kernel_initializer="he_normal")
        self.block1 = ResidualBlock(2, 30)
        self.block2 = ResidualBlock(2, 30)
        self.out = keras.layers.Dense(output_dim)

    def call(self, inputs):
        Z = self.hidden1(inputs)
        for _ in range(1 + 3):
            Z = self.block1(Z)
        Z = self.block2(Z)
        return self.out(Z)
```

We create the layers in the constructor, and use them in the `call()` method. This model can then be used like any other model (compile it, fit it, evaluate it and use it to make predictions). If you also want to be able to save the model using the `save()` method, and load it using the `keras.models.load_model()` function, you must implement the `get_config()` method (as we did earlier) in both the `ResidualBlock` class and the `ResidualRegressor` class. Alternatively, you can just save and load the weights using the `save_weights()` and `load_weights()` methods.

The `Model` class is actually a subclass of the `Layer` class, so models can be defined and used exactly like layers. But a model also has some extra functionalities, including of course its `compile()`, `fit()`, `evaluate()` and `predict()` methods (and a few var-

iants, such as `train_on_batch()` or `fit_generator()`), plus the `get_layers()` method (which can return any of the model's layers by name or by index), and the `save()` method (and support for `keras.models.load_model()` and `keras.models.clone_model()`). So if models provide more functionalities than layers, why not just define every layer as a model? Well, technically you could, but it is probably cleaner to distinguish the internal components of your model (layers or reusable blocks of layers) from the model itself. The former should subclass the `Layer` class, while the latter should subclass the `Model` class.

With that, you can quite naturally and concisely build almost any model that you find in a paper, either using the sequential API, the functional API, the subclassing API, or even a mix of these. "Almost" any model? Yes, there are still a couple things that we need to look at: first, how to define losses or metrics based on model internals, and second how to build a custom training loop.

Losses and Metrics Based on Model Internals

The custom losses and metrics we defined earlier were all based on the labels and the predictions (and optionally sample weights). However, you will occasionally want to define losses based on other parts of your model, such as the weights or activations of its hidden layers. This may be useful for regularization purposes, or to monitor some internal aspect of your model.

To define a custom loss based on model internals, just compute it based on any part of the model you want, then pass the result to the `add_loss()` method. For example, the following custom model represents a standard MLP regressor with 5 hidden layers, except it also implements a *reconstruction loss* (see ???): we add an extra `Dense` layer on top of the last hidden layer, and its role is to try to reconstruct the inputs of the model. Since the reconstruction must have the same shape as the model's inputs, we need to create this `Dense` layer in the `build()` method to have access to the shape of the inputs. In the `call()` method, we compute both the regular output of the MLP, plus the output of the reconstruction layer. We then compute the mean squared difference between the reconstructions and the inputs, and we add this value (times 0.05) to the model's list of losses by calling `add_loss()`. During training, Keras will add this loss to the main loss (which is why we scaled down the reconstruction loss, to ensure the main loss dominates). As a result, the model will be forced to preserve as much information as possible through the hidden layers, even information that is not directly useful for the regression task itself. In practice, this loss sometimes improves generalization; it is a regularization loss:

```
class ReconstructingRegressor(keras.models.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(30, activation="selu",
                                          kernel_initializer="lecun_normal")
```

```
                    for _ in range(5)]
        self.out = keras.layers.Dense(output_dim)

    def build(self, batch_input_shape):
        n_inputs = batch_input_shape[-1]
        self.reconstruct = keras.layers.Dense(n_inputs)
        super().build(batch_input_shape)

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        reconstruction = self.reconstruct(Z)
        recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
        self.add_loss(0.05 * recon_loss)
        return self.out(Z)
```

Similarly, you can add a custom metric based on model internals by computing it in any way you want, as long at the result is the output of a metric object. For example, you can create a `keras.metrics.Mean()` object in the constructor, then call it in the `call()` method, passing it the `recon_loss`, and finally add it to the model by calling the model's `add_metric()` method. This way, when you train the model, Keras will display both the mean loss over each epoch (the loss is the sum of the main loss plus 0.05 times the reconstruction loss) and the mean reconstruction error over each epoch. Both will go down during training:

```
Epoch 1/5
11610/11610 [==============] [...] loss: 4.3092 - reconstruction_error: 1.7360
Epoch 2/5
11610/11610 [==============] [...] loss: 1.1232 - reconstruction_error: 0.8964
[...]
```

In over 99% of the cases, everything we have discussed so far will be sufficient to implement whatever model you want to build, even with complex architectures, losses, metrics, and so on. However, in some rare cases you may need to customize the training loop itself. However, before we get there, we need to look at how to compute gradients automatically in TensorFlow.

Computing Gradients Using Autodiff

To understand how to use autodiff (see Chapter 10 and ???) to compute gradients automatically, let's consider a simple toy function:

```
def f(w1, w2):
    return 3 * w1 ** 2 + 2 * w1 * w2
```

If you know calculus, you can analytically find that the partial derivative of this function with regards to `w1` is `6 * w1 + 2 * w2`. You can also find that its partial derivative with regards to `w2` is `2 * w1`. For example, at the point `(w1, w2) = (5, 3)`, these par-

tial derivatives are equal to 36 and 10, respectively, so the gradient vector at this point is (36, 10). But if this were a neural network, the function would be much more complex, typically with tens of thousands of parameters, and finding the partial derivatives analytically by hand would be an almost impossible task. One solution could be to compute an approximation of each partial derivative by measuring how much the function's output changes when you tweak the corresponding parameter:

```
>>> w1, w2 = 5, 3
>>> eps = 1e-6
>>> (f(w1 + eps, w2) - f(w1, w2)) / eps
36.000003007075065
>>> (f(w1, w2 + eps) - f(w1, w2)) / eps
10.000000003174137
```

Looks about right! This works rather well and it is trivial to implement, but it is just an approximation, and importantly you need to call `f()` at least once per parameter (not twice, since we could compute `f(w1, w2)` just once). This makes this approach intractable for large neural networks. So instead we should use autodiff (see Chapter 10 and ???). TensorFlow makes this pretty simple:

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
```

We first define two variables `w1` and `w2`, then we create a `tf.GradientTape` context that will automatically record every operation that involves a variable, and finally we ask this tape to compute the gradients of the result `z` with regards to both variables `[w1, w2]`. Let's take a look at the gradients that TensorFlow computed:

```
>>> gradients
[<tf.Tensor: id=828234, shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: id=828229, shape=(), dtype=float32, numpy=10.0>]
```

Perfect! Not only is the result accurate (the precision is only limited by the floating point errors), but the `gradient()` method only goes through the recorded computations once (in reverse order), no matter how many variables there are, so it is incredibly efficient. It's like magic!

Only put the strict minimum inside the `tf.GradientTape()` block, to save memory. Alternatively, you can pause recording by creating a `with tape.stop_recording()` block inside the `tf.GradientTape()` block.

The tape is automatically erased immediately after you call its `gradient()` method, so you will get an exception if you try to call `gradient()` twice:

```
with tf.GradientTape() as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) # => tensor 36.0
dz_dw2 = tape.gradient(z, w2) # RuntimeError!
```

If you need to call `gradient()` more than once, you must make the tape persistent, and delete it when you are done with it to free resources:

```
with tf.GradientTape(persistent=True) as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) # => tensor 36.0
dz_dw2 = tape.gradient(z, w2) # => tensor 10.0, works fine now!
del tape
```

By default, the tape will only track operations involving variables, so if you try to compute the gradient of `z` with regards to anything else than a variable, the result will be `None`:

```
c1, c2 = tf.constant(5.), tf.constant(3.)
with tf.GradientTape() as tape:
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # returns [None, None]
```

However, you can force the tape to watch any tensors you like, to record every operation that involves them. You can then compute gradients with regards to these tensors, as if they were variables:

```
with tf.GradientTape() as tape:
    tape.watch(c1)
    tape.watch(c2)
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # returns [tensor 36., tensor 10.]
```

This can be useful in some cases, for example if you want to implement a regularization loss that penalizes activations that vary a lot when the inputs vary little: the loss will be based on the gradient of the activations with regards to the inputs. Since the inputs are not variables, you would need to tell the tape to watch them.

If you compute the gradient of a list of tensors (e.g., `[z1, z2, z3]`) with regards to some variables (e.g., `[w1, w2]`), TensorFlow actually efficiently computes the sum of the gradients of these tensors (i.e., `gradient(z1, [w1, w2])`, plus `gradient(z2, [w1, w2])`, plus `gradient(z3, [w1, w2])`). Due to the way reverse-mode autodiff works, it is not possible to compute the individual gradients (`z1`, `z2` and `z3`) without actually calling `gradient()` multiple times (once for `z1`, once for `z2` and once for `z3`), which requires making the tape persistent (and deleting it afterwards).

Moreover, it is actually possible to compute second order partial derivatives (the Hessians, i.e., the partial derivatives of the partial derivatives)! To do this, we need to record the operations that are performed when computing the first-order partial derivatives (the Jacobians): this requires a second tape. Here is how it works:

```
with tf.GradientTape(persistent=True) as hessian_tape:
    with tf.GradientTape() as jacobian_tape:
        z = f(w1, w2)
    jacobians = jacobian_tape.gradient(z, [w1, w2])
hessians = [hessian_tape.gradient(jacobian, [w1, w2])
            for jacobian in jacobians]
del hessian_tape
```

The inner tape is used to compute the Jacobians, as we did earlier. The outer tape is used to compute the partial derivatives of each Jacobian. Since we need to call `gradient()` once for each Jacobian (or else we would get the sum of the partial derivatives over all the Jabobians, as explained earlier), we need the outer tape to be persistent, so we delete it at the end. The Jacobians are obviously the same as earlier (36 and 5), but now we also have the Hessians:

```
>>> hessians # dz_dw1_dw1, dz_dw1_dw2, dz_dw2_dw1, dz_dw2_dw2
[[<tf.Tensor: id=830578, shape=(), dtype=float32, numpy=6.0>,
  <tf.Tensor: id=830595, shape=(), dtype=float32, numpy=2.0>],
 [<tf.Tensor: id=830600, shape=(), dtype=float32, numpy=2.0>, None]]
```

Let's verify these Hessians. The first two are the partial derivatives of `6 * w1 + 2 * w2` (which is, as we saw earlier, the partial derivative of `f` with regards to `w1`), with regards to `w1` and `w2`. The result is correct: 6 for `w1` and 2 for `w2`. The next two are the partial derivatives of `2 * w1` (the partial derivative of `f` with regards to `w2`), with regards to `w1` and `w2`, which are 2 for `w1` and 0 for `w2`. Note that TensorFlow returns `None` instead of 0 since `w2` does not appear at all in `2 * w1`. TensorFlow also returns `None` when you use an operation whose gradients are not defined (e.g., `tf.argmax()`).

In some rare cases you may want to stop gradients from backpropagating through some part of your neural network. To do this, you must use the `tf.stop_gradient()` function: it just returns its inputs during the forward pass (like `tf.identity()`), but it does not let gradients through during backpropagation (it acts like a constant). For example:

```
def f(w1, w2):
    return 3 * w1 ** 2 + tf.stop_gradient(2 * w1 * w2)

with tf.GradientTape() as tape:
    z = f(w1, w2) # same result as without stop_gradient()

gradients = tape.gradient(z, [w1, w2]) # => returns [tensor 30., None]
```

Finally, you may occasionally run into some numerical issues when computing gradients. For example, if you compute the gradients of the `my_softplus()` function for large inputs, the result will be NaN:

```
>>> x = tf.Variable([100.])
>>> with tf.GradientTape() as tape:
...     z = my_softplus(x)
...
>>> tape.gradient(z, [x])
<tf.Tensor: [...] numpy=array([nan], dtype=float32)>
```

This is because computing the gradients of this function using autodiff leads to some numerical difficulties: due to floating point precision errors, autodiff ends up computing infinity divided by infinity (which returns NaN). Fortunately, we can analytically find that the derivative of the softplus function is just 1 / (1 + 1 / exp(x)), which is numerically stable. Next, we can tell TensorFlow to use this stable function when computing the gradients of the `my_softplus()` function, by decorating it with `@tf.custom_gradient`, and making it return both its normal output and the function that computes the derivatives (note that it will receive as input the gradients that were backpropagated so far, down to the softplus function, and according to the chain rule we should multiply them with this function's gradients):

```
@tf.custom_gradient
def my_better_softplus(z):
    exp = tf.exp(z)
    def my_softplus_gradients(grad):
        return grad / (1 + 1 / exp)
    return tf.math.log(exp + 1), my_softplus_gradients
```

Now when we compute the gradients of the `my_better_softplus()` function, we get the proper result, even for large input values (however, the main output still explodes because of the exponential: one workaround is to use `tf.where()` to just return the inputs when they are large).

Congratulations! You can now compute the gradients of any function (provided it is differentiable at the point where you compute it), you can even compute Hessians, block backpropagation when needed and even write your own gradient functions! This is probably more flexibility than you will ever need, even if you build your own custom training loops, as we will see now.

Custom Training Loops

In some rare cases, the `fit()` method may not be flexible enough for what you need to do. For example, the Wide and Deep paper we discussed in Chapter 10 actually uses two different optimizers: one for the wide path and the other for the deep path. Since the `fit()` method only uses one optimizer (the one that we specify when

compiling the model), implementing this paper requires writing your own custom loop.

You may also like to write your own custom training loops simply to feel more confident that it does precisely what you intent it to do (perhaps you are unsure about some details of the `fit()` method). It can sometimes feel safer to make everything explicit. However, remember that writing a custom training loop will make your code longer, more error prone and harder to maintain.

Unless you really need the extra flexibility, you should prefer using the `fit()` method rather than implementing your own training loop, especially if you work in a team.

First, let's build a simple model. No need to compile it, since we will handle the training loop manually:

```
l2_reg = keras.regularizers.l2(0.05)
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="elu", kernel_initializer="he_normal",
                       kernel_regularizer=l2_reg),
    keras.layers.Dense(1, kernel_regularizer=l2_reg)
])
```

Next, let's create a tiny function that will randomly sample a batch of instances from the training set (in Chapter 13 we will discuss the Data API, which offers a much better alternative):

```
def random_batch(X, y, batch_size=32):
    idx = np.random.randint(len(X), size=batch_size)
    return X[idx], y[idx]
```

Let's also define a function that will display the training status, including the number of steps, the total number of steps, the mean loss since the start of the epoch (i.e., we will use the `Mean` metric to compute it), and other metrics:

```
def print_status_bar(iteration, total, loss, metrics=None):
    metrics = " - ".join(["{}: {:.4f}".format(m.name, m.result())
                          for m in [loss] + (metrics or [])])
    end = "" if iteration < total else "\n"
    print("\r{}/{} - ".format(iteration, total) + metrics,
          end=end)
```

This code is self-explanatory, unless you are unfamiliar with Python string formatting: `{:.4f}` will format a float with 4 digits after the decimal point. Moreover, using `\r` (carriage return) along with `end=""` ensures that the status bar always gets printed on the same line. In the notebook, the `print_status_bar()` function also includes a progress bar, but you could use the handy tqdm library instead.

With that, let's get down to business! First, we need to define some hyperparameters, choose the optimizer, the loss function and the metrics (just the MAE in this example):

```
n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = keras.optimizers.Nadam(lr=0.01)
loss_fn = keras.losses.mean_squared_error
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.MeanAbsoluteError()]
```

And now we are ready to build the custom loop!

```
for epoch in range(1, n_epochs + 1):
    print("Epoch {}/{}".format(epoch, n_epochs))
    for step in range(1, n_steps + 1):
        X_batch, y_batch = random_batch(X_train_scaled, y_train)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch, training=True)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
        mean_loss(loss)
        for metric in metrics:
            metric(y_batch, y_pred)
        print_status_bar(step * batch_size, len(y_train), mean_loss, metrics)
    print_status_bar(len(y_train), len(y_train), mean_loss, metrics)
    for metric in [mean_loss] + metrics:
        metric.reset_states()
```

There's a lot going on in this code, so let's walk through it:

- We create two nested loops: one for the epochs, the other for the batches within an epoch.
- Then we sample a random batch from the training set.
- Inside the `tf.GradientTape()` block, we make a prediction for one batch (using the model as a function), and we compute the loss: it is equal to the main loss plus the other losses (in this model, there is one regularization loss per layer). Since the `mean_squared_error()` function returns one loss per instance, we compute the mean over the batch using `tf.reduce_mean()` (if you wanted to apply different weights to each instance, this is where you would do it). The regularization losses are already reduced to a single scalar each, so we just need to sum them (using `tf.add_n()`, which sums multiple tensors of the same shape and data type).

- Next, we ask the `tape` to compute the gradient of the loss with regards to each trainable variable (*not* all variables!), and we apply them to the optimizer to perform a Gradient Descent step.
- Next we update the mean loss and the metrics (over the current epoch), and we display the status bar.
- At the end of each epoch, we display the status bar again to make it look complete[11] and to print a line feed, and we reset the states of the mean loss and the metrics.

If you set the optimizer's `clipnorm` or `clipvalue` hyperparameters, it will take care of this for you. If you want to apply any other transformation to the gradients, simply do so before calling the `apply_gradients()` method.

If you add weight constraints to your model (e.g., by setting `kernel_constraint` or `bias_constraint` when creating a layer), you should update the training loop to apply these constraints just after `apply_gradients()`:

```
for variable in model.variables:
    if variable.constraint is not None:
        variable.assign(variable.constraint(variable))
```

Most importantly, this training loop does not handle layers that behave differently during training and testing (e.g., `BatchNormalization` or `Dropout`). To handle these, you need to call the model with `training=True` and make sure it propagates this to every layer that needs it.[12]

As you can see, there are quite a lot of things you need to get right, it is easy to make a mistake. But on the bright side, you get full control, so it's your call.

Now that you know how to customize any part of your models[13] and training algorithms, let's see how you can use TensorFlow's automatic graph generation feature: it can speed up your custom code considerably, and it will also make it portable to any platform supported by TensorFlow (see ???).

TensorFlow Functions and Graphs

In TensorFlow 1, graphs were unavoidable (as were the complexities that came with them): they were a central part of TensorFlow's API. In TensorFlow 2, they are still

11 The truth is we did not process every single instance in the training set because we sampled instances randomly, so some were processed more than once while others were not processed at all. In practice that's fine. Moreover, if the training set size is not a multiple of the batch size, we will miss a few instances.

12 Alternatively, check out `K.learning_phase()`, `K.set_learning_phase()` and `K.learning_phase_scope()`.

13 With the exception of optimizers, as very few people ever customize these: see the notebook for an example.

there, but not as central, and much (much!) simpler to use. To demonstrate this, let's start with a trivial function that just computes the cube of its input:

```
def cube(x):
    return x ** 3
```

We can obviously call this function with a Python value, such as an int or a float, or we can call it with a tensor:

```
>>> cube(2)
8
>>> cube(tf.constant(2.0))
<tf.Tensor: id=18634148, shape=(), dtype=float32, numpy=8.0>
```

Now, let's use `tf.function()` to convert this Python function to a *TensorFlow Function*:

```
>>> tf_cube = tf.function(cube)
>>> tf_cube
<tensorflow.python.eager.def_function.Function at 0x1546fc080>
```

This TF Function can then be used exactly like the original Python function, and it will return the same result (but as tensors):

```
>>> tf_cube(2)
<tf.Tensor: id=18634201, shape=(), dtype=int32, numpy=8>
>>> tf_cube(tf.constant(2.0))
<tf.Tensor: id=18634211, shape=(), dtype=float32, numpy=8.0>
```

Under the hood, `tf.function()` analyzed the computations performed by the `cube()` function and generated an equivalent computation graph! As you can see, it was rather painless (we will see how this works shortly). Alternatively, we could have used `tf.function` as a decorator; this is actually more common:

```
@tf.function
def tf_cube(x):
    return x ** 3
```

The original Python function is still available via the TF Function's `python_function` attribute, in case you ever need it:

```
>>> tf_cube.python_function(2)
8
```

TensorFlow optimizes the computation graph, pruning unused nodes, simplifying expressions (e.g., 1 + 2 would get replaced with 3), and more. Once the optimized graph is ready, the TF Function efficiently executes the operations in the graph, in the appropriate order (and in parallel when it can). As a result, a TF Function will usually run much faster than the original Python function, especially if it performs complex

computations.[14] Most of the time you will not really need to know more than that: when you want to boost a Python function, just transform it into a TF Function. That's all!

Moreover, when you write a custom loss function, a custom metric, a custom layer or any other custom function, and you use it in a Keras model (as we did throughout this chapter), Keras automatically converts your function into a TF Function, no need to use `tf.function()`. So most of the time, all this magic is 100% transparent.

You can tell Keras *not* to convert your Python functions to TF Functions by setting `dynamic=True` when creating a custom layer or a custom model. Alternatively, you can set `run_eagerly=True` when calling the model's `compile()` method.

TF Function generates a new graph for every unique set of input shapes and data types, and it caches it for subsequent calls. For example, if you call `tf_cube(tf.constant(10))`, a graph will be generated for int32 tensors of shape []. Then if you call `tf_cube(tf.constant(20))`, the same graph will be reused. But if you then call `tf_cube(tf.constant([10, 20]))`, a new graph will be generated for int32 tensors of shape [2]. This is how TF Functions handle polymorphism (i.e., varying argument types and shapes). However, this is only true for tensor arguments: if you pass numerical Python values to a TF Function, a new graph will be generated for every distinct value: for example, calling `tf_cube(10)` and `tf_cube(20)` will generate two graphs.

If you call a TF Function many times with different numerical Python values, then many graphs will be generated, slowing down your program and using up a lot of RAM. Python values should be reserved for arguments that will have few unique values, such as hyperparameters like the number of neurons per layer. This allows TensorFlow to better optimize each variant of your model.

Autograph and Tracing

So how does TensorFlow generate graphs? Well, first it starts by analyzing the Python function's source code to capture all the control flow statements, such as `for` loops and `while` loops, `if` statements, as well as `break`, `continue` and `return` statements. This first step is called *autograph*. The reason TensorFlow has to analyze the source code is that Python does not provide any other way to capture control flow statements: it offers magic methods like `__add__()` or `__mul__()` to capture operators like

14 However, in this trivial example, the computation graph is so small that there is nothing at all to optimize, so `tf_cube()` actually runs much slower than `cube()`.

+ and *, but there are no `__while__()` or `__if__()` magic methods. After analyzing the function's code, autograph outputs an upgraded version of that function in which all the control flow statements are replaced by the appropriate TensorFlow operations, such as `tf.while_loop()` for loops and `tf.cond()` for `if` statements. For example, in Figure 12-4, autograph analyzes the source code of the `sum_squares()` Python function, and it generates the `tf__sum_squares()` function. In this function, the `for` loop is replaced by the definition of the `loop_body()` function (containing the body of the original `for` loop), followed by a call to the `for_stmt()` function. This call will build the appropriate `tf.while_loop()` operation in the computation graph.

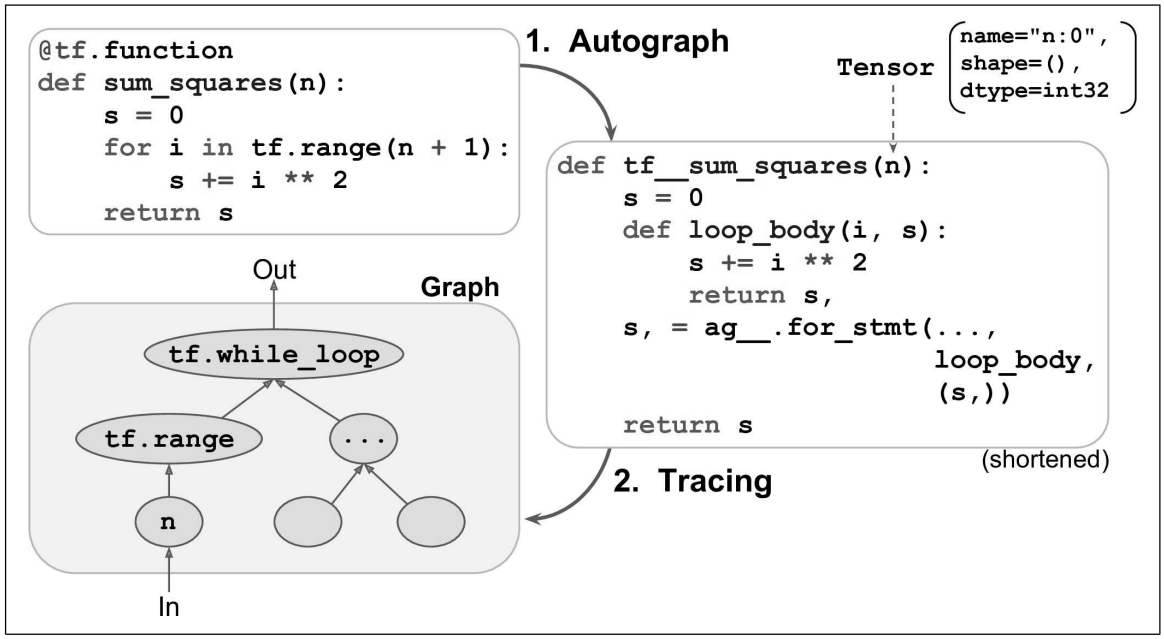

Figure 12-4. How TensorFlow generates graphs using autograph and tracing

Next, TensorFlow calls this "upgraded" function, but instead of passing the actual argument, it passes a *symbolic tensor*, meaning a tensor without any actual value, only a name, a data type, and a shape. For example, if you call `sum_squares(tf.constant(10))`, then the `tf__sum_squares()` function will actually be called with a symbolic tensor of type int32 and shape []. The function will run in *graph mode*, meaning that each TensorFlow operation will just add a node in the graph to represent itself and its output tensor(s) (as opposed to the regular mode, called *eager execution*, or *eager mode*). In graph mode, TF operations do not perform any actual computations. This should feel familiar if you know TensorFlow 1, as graph mode was the default mode. In Figure 12-4, you can see the `tf__sum_squares()` function being called with a symbolic tensor as argument (in this case, an int32 tensor of shape []), and the final graph generated during tracing. The ellipses represent operations, and the arrows represent tensors (both the generated function and the graph are simplified).

To view the generated function's source code, you can call `tf.autograph.to_code(sum_squares.python_function)`. The code is not meant to be pretty, but it can sometimes help for debugging.

TF Function Rules

Most of the time, converting a Python function that performs TensorFlow operations into a TF Function is trivial: just decorate it with `@tf.function` or let Keras take care of it for you. However, there are a few rules to respect:

- If you call any external library, including NumPy or even the standard library, this call will run only during tracing, it will not be part of the graph. Indeed, a TensorFlow graph can only include TensorFlow constructs (tensors, operations, variables, datasets, and so on). So make sure you use `tf.reduce_sum()` instead of `np.sum()`, and `tf.sort()` instead of the built-in `sorted()` function, and so on (unless you really want the code to run only during tracing).
  - For example, if you define a TF function `f(x)` that just returns `np.random.rand()`, a random number will only be generated when the function is traced, so `f(tf.constant(2.))` and `f(tf.constant(3.))` will return the same random number, but `f(tf.constant([2., 3.]))` will return a different one. If you replace `np.random.rand()` with `tf.random.uniform([])`, then a new random number will be generated upon every call, since the operation will be part of the graph.
  - If your non-TensorFlow code has side-effects (such as logging something or updating a Python counter), then you should not expect that side-effect to occur every time you call the TF Function, as it will only occur when the function is traced.
  - You can wrap arbitrary Python code in a `tf.py_function()` operation, but this will hinder performance, as TensorFlow will not be able to do any graph optimization on this code, and it will also reduce portability, as the graph will only run on platforms where Python is available (and the right libraries installed).
- You can call other Python functions or TF Functions, but they should follow the same rules, as TensorFlow will also capture their operations in the computation graph. Note that these other functions do not need to be decorated with `@tf.function`.
- If the function creates a TensorFlow variable (or any other stateful TensorFlow object, such as a dataset or a queue), it must do so upon the very first call, and only then, or else you will get an exception. It is usually preferable to create variables outside of the TF Function (e.g., in the `build()` method of a custom layer).

- The source code of your Python function should be available to TensorFlow. If the source code is unavailable (for example, if you define your function in the Python shell, which does not give access to the source code, or if you deploy only the compiled Python files `*.pyc` to production), then the graph generation process will fail or have limited functionality.
- TensorFlow will only capture `for` loops that iterate over a tensor or a `Dataset`. So make sure you use `for i in tf.range(10)` rather than `for i in range(10)`, or else the loop will not be captured in the graph. Instead, it will run during tracing. This may be what you want, if the `for` loop is meant to build the graph, for example to create each layer in a neural network.
- And as always, for performance reasons, you should prefer a vectorized implementation whenever you can, rather than using loops.

It's time to sum up! In this chapter we started with a brief overview of TensorFlow, then we looked at TensorFlow's low-level API, including tensors, operations, variables and special data structures. We then used these tools to customize almost every component in tf.keras. Finally, we looked at how TF Functions can boost performance, how graphs are generated using autograph and tracing, and what rules to follow when you write TF Functions (if you would like to open the black box a bit further, for example to explore the generated graphs, you will find further technical details in ???).

In the next chapter, we will look at how to efficiently load and preprocess data with TensorFlow.

CHAPTER 13

Loading and Preprocessing Data with TensorFlow

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 13 in the final release of the book.

So far we have used only datasets that fit in memory, but Deep Learning systems are often trained on very large datasets that will not fit in RAM. Ingesting a large dataset and preprocessing it efficiently can be tricky to implement with other Deep Learning libraries, but TensorFlow makes it easy thanks to the *Data API*: you just create a dataset object, tell it where to get the data, then transform it in any way you want, and TensorFlow takes care of all the implementation details, such as multithreading, queuing, batching, prefetching, and so on.

Off the shelf, the Data API can read from text files (such as CSV files), binary files with fixed-size records, and binary files that use TensorFlow's TFRecord format, which supports records of varying sizes. TFRecord is a flexible and efficient binary format based on Protocol Buffers (an open source binary format). The Data API also has support for reading from SQL databases. Moreover, many Open Source extensions are available to read from all sorts of data sources, such as Google's BigQuery service.

However, reading huge datasets efficiently is not the only difficulty: the data also needs to be preprocessed. Indeed, it is not always composed strictly of convenient numerical fields: sometimes there will be text features, categorical features, and so on. To handle this, TensorFlow provides the *Features API*: it lets you easily convert these features to numerical features that can be consumed by your neural network. For

example, categorical features with a large number of categories (such as cities, or words) can be encoded using *embeddings* (as we will see, an embedding is a trainable dense vector that represents a category).

Both the Data API and the Features API work seamlessly with tf.keras.

In this chapter, we will cover the Data API, the TFRecord format and the Features API in detail. We will also take a quick look at a few related projects from TensorFlow's ecosystem:

- TF Transform (*tf.Transform*) makes it possible to write a single preprocessing function that can be run both in batch mode on your full training set, before training (to speed it up), and then exported to a TF Function and incorporated into your trained model, so that once it is deployed in production, it can take care of preprocessing new instances on the fly.
- TF Datasets (TFDS) provides a convenient function to download many common datasets of all kinds, including large ones like ImageNet, and it provides convenient dataset objects to manipulate them using the Data API.

So let's get started!

The Data API

The whole Data API revolves around the concept of a *dataset*: as you might suspect, this represents a sequence of data items. Usually you will use datasets that gradually read data from disk, but for simplicity let's just create a dataset entirely in RAM using `tf.data.Dataset.from_tensor_slices()`:

```
>>> X = tf.range(10)  # any data tensor
>>> dataset = tf.data.Dataset.from_tensor_slices(X)
>>> dataset
<TensorSliceDataset shapes: (), types: tf.int32>
```

The `from_tensor_slices()` function takes a tensor and creates a `tf.data.Dataset` whose elements are all the slices of `X` (along the first dimension), so this dataset contains 10 items: tensors 0, 1, 2, …, 9. In this case we would have obtained the same dataset if we had used `tf.data.Dataset.range(10)`.

You can simply iterate over a dataset's items like this:

```
>>> for item in dataset:
...     print(item)
```

```
...
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
[...]
tf.Tensor(9, shape=(), dtype=int32)
```

Chaining Transformations

Once you have a dataset, you can apply all sorts of transformations to it by calling its transformation methods. Each method returns a new dataset, so you can chain transformations like this (this chain is illustrated in Figure 13-1):

```
>>> dataset = dataset.repeat(3).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)
tf.Tensor([8 9], shape=(2,), dtype=int32)
```

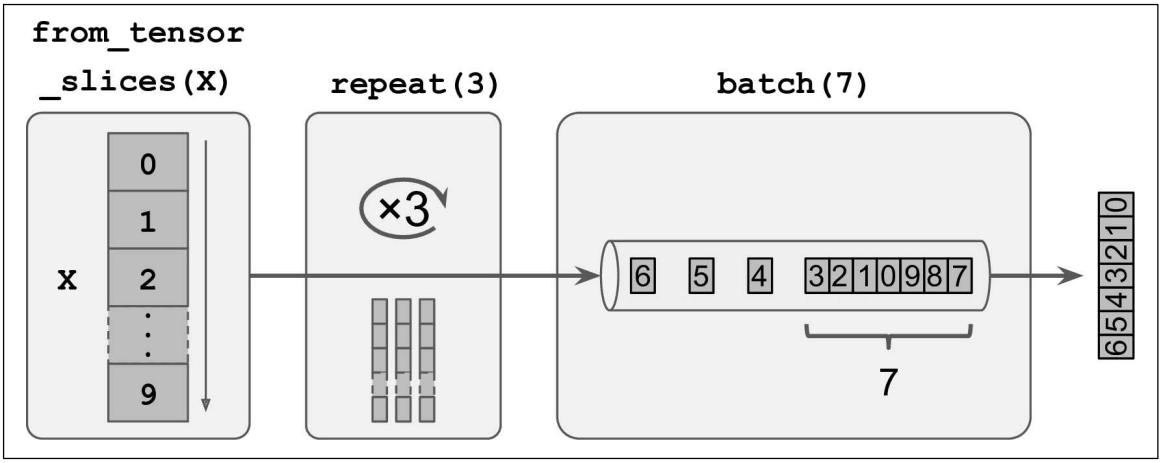

Figure 13-1. Chaining Dataset Transformations

In this example, we first call the `repeat()` method on the original dataset, and it returns a new dataset that will repeat the items of the original dataset 3 times. Of course, this will not copy the whole data in memory 3 times! In fact, if you call this method with no arguments, the new dataset will repeat the source dataset forever. Then we call the `batch()` method on this new dataset, and again this creates a new dataset. This one will group the items of the previous dataset in batches of 7 items. Finally, we iterate over the items of this final dataset. As you can see, the `batch()` method had to output a final batch of size 2 instead of 7, but you can call it with `drop_remainder=True` if you want it to drop this final batch so that all batches have the exact same size.

The dataset methods do *not* modify datasets, they create new ones, so make sure to keep a reference to these new datasets (e.g., `dataset = ...`), or else nothing will happen.

You can also apply any transformation you want to the items by calling the `map()` method. For example, this creates a new dataset with all items doubled:

```
>>> dataset = dataset.map(lambda x: x * 2) # Items: [0,2,4,6,8,10,12]
```

This function is the one you will call to apply any preprocessing you want to your data. Sometimes, this will include computations that can be quite intensive, such as reshaping or rotating an image, so you will usually want to spawn multiple threads to speed things up: it's as simple as setting the `num_parallel_calls` argument.

While the `map()` applies a transformation to each item, the `apply()` method applies a transformation to the dataset as a whole. For example, the following code "unbatches" the dataset, by applying the `unbatch()` function to the dataset (this function is currently experimental, but it will most likely move to the core API in a future release). Each item in the new dataset will be a single integer tensor instead of a batch of 7 integers:

```
>>> dataset = dataset.apply(tf.data.experimental.unbatch()) # Items: 0,2,4,...
```

It is also possible to simply filter the dataset using the `filter()` method:

```
>>> dataset = dataset.filter(lambda x: x < 10) # Items: 0 2 4 6 8 0 2 4 6...
```

You will often want to look at just a few items from a dataset. You can use the `take()` method for that:

```
>>> for item in dataset.take(3):
...     print(item)
...
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
```

Shuffling the Data

As you know, Gradient Descent works best when the instances in the training set are independent and identically distributed (see Chapter 4). A simple way to ensure this is to shuffle the instances. For this, you can just use the `shuffle()` method. It will create a new dataset that will start by filling up a buffer with the first items of the source dataset, then whenever it is asked for an item, it will pull one out randomly from the buffer, and replace it with a fresh one from the source dataset, until it has iterated entirely through the source dataset. At this point it continues to pull out items randomly from the buffer until it is empty. You must specify the buffer size, and

it is important to make it large enough or else shuffling will not be very efficient.[1] However, obviously do not exceed the amount of RAM you have, and even if you have plenty of it, there's no need to go well beyond the dataset's size. You can provide a random seed if you want the same random order every time you run your program.

```
>>> dataset = tf.data.Dataset.range(10).repeat(3) # 0 to 9, three times
>>> dataset = dataset.shuffle(buffer_size=5, seed=42).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 2 3 6 7 9 4], shape=(7,), dtype=int64)
tf.Tensor([5 0 1 1 8 6 5], shape=(7,), dtype=int64)
tf.Tensor([4 8 7 1 2 3 0], shape=(7,), dtype=int64)
tf.Tensor([5 4 2 7 8 9 9], shape=(7,), dtype=int64)
tf.Tensor([3 6], shape=(2,), dtype=int64)
```

If you call `repeat()` on a shuffled dataset, by default it will generate a new order at every iteration. This is generally a good idea, but if you prefer to reuse the same order at each iteration (e.g., for tests or debugging), you can set `reshuffle_each_iteration=False`.

For a large dataset that does not fit in memory, this simple shuffling-buffer approach may not be sufficient, since the buffer will be small compared to the dataset. One solution is to shuffle the source data itself (for example, on Linux you can shuffle text files using the `shuf` command). This will definitely improve shuffling a lot! However, even if the source data is shuffled, you will usually want to shuffle it some more, or else the same order will be repeated at each epoch, and the model may end up being biased (e.g., due to some spurious patterns present by chance in the source data's order). To shuffle the instances some more, a common approach is to split the source data into multiple files, then read them in a random order during training. However, instances located in the same file will still end up close to each other. To avoid this you can pick multiple files randomly, and read them simultaneously, interleaving their lines. Then on top of that you can add a shuffling buffer using the `shuffle()` method. If all this sounds like a lot of work, don't worry: the Data API actually makes all this possible in just a few lines of code. Let's see how to do this.

1 Imagine a sorted deck of cards on your left: suppose you just take the top 3 cards and shuffle them, then pick one randomly and put it to your right, keeping the other 2 in your hands. Take another card on your left, shuffle the 3 cards in your hands and pick one of them randomly, and put it on your right. When you are done going through all the cards like this, you will have a deck of cards on your right: do you think it will be perfectly shuffled?

Interleaving Lines From Multiple Files

First, let's suppose that you loaded the California housing dataset, you shuffled it (unless it was already shuffled), you split it into a training set, a validation set and a test set, then you split each set into many CSV files that each look like this (each row contains 8 input features plus the target median house value):

```
MedInc,HouseAge,AveRooms,AveBedrms,Popul,AveOccup,Lat,Long,MedianHouseValue
3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442
5.3275,5.0,6.4900,0.9910,3464.0,3.4433,33.69,-117.39,1.687
3.1,29.0,7.5423,1.5915,1328.0,2.2508,38.44,-122.98,1.621
[...]
```

Let's also suppose `train_filepaths` contains the list of file paths (and you also have `valid_filepaths` and `test_filepaths`):

```
>>> train_filepaths
['datasets/housing/my_train_00.csv', 'datasets/housing/my_train_01.csv',...]
```

Now let's create a dataset containing only these file paths:

```
filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

By default, the `list_files()` function returns a dataset that shuffles the file paths. In general this is a good thing, but you can set `shuffle=False` if you do not want that, for some reason.

Next, we can call the `interleave()` method to read from 5 files at a time and interleave their lines (skipping the first line of each file, which is the header row, using the `skip()` method):

```
n_readers = 5
dataset = filepath_dataset.interleave(
    lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
    cycle_length=n_readers)
```

The `interleave()` method will create a dataset that will pull 5 file paths from the `filepath_dataset`, and for each one it will call the function we gave it (a lambda in this example) to create a new dataset, in this case a `TextLineDataset`. It will then cycle through these 5 datasets, reading one line at a time from each until all datasets are out of items. Then it will get the next 5 file paths from the `filepath_dataset`, and interleave them the same way, and so on until it runs out of file paths.

For interleaving to work best, it is preferable to have files of identical length, or else the end of the longest files will not be interleaved.

By default, `interleave()` does not use parallelism, it just reads one line at a time from each file, sequentially. However, if you want it to actually read files in parallel, you can set the `num_parallel_calls` argument to the number of threads you want. You can even set it to `tf.data.experimental.AUTOTUNE` to make TensorFlow choose the right number of threads dynamically based on the available CPU (however, this is an experimental feature for now). Let's look at what the dataset contains now:

```
>>> for line in dataset.take(5):
...     print(line.numpy())
...
b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782'
b'4.1812,52.0,5.7013,0.9965,692.0,2.4027,33.73,-118.31,3.215'
b'3.6875,44.0,4.5244,0.9930,457.0,3.1958,34.04,-118.15,1.625'
b'3.3456,37.0,4.5140,0.9084,458.0,3.2253,36.67,-121.7,2.526'
b'3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442'
```

These are the first rows (ignoring the header row) of 5 CSV files, chosen randomly. Looks good! But as you can see, these are just byte strings, we need to parse them, and also scale the data.

Preprocessing the Data

Let's implement a small function that will perform this preprocessing:

```
X_mean, X_std = [...] # mean and scale of each feature in the training set
n_inputs = 8

def preprocess(line):
  defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
  fields = tf.io.decode_csv(line, record_defaults=defs)
  x = tf.stack(fields[:-1])
  y = tf.stack(fields[-1:])
  return (x - X_mean) / X_std, y
```

Let's walk through this code:

- First, we assume that you have precomputed the mean and standard deviation of each feature in the training set. `X_mean` and `X_std` are just 1D tensors (or NumPy arrays) containing 8 floats, one per input feature.
- The `preprocess()` function takes one CSV line, and starts by parsing it. For this, it uses the `tf.io.decode_csv()` function, which takes two arguments: the first is the line to parse, and the second is an array containing the default value for each column in the CSV file. This tells TensorFlow not only the default value for each column, but also the number of columns and the type of each column. In this example, we tell it that all feature columns are floats and missing values should default to 0, but we provide an empty array of type `tf.float32` as the default value for the last column (the target): this tells TensorFlow that this column con-

tains floats, but that there is no default value, so it will raise an exception if it encounters a missing value.

- The `decode_csv()` function returns a list of scalar tensors (one per column) but we need to return 1D tensor arrays. So we call `tf.stack()` on all tensors except for the last one (the target): this will stack these tensors into a 1D array. We then do the same for the target value (this makes it a 1D tensor array with a single value, rather than a scalar tensor).
- Finally, we scale the input features by subtracting the feature means and then dividing by the feature standard deviations, and we return a tuple containing the scaled features and the target.

Let's test this preprocessing function:

```
>>> preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782')
(<tf.Tensor: id=6227, shape=(8,), dtype=float32, numpy=
 array([ 0.16579159,  1.216324  , -0.05204564, -0.39215982, -0.5277444 ,
        -0.2633488 ,  0.8543046 , -1.3072058 ], dtype=float32)>,
 <tf.Tensor: [...], numpy=array([2.782], dtype=float32)>)
```

We can now apply this preprocessing function to the dataset.

Putting Everything Together

To make the code reusable, let's put together everything we have discussed so far into a small helper function: it will create and return a dataset that will efficiently load California housing data from multiple CSV files, then shuffle it, preprocess it and batch it (see Figure 13-2):

```
def csv_reader_dataset(filepaths, repeat=None, n_readers=5,
                       n_read_threads=None, shuffle_buffer_size=10000,
                       n_parse_threads=5, batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths).repeat(repeat)
    dataset = dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
        cycle_length=n_readers, num_parallel_calls=n_read_threads)
    dataset = dataset.shuffle(shuffle_buffer_size)
    dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
    dataset = dataset.batch(batch_size)
    return dataset.prefetch(1)
```

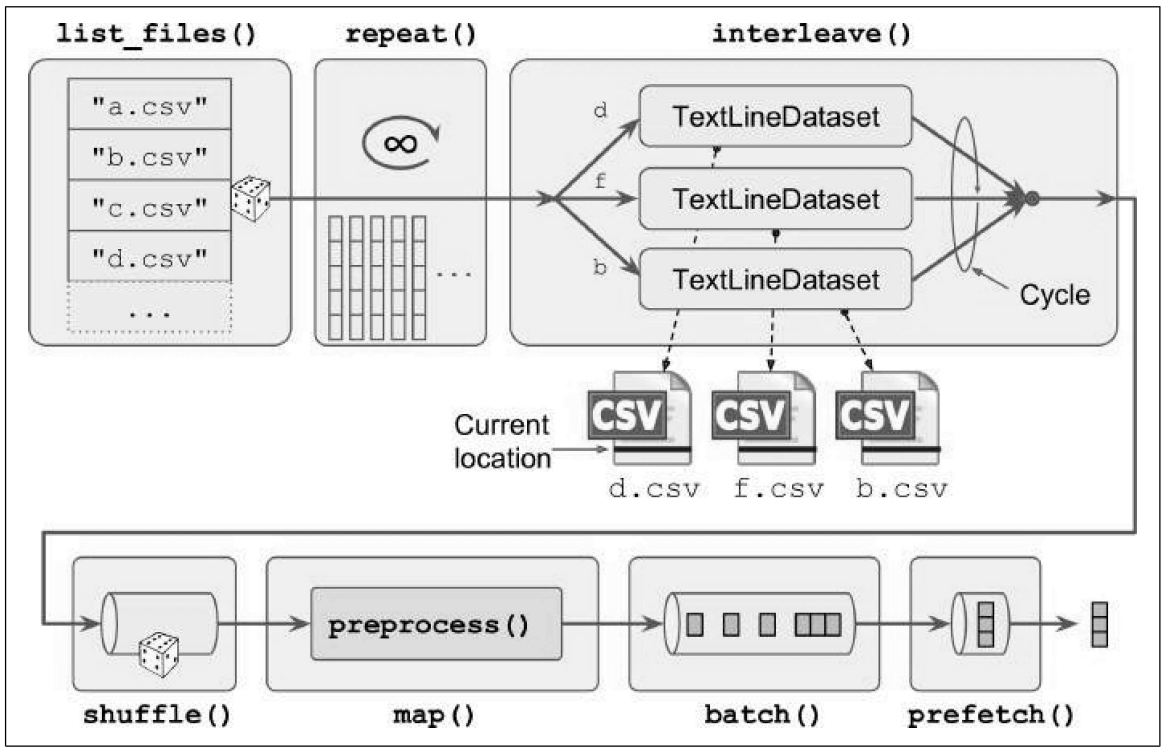

Figure 13-2. Loading and Preprocessing Data From Multiple CSV Files

Everything should make sense in this code, except the very last line (`prefetch(1)`), which is actually quite important for performance.

Prefetching

By calling `prefetch(1)` at the end, we are creating a dataset that will do its best to always be one batch ahead[2]. In other words, while our training algorithm is working on one batch, the dataset will already be working in parallel on getting the next batch ready. This can improve performance dramatically, as is illustrated on Figure 13-3. If we also ensure that loading and preprocessing are multithreaded (by setting `num_parallel_calls` when calling `interleave()` and `map()`), we can exploit multiple cores on the CPU and hopefully make preparing one batch of data shorter than running a training step on the GPU: this way the GPU will be almost 100% utilized (except for the data transfer time from the CPU to the GPU), and training will run much faster.

2 In general, just prefetching one batch is fine, but in some cases you may need to prefetch a few more. Alternatively, you can let TensorFlow decide automatically by passing `tf.data.experimental.AUTOTUNE` (this is an experimental feature for now).

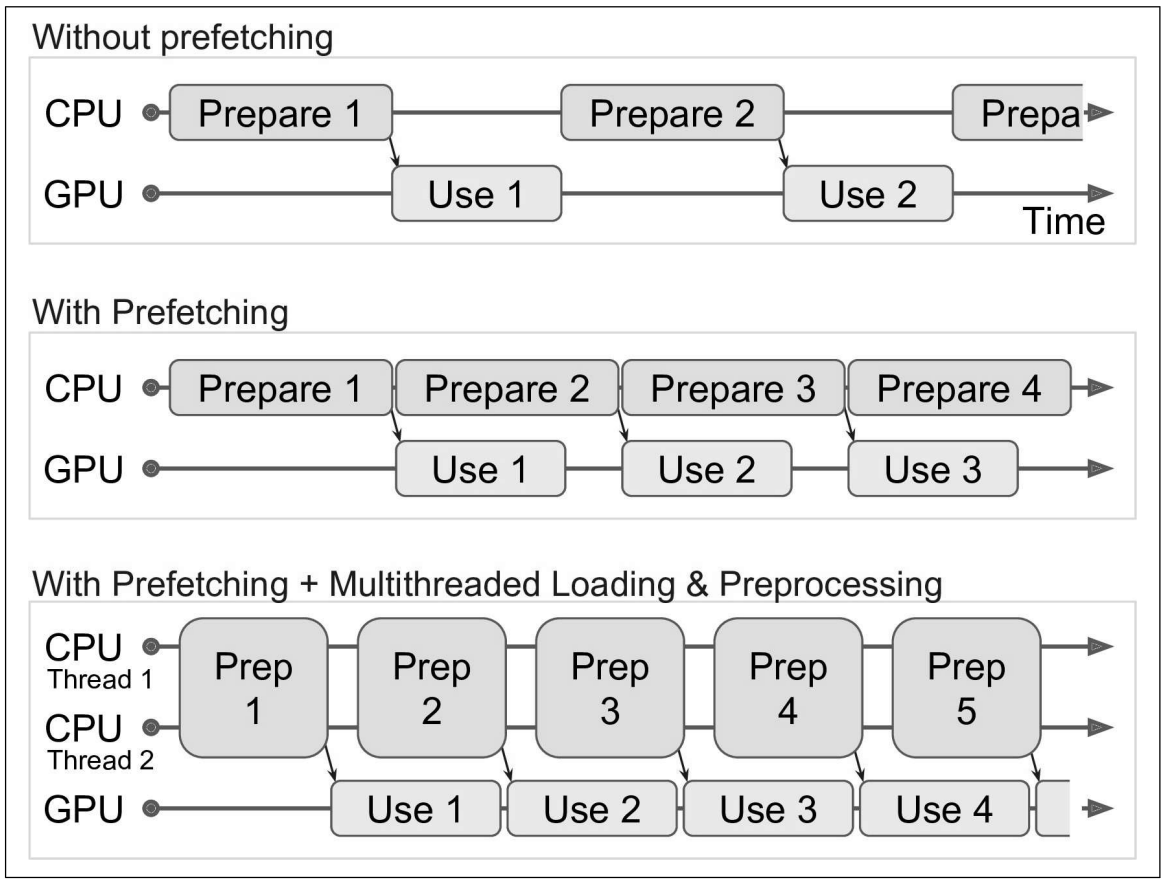

Figure 13-3. Speedup Training Thanks to Prefetching and Multithreading

If you plan to purchase a GPU card, its processing power and its memory size are of course very important (in particular, a large RAM is crucial for computer vision), but its *memory bandwidth* is just as important as the processing power to get good performance: this is the number of gigabytes of data it can get in or out of its RAM per second.

With that, you can now build efficient input pipelines to load and preprocess data from multiple text files. We have discussed the most common dataset methods, but there are a few more you may want to look at: `concatenate()`, `zip()`, `window()`, `reduce()`, `cache()`, `shard()`, `flat_map()` and `padded_batch()`. There are also a couple more class methods: `from_generator()` and `from_tensors()`, which create a new dataset from a Python generator or a list of tensors respectively. Please check the API documentation for more details. Also note that there are experimental features available in `tf.data.experimental`, many of which will most likely make it to the core API in future releases (e.g., check out the `CsvDataset` class and the `SqlDataset` classes).

Using the Dataset With tf.keras

Now we can use the `csv_reader_dataset()` function to create a dataset for the training set (ensuring it repeats the data forever), the validation set and the test set:

```
train_set = csv_reader_dataset(train_filepaths, repeat=None)
valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)
```

And now we can simply build and train a Keras model using these datasets.[3] All we need to do is to call the `fit()` method with the datasets instead of `X_train` and `y_train`, and specify the number of steps per epoch for each set:[4]

```
model = keras.models.Sequential([...])
model.compile([...])
model.fit(train_set, steps_per_epoch=len(X_train) // batch_size, epochs=10,
          validation_data=valid_set,
          validation_steps=len(X_valid) // batch_size)
```

Similarly, we can pass a dataset to the `evaluate()` and `predict()` methods (and again specify the number of steps per epoch):

```
model.evaluate(test_set, steps=len(X_test) // batch_size)
model.predict(new_set, steps=len(X_new) // batch_size)
```

Unlike the other sets, the `new_set` will usually not contain labels (if it does, Keras will just ignore them). Note that in all these cases, you can still use NumPy arrays instead of datasets if you want (but of course they need to have been loaded and preprocessed first).

If you want to build your own custom training loop (as in Chapter 12), you can just iterate over the training set, very naturally:

```
for X_batch, y_batch in train_set:
    [...] # perform one gradient descent step
```

In fact, it is even possible to create a tf.function (see Chapter 12) that performs the whole training loop![5]

```
@tf.function
def train(model, optimizer, loss_fn, n_epochs, [...]):
    train_set = csv_reader_dataset(train_filepaths, repeat=n_epochs, [...])
    for X_batch, y_batch in train_set:
        with tf.GradientTape() as tape:
```

3 Support for datasets is specific to tf.keras, it will not work on other implementations of the Keras API.

4 The number of steps per epoch is optional if the dataset just goes through the data once, but if you do not specify it, the progress bar will not be displayed during the first epoch.

5 Note that for now the dataset must be created within the TF Function. This may be fixed by the time you read these lines (see TensorFlow issue #25414).

```
            y_pred = model(X_batch)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Congratulations, you now know how to build powerful input pipelines using the Data API! However, so far we have used CSV files, which are common, simple and convenient, but they are not really efficient, and they do not support large or complex data structures very well, such as images or audio. So let's use TFRecords instead.

If you are happy with CSV files (or whatever other format you are using), you do not *have* to use TFRecords. As the saying goes, if it ain't broke, don't fix it! TFRecords are useful when the bottleneck during training is loading and parsing the data.

The TFRecord Format

The TFRecord format is TensorFlow's preferred format for storing large amounts of data and reading it efficiently. It is a very simple binary format that just contains a sequence of binary records of varying sizes (each record just has a length, a CRC checksum to check that the length was not corrupted, then the actual data, and finally a CRC checksum for the data). You can easily create a TFRecord file using the `tf.io.TFRecordWriter` class:

```
with tf.io.TFRecordWriter("my_data.tfrecord") as f:
    f.write(b"This is the first record")
    f.write(b"And this is the second record")
```

And you can then use a `tf.data.TFRecordDataset` to read one or more TFRecord files:

```
filepaths = ["my_data.tfrecord"]
dataset = tf.data.TFRecordDataset(filepaths)
for item in dataset:
    print(item)
```

This will output:

```
tf.Tensor(b'This is the first record', shape=(), dtype=string)
tf.Tensor(b'And this is the second record', shape=(), dtype=string)
```

By default, a `TFRecordDataset` will read files one by one, but you can make it read multiple files in parallel and interleave their records by setting `num_parallel_reads`. Alternatively, you could obtain the same result by using `list_files()` and `interleave()` as we did earlier to read multiple CSV files.

Compressed TFRecord Files

It can sometimes be useful to compress your TFRecord files, especially if they need to be loaded via a network connection. You can create a compressed TFRecord file by setting the `options` argument:

```
options = tf.io.TFRecordOptions(compression_type="GZIP")
with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:
  [...]
```

When reading a compressed TFRecord file, you need to specify the compression type:

```
dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"],
                                  compression_type="GZIP")
```

A Brief Introduction to Protocol Buffers

Even though each record can use any binary format you want, TFRecord files usually contain serialized Protocol Buffers (also called *protobufs*). This is a portable, extensible and efficient binary format developed at Google back in 2001 and Open Sourced in 2008, and they are now widely used, in particular in gRPC, Google's remote procedure call system. Protocol Buffers are defined using a simple language that looks like this:

```
syntax = "proto3";
message Person {
  string name = 1;
  int32 id = 2;
  repeated string email = 3;
}
```

This definition says we are using the protobuf format version 3, and it specifies that each `Person` object[6] may (optionally) have a `name` of type `string`, an `id` of type `int32`, and zero or more `email` fields, each of type `string`. The numbers 1, 2 and 3 are the field identifiers: they will be used in each record's binary representation. Once you have a definition in a `.proto` file, you can compile it. This requires `protoc`, the protobuf compiler, to generate access classes in Python (or some other language). Note that the protobuf definitions we will use have already been compiled for you, and their Python classes are part of TensorFlow, so you will not need to use `protoc`. All you need to know is how to use protobuf access classes in Python. To illustrate the basics, let's look at a simple example that uses the access classes generated for the `Person` protobuf (the code is explained in the comments):

```
>>> from person_pb2 import Person  # import the generated access class
>>> person = Person(name="Al", id=123, email=["a@b.com"])  # create a Person
>>> print(person)  # display the Person
```

6 Since protobuf objects are meant to be serialized and transmitted, they are called *messages*.

```
name: "Al"
id: 123
email: "a@b.com"
>>> person.name  # read a field
"Al"
>>> person.name = "Alice"  # modify a field
>>> person.email[0]  # repeated fields can be accessed like arrays
"a@b.com"
>>> person.email.append("c@d.com")  # add an email address
>>> s = person.SerializeToString()  # serialize the object to a byte string
>>> s
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person()  # create a new Person
>>> person2.ParseFromString(s)  # parse the byte string (27 bytes long)
27
>>> person == person2  # now they are equal
True
```

In short, we import the `Person` class generated by `protoc`, we create an instance and we play with it, visualizing it, reading and writing some fields, then we serialize it using the `SerializeToString()` method. This is the binary data that is ready to be saved or transmitted over the network. When reading or receiving this binary data, we can parse it using the `ParseFromString()` method, and we get a copy of the object that was serialized.[7]

We could save the serialized `Person` object to a TFRecord file, then we could load and parse it: everything would work fine. However, `SerializeToString()` and `ParseFromString()` are not TensorFlow operations (and neither are the other operations in this code), so they cannot be included in a TensorFlow Function (except by wrapping them in a `tf.py_function()` operation, which would make the code slower and less portable, as we saw in Chapter 12). Fortunately, TensorFlow does include special protobuf definitions for which it provides parsing operations.

TensorFlow Protobufs

The main protobuf typically used in a TFRecord file is the `Example` protobuf, which represents one instance in a dataset. It contains a list of named features, where each feature can either be a list of byte strings, a list of floats or a list of integers. Here is the protobuf definition:

```
syntax = "proto3";
message BytesList { repeated bytes value = 1; }
message FloatList { repeated float value = 1 [packed = true]; }
message Int64List { repeated int64 value = 1 [packed = true]; }
```

7 This chapter contains the bare minimum you need to know about protobufs to use TFRecords. To learn more about protobufs, please visit *https://homl.info/protobuf*.

```
message Feature {
    oneof kind {
        BytesList bytes_list = 1;
        FloatList float_list = 2;
        Int64List int64_list = 3;
    }
};
message Features { map<string, Feature> feature = 1; };
message Example { Features features = 1; };
```

The definitions of `BytesList`, `FloatList` and `Int64List` are straightforward enough (`[packed = true]` is used for repeated numerical fields, for a more efficient encoding). A `Feature` either contains a `BytesList`, a `FloatList` or an `Int64List`. A `Features` (with an s) contains a dictionary that maps a feature name to the corresponding feature value. And finally, an `Example` just contains a `Features` object.[8] Here is how you could create a `tf.train.Example` representing the same person as earlier, and write it to TFRecord file:

```
from tensorflow.train import BytesList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example

person_example = Example(
    features=Features(
        feature={
            "name": Feature(bytes_list=BytesList(value=[b"Alice"])),
            "id": Feature(int64_list=Int64List(value=[123])),
            "emails": Feature(bytes_list=BytesList(value=[b"a@b.com",
                                                          b"c@d.com"]))
        }))
```

The code is a bit verbose and repetitive, but it's rather straightforward (and you could easily wrap it inside a small helper function). Now that we have an `Example` protobuf, we can serialize it by calling its `SerializeToString()` method, then write the resulting data to a TFRecord file:

```
with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    f.write(person_example.SerializeToString())
```

Normally you would write much more than just one example! Typically, you would create a conversion script that reads from your current format (say, CSV files), creates an `Example` protobuf for each instance, serializes them and saves them to several TFRecord files, ideally shuffling them in the process. This requires a bit of work, so once again make sure it is really necessary (perhaps your pipeline works fine with CSV files).

8 Why was `Example` even defined since it contains no more than a `Features` object? Well, TensorFlow may one day decide to add more fields to it. As long as the new `Example` definition still contains the `features` field, with the same id, it will be backward compatible. This extensibility is one of the great features of protobufs.

Now that we have a nice TFRecord file containing a serialized `Example`, let's try to load it.

Loading and Parsing Examples

To load the serialized `Example` protobufs, we will use a `tf.data.TFRecordDataset` once again, and we will parse each `Example` using `tf.io.parse_single_example()`. This is a TensorFlow operation so it can be included in a TF Function. It requires at least two arguments: a string scalar tensor containing the serialized data, and a description of each feature. The description is a dictionary that maps each feature name to either a `tf.io.FixedLenFeature` descriptor indicating the feature's shape, type and default value, or a `tf.io.VarLenFeature` descriptor indicating only the type (if the length may vary, such as for the `"emails"` feature). For example:

```
feature_description = {
    "name": tf.io.FixedLenFeature([], tf.string, default_value=""),
    "id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
    "emails": tf.io.VarLenFeature(tf.string),
}

for serialized_example in tf.data.TFRecordDataset(["my_contacts.tfrecord"]):
    parsed_example = tf.io.parse_single_example(serialized_example,
                                                feature_description)
```

The fixed length features are parsed as regular tensors, but the variable length features are parsed as sparse tensors. You can convert a sparse tensor to a dense tensor using `tf.sparse.to_dense()`, but in this case it is simpler to just access its values:

```
>>> tf.sparse.to_dense(parsed_example["emails"], default_value=b"")
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
>>> parsed_example["emails"].values
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
```

A `BytesList` can contain any binary data you want, including any serialized object. For example, you can use `tf.io.encode_jpeg()` to encode an image using the JPEG format, and put this binary data in a `BytesList`. Later, when your code reads the TFRecord, it will start by parsing the `Example`, then you will need to call `tf.io.decode_jpeg()` to parse the data and get the original image (or you can use `tf.io.decode_image()`, which can decode any BMP, GIF, JPEG or PNG image). You can also store any tensor you want in a `BytesList` by serializing the tensor using `tf.io.serialize_tensor()`, then putting the resulting byte string in a `BytesList` feature. Later, when you parse the TFRecord, you can parse this data using `tf.io.parse_tensor()`.

Instead of parsing examples one by one using `tf.io.parse_single_example()`, you may want to parse them batch by batch using `tf.io.parse_example()`:

```
dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).batch(10)
for serialized_examples in dataset:
    parsed_examples = tf.io.parse_example(serialized_examples,
                                          feature_description)
```

As you can see, the `Example` proto will probably be sufficient for most use cases. However, it may be a bit cumbersome to use when you are dealing with lists of lists. For example, suppose you want to classify text documents. Each document may be represented as a list of sentences, where each sentence is represented as a list of words. And perhaps each document also has a list of comments, where each comment is also represented as a list of words. Moreover, there may be some contextual data as well, such as the document's author, title and publication date. TensorFlow's `SequenceExample` protobuf is designed for such use cases.

Handling Lists of Lists Using the `SequenceExample` Protobuf

Here is the definition of the `SequenceExample` protobuf:

```
message FeatureList { repeated Feature feature = 1; };
message FeatureLists { map<string, FeatureList> feature_list = 1; };
message SequenceExample {
    Features context = 1;
    FeatureLists feature_lists = 2;
};
```

A `SequenceExample` contains a `Features` object for the contextual data and a `FeatureLists` object which contains one or more named `FeatureList` objects (e.g., a `FeatureList` named `"content"` and another named `"comments"`). Each `FeatureList` just contains a list of `Feature` objects, each of which may be a list of byte strings, a list of 64-bit integers or a list of floats (in this example, each `Feature` would represent a sentence or a comment, perhaps in the form of a list of word identifiers). Building a `SequenceExample`, serializing it and parsing it is very similar to building, serializing and parsing an `Example`, but you must use `tf.io.parse_single_sequence_example()` to parse a single `SequenceExample` or `tf.io.parse_sequence_example()` to parse a batch, and both functions return a tuple containing the context features (as a dictionary) and the feature lists (also as a dictionary). If the feature lists contain sequences of varying sizes (as in the example above), you may want to convert them to ragged tensors using `tf.RaggedTensor.from_sparse()` (see the notebook for the full code):

```
parsed_context, parsed_feature_lists = tf.io.parse_single_sequence_example(
    serialized_sequence_example, context_feature_descriptions,
    sequence_feature_descriptions)
parsed_content = tf.RaggedTensor.from_sparse(parsed_feature_lists["content"])
```

Now that you know how to efficiently store, load and parse data, the next step is to prepare it so that it can be fed to a neural network. This means converting all features

into numerical features (ideally not too sparse), scaling them, and more. In particular, if your data contains categorical features or text features, they need to be converted to numbers. For this, the *Features API* can help.

The Features API

Preprocessing your data can be performed in many ways: it can be done ahead of time when preparing your data files, using any tool you like. Or you can preprocess your data on the fly when loading it with the Data API (e.g., using the dataset's `map()` method, as we saw earlier). Or you can include a preprocessing layer directly in your model. Whichever solution you prefer, the Features API can help you: it is a set of functions available in the `tf.feature_column` package, which let you define how each feature (or group of features) in your data should be preprocessed (therefore you can think of this API as the analog of Scikit-Learn's `ColumnTransformer` class). We will start by looking at the different types of columns available, and then we will look at how to use them.

Let's go back to the variant of the California housing dataset that we used in Chapter 2, since it includes a categorical feature and missing data. Here is a simple numerical column named `"housing_median_age"`:

```
housing_median_age = tf.feature_column.numeric_column("housing_median_age")
```

Numeric columns let you specify a normalization function using the `normalizer_fn` argument. For example, let's tweak the `"housing_median_age"` column to define how it should be scaled. Note that this requires computing ahead of time the mean and standard deviation of this feature in the training set:

```
age_mean, age_std = X_mean[1], X_std[1]  # The median age is column in 1
housing_median_age = tf.feature_column.numeric_column(
    "housing_median_age", normalizer_fn=lambda x: (x - age_mean) / age_std)
```

In some cases, it might improve performance to bucketize some numerical features, effectively transforming a numerical feature into a categorical feature. For example, let's create a bucketized column based on the `median_income` column, with 5 buckets: less than 1.5 ($15,000), then 1.5 to 3, 3 to 4.5, 4.5 to 6., and above 6. (notice that when you specify 4 boundaries, there are actually 5 buckets):

```
median_income = tf.feature_column.numeric_column("median_income")
bucketized_income = tf.feature_column.bucketized_column(
    median_income, boundaries=[1.5, 3., 4.5, 6.])
```

If the `median_income` feature is equal to, say, 3.2, then the `bucketized_income` feature will automatically be equal to 2 (i.e., the index of the corresponding income bucket). Choosing the right boundaries can be somewhat of an art, but one approach is to just use percentiles of the data (e.g., the 10th percentile, the 20th percentile, and so on). If a feature is *multimodal*, meaning it has separate peaks in its distribution, you may

want to define a bucket for each mode, placing the boundaries in between the peaks. Whether you use the percentiles or the modes, you need to analyze the distribution of your data ahead of time, just like we had to measure the mean and standard deviation ahead of time to normalize the `housing_median_age` column.

Categorical Features

For categorical features such as `ocean_proximity`, there are several options. If it is already represented as a category ID (i.e., an integer from 0 to the max ID), then you can use the `categorical_column_with_identity()` function (specifying the max ID). If not, and you know the list of all possible categories, then you can use `categorical_column_with_vocabulary_list()`:

```
ocean_prox_vocab = ['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN']
ocean_proximity = tf.feature_column.categorical_column_with_vocabulary_list(
    "ocean_proximity", ocean_prox_vocab)
```

If you prefer to have TensorFlow load the vocabulary from a file, you can call `categorical_column_with_vocabulary_file()` instead. As you might expect, these two functions will simply map each category to its index in the vocabulary (e.g., *NEAR BAY* will be mapped to 3), and unknown categories will be mapped to -1.

For categorical columns with a large vocabulary (e.g., for zipcodes, cities, words, products, users, etc.), it may not be convenient to get the full list of possible categories, or perhaps categories may be added or removed so frequently that using category indices would be too unreliable. In this case, you may prefer to use a `categorical_column_with_hash_bucket()`. If we had a `"city"` feature in the dataset, we could encode it like this:

```
city_hash = tf.feature_column.categorical_column_with_hash_bucket(
    "city", hash_bucket_size=1000)
```

This feature will compute a hash for each category (i.e., for each city), modulo the number of hash buckets (`hash_bucket_size`). You must set the number of buckets high enough to avoid getting too many collisions (i.e., different categories ending up in the same bucket), but the higher you set it, the more RAM will be used (by the embedding table, as we will see shortly).

Crossed Categorical Features

If you suspect that two (or more) categorical features are more meaningful when used jointly, then you can create a *crossed column*. For example, suppose people are particularly fond of old houses inland and new houses near the ocean, then it might help to

create a bucketized column for the `housing_median_age` feature[9], and cross it with the `ocean_proximity` column. The crossed column will compute a hash of every age & ocean proximity combination it comes across, modulo the `hash_bucket_size`, and this will give it the cross category ID. You may then choose to use only this crossed column in your model, or also include the individual columns.

```
bucketized_age = tf.feature_column.bucketized_column(
    housing_median_age, boundaries=[-1., -0.5, 0., 0.5, 1.]) # age was scaled
age_and_ocean_proximity = tf.feature_column.crossed_column(
    [bucketized_age, ocean_proximity], hash_bucket_size=100)
```

Another common use case for crossed columns is to cross latitude and longitude into a single categorical feature: you start by bucketizing the latitude and longitude, for example into 20 buckets each, then you cross these bucketized features into a `location` column. This will create a 20×20 grid over California, and each cell in the grid will correspond to one category:

```
latitude = tf.feature_column.numeric_column("latitude")
longitude = tf.feature_column.numeric_column("longitude")
bucketized_latitude = tf.feature_column.bucketized_column(
    latitude, boundaries=list(np.linspace(32., 42., 20 - 1)))
bucketized_longitude = tf.feature_column.bucketized_column(
    longitude, boundaries=list(np.linspace(-125., -114., 20 - 1)))
location = tf.feature_column.crossed_column(
    [bucketized_latitude, bucketized_longitude], hash_bucket_size=1000)
```

Encoding Categorical Features Using One-Hot Vectors

No matter which option you choose to build a categorical feature (categorical columns, bucketized columns or crossed columns), it must be encoded before you can feed it to a neural network. There are two options to encode a categorical feature: one-hot vectors or *embeddings*. For the first option, simply use the `indicator_column()` function:

```
ocean_proximity_one_hot = tf.feature_column.indicator_column(ocean_proximity)
```

A one-hot vector encoding has the size of the vocabulary length, which is fine if there are just a few possible categories, but if the vocabulary is large, you will end up with too many inputs fed to your neural network: it will have too many weights to learn and it will probably not perform very well. In particular, this will typically be the case when you use hash buckets. In this case, you should probably encode them using *embeddings* instead.

9 Since the `housing_median_age` feature was normalized, the boundaries are for normalized ages.

As a rule of thumb (but your mileage may vary!), if the number of categories is lower than 10, then one-hot encoding is generally the way to go. If the number of categories is greater than 50 (which is often the case when you use hash buckets), then embeddings are usually preferable. In between 10 and 50 categories, you may want to experiment with both options and see which one works best for your use case. Also, embeddings typically require more training data, unless you can reuse pretrained embeddings.

Encoding Categorical Features Using Embeddings

An embedding is a trainable dense vector that represents a category. By default, embeddings are initialized randomly, so for example the `"NEAR BAY"` category could be represented initially by a random vector such as `[0.131, 0.890]`, while the `"NEAR OCEAN"` category may be represented by another random vector such as `[0.631, 0.791]` (in this example, we are using 2D embeddings, but the number of dimensions is a hyperparameter you can tweak). Since these embeddings are trainable, they will gradually improve during training, and as they represent fairly similar categories, Gradient Descent will certainly end up pushing them closer together, while it will tend to move them away from the `"INLAND"` category's embedding (see Figure 13-4). Indeed, the better the representation, the easier it will be for the neural network to make accurate predictions, so training tends to make embeddings useful representations of the categories. This is called *representation learning* (we will see other types of representation learning in ???).

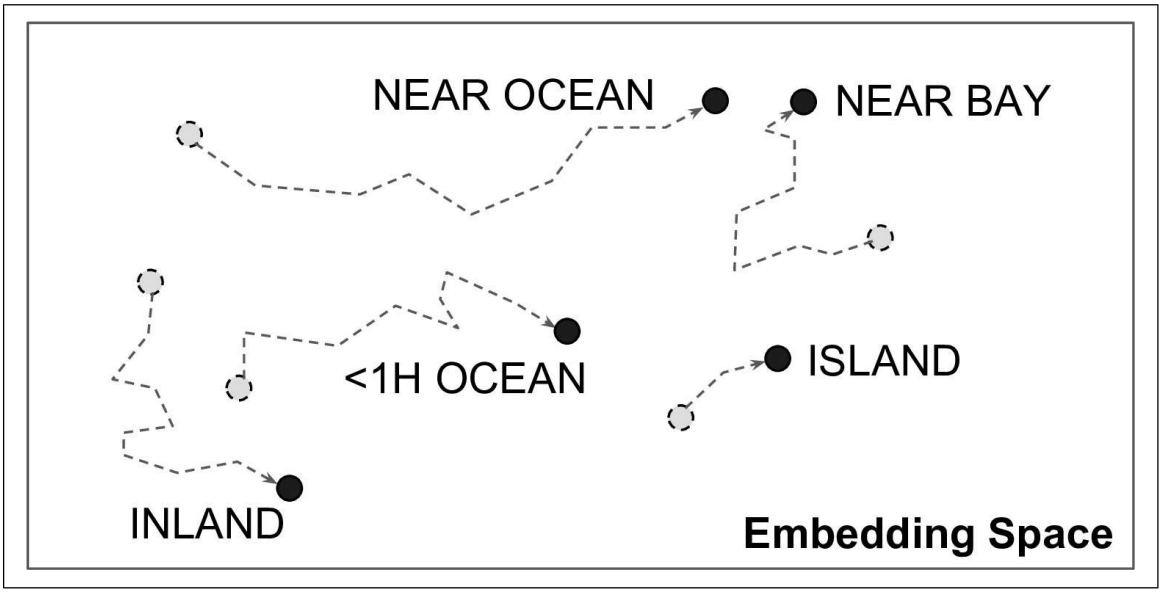

Figure 13-4. Embeddings Will Gradually Improve During Training

Word Embeddings

Not only will embeddings generally be useful representations for the task at hand, but quite often these same embeddings can be reused successfully for other tasks as well. The most common example of this is *word embeddings* (i.e., embeddings of individual words): when you are working on a natural language processing task, you are often better off reusing pretrained word embeddings than training your own. The idea of using vectors to represent words dates back to the 1960s, and many sophisticated techniques have been used to generate useful vectors, including using neural networks, but things really took off in 2013, when Tomáš Mikolov and other Google researchers published a paper[10] describing how to learn word embeddings using deep neural networks, much faster than previous attempts. This allowed them to learn embeddings on a very large corpus of text: they trained a deep neural network to predict the words near any given word. This allowed them to obtain astounding word embeddings. For example, synonyms had very close embeddings, and semantically related words such as France, Spain, Italy, and so on, ended up clustered together. But it's not just about proximity: word embeddings were also organized along meaningful axes in the embedding space. Here is a famous example: if you compute King – Man + Woman (adding and subtracting the embedding vectors of these words), then the result will be very close to the embedding of the word Queen (see Figure 13-5). In other words, the word embeddings encode the concept of gender! Similarly, you can compute Madrid – Spain + France, and of course the result is close to Paris, which seems to show that the notion of capital city was also encoded in the embeddings.

10 "Distributed Representations of Words and Phrases and their Compositionality", T. Mikolov et al. (2013).

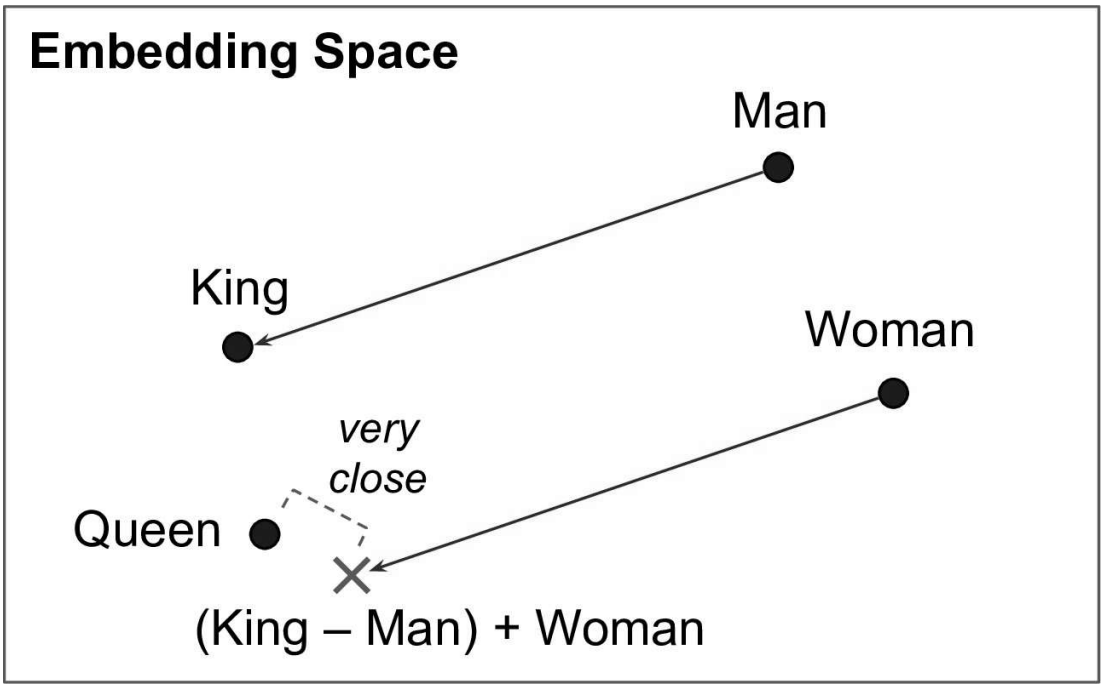

Figure 13-5. Word Embeddings

Let's go back to the Features API. Here is how you could encode the `ocean_proximity` categories as 2D embeddings:

```
ocean_proximity_embed = tf.feature_column.embedding_column(ocean_proximity,
                                                           dimension=2)
```

Each of the five `ocean_proximity` categories will now be represented as a 2D vector. These vectors are stored in an *embedding matrix* with one row per category, and one column per embedding dimension, so in this example it is a 5×2 matrix. When an embedding column is given a category index as input (say, 3, which corresponds to the category `"NEAR BAY"`), it just performs a lookup in the embedding matrix and returns the corresponding row (say, `[0.331, 0.190]`). Unfortunately, the embedding matrix can be quite large, especially when you have a large vocabulary: if this is the case, the model can only learn good representations for the categories for which it has sufficient training data. To reduce the size of the embedding matrix, you can of course try lowering the `dimension` hyperparameter, but if you reduce this parameter too much, the representations may not be as good. Another option is to reduce the vocabulary size (e.g., if you are dealing with text, you can try dropping the rare words from the vocabulary, and replace them all with a token like `"<unknown>"` or `"<UNK>"`). If you are using hash buckets, you can also try reducing the `hash_bucket_size` (but not too much, or else you will get collisions).

If there are no pretrained embeddings that you can reuse for the task you are trying to tackle, and if you do not have enough training data to learn them, then you can try to learn them on some auxiliary task for which it is easier to obtain plenty of training data. After that, you can reuse the trained embeddings for your main task.

Using Feature Columns for Parsing

Let's suppose you have created feature columns for each of your input features, as well as for the target. What can you do with them? Well, for one you can pass them to the `make_parse_example_spec()` function to generate feature descriptions (so you don't have to do it manually, as we did earlier):

```
columns = [bucketized_age, ....., median_house_value] # all features + target
feature_descriptions = tf.feature_column.make_parse_example_spec(columns)
```

You don't always have to create a separate feature column for each and every feature. For example, instead of having 2 numerical feature columns, you could choose to have a single 2D column: just set `shape=[2]` when calling `numerical_column()`.

You can then create a function that parses serialized examples using these feature descriptions, and separates the target column from the input features:

```
def parse_examples(serialized_examples):
    examples = tf.io.parse_example(serialized_examples, feature_descriptions)
    targets = examples.pop("median_house_value") # separate the targets
    return examples, targets
```

Next, you can create a `TFRecordDataset` that will read batches of serialized examples (assuming the TFRecord file contains serialized `Example` protobufs with the appropriate features):

```
batch_size = 32
dataset = tf.data.TFRecordDataset(["my_data_with_features.tfrecords"])
dataset = dataset.repeat().shuffle(10000).batch(batch_size).map(parse_examples)
```

Using Feature Columns in Your Models

Feature columns can also be used directly in your model, to convert all your input features into a single dense vector which the neural network can then process. For this, all you need to do is add a `keras.layers.DenseFeatures` layer as the first layer in your model, passing it the list of feature columns (excluding the target column):

```
columns_without_target = columns[:-1]
model = keras.models.Sequential([
    keras.layers.DenseFeatures(feature_columns=columns_without_target),
```

```
    keras.layers.Dense(1)
])
model.compile(loss="mse", optimizer="sgd", metrics=["accuracy"])
steps_per_epoch = len(X_train) // batch_size
history = model.fit(dataset, steps_per_epoch=steps_per_epoch, epochs=5)
```

The `DenseFeatures` layer will take care of converting every input feature to a dense representation, and it will also apply any extra transformation we specified, such as scaling the `housing_median_age` using the `normalizer_fn` function we provided. You can take a closer look at what the `DenseFeatures` layer does by calling it directly:

```
>>> some_columns = [ocean_proximity_embed, bucketized_income]
>>> dense_features = keras.layers.DenseFeatures(some_columns)
>>> dense_features({
...     "ocean_proximity": [["NEAR OCEAN"], ["INLAND"], ["INLAND"]],
...     "median_income": [[3.], [7.2], [1.]]
... })
...
<tf.Tensor: id=559790, shape=(3, 7), dtype=float32, numpy=
array([[ 0. , 0. , 1. , 0. , 0. ,-0.36277947 , 0.30109018],
       [ 0. , 0. , 0. , 0. , 1. , 0.22548223 , 0.33142096],
       [ 1. , 0. , 0. , 0. , 0. , 0.22548223 , 0.33142096]], dtype=float32)>
```

In this example, we create a `DenseFeatures` layer with just two columns, and we call it with some data, in the form of a dictionary of features. In this case, since the `bucketized_income` column relies on the `median_income` column, the dictionary must include the `"median_income"` key, and similarly since the `ocean_proximity_embed` column is based on the `ocean_proximity` column, the dictionary must include the `"ocean_proximity"` key. Columns are handled in alphabetical order, so first we look at the bucketized income column (its name is the same as the `median_income` column name, plus `"_bucketized"`). The incomes 3, 7.2 and 1 get mapped respectively to category 2 (for incomes between 1.5 and 3), category 0 (for incomes below 1.5), and category 4 (for incomes greater than 6). Then these category IDs get one-hot encoded: category 2 gets encoded as `[0., 0., 1., 0., 0.]` and so on (note that bucketized columns get one-hot encoded by default, no need to call `indicator_column()`). Now on to the `ocean_proximity_embed` column. The `"NEAR OCEAN"` and `"INLAND"` categories just get mapped to their respective embeddings (which were initialized randomly). The resulting tensor is the concatenation of the one-hot vectors and the embeddings.

Now you can feed all kinds of features to a neural network, including numerical features, categorical features, and even text (by splitting the text into words, then using word embedding)! However, performing all the preprocessing on the fly can slow down training. Let's see how this can be improved.

TF Transform

If preprocessing is computationally expensive, then handling it before training rather than on the fly may give you a significant speedup: the data will be preprocessed just once per instance *before* training, rather than once per instance and per epoch *during* training. Tools like Apache Beam let you run efficient data processing pipelines over large amounts of data, even distributed across multiple servers, so why not use it to preprocess all the training data? This works great and indeed can speed up training, but there is one problem: once your model is trained, suppose you want to deploy it to a mobile app: you will need to write some code in your app to take care of preprocessing the data before it is fed to the model. And suppose you also want to deploy the model to TensorFlow.js so it runs in a web browser? Once again, you will need to write some preprocessing code. This can become a maintenance nightmare: whenever you want to change the preprocessing logic, you will need to update your Apache Beam code, your mobile app code and your Javascript code. It is not only time consuming, but also error prone: you may end up with subtle differences between the preprocessing operations performed before training and the ones performed in your app or in the browser. This *training/serving skew* will lead to bugs or degraded performance.

One improvement would be to take the trained model (trained on data that was preprocessed by your Apache Beam code), and before deploying it to your app or the browser, add an extra input layer to take care of preprocessing on the fly (either by writing a custom layer or by using a `DenseFeatures` layer). That's definitely better, since now you just have two versions of your preprocessing code: the Apache Beam code and the preprocessing layer's code.

But what if you could define your preprocessing operations just once? This is what TF Transform was designed for. It is part of TensorFlow Extended (TFX), an end-to-end platform for productionizing TensorFlow models. First, to use a TFX component, such as TF Transform, you must install it, it does not come bundled with TensorFlow. You define your preprocessing function just once (in Python), by using TF Transform functions for scaling, bucketizing, crossing features, and more. You can also use any TensorFlow operation you need. Here is what this preprocessing function might look like if we just had two features:

```
import tensorflow_transform as tft

def preprocess(inputs):  # inputs is a batch of input features
    median_age = inputs["housing_median_age"]
    ocean_proximity = inputs["ocean_proximity"]
    standardized_age = tft.scale_to_z_score(median_age - tft.mean(median_age))
    ocean_proximity_id = tft.compute_and_apply_vocabulary(ocean_proximity)
    return {
        "standardized_median_age": standardized_age,
```

```
        "ocean_proximity_id": ocean_proximity_id
    }
```

Next, TF Transform lets you apply this `preprocess()` function to the whole training set using Apache Beam (it provides an `AnalyzeAndTransformDataset` class that you can use for this purpose in your Apache Beam pipeline). In the process, it will also compute all the necessary statistics over the whole training set: in this example, the mean and standard deviation of the `housing_median_age` feature, and the vocabulary for the `ocean_proximity` feature. The components that compute these statistics are called *analyzers*.

Importantly, TF Transform will also generate an equivalent TensorFlow Function that you can plug into the model you deploy. This TF Function contains all the necessary statistics computed by Apache Beam (the mean, standard deviation, and vocabulary), simply included as constants.

At the time of this writing, TF Transform only supports TensorFlow 1. Moreover, Apache Beam only has partial support for Python 3. That said, both these limitations will likely be fixed by the time your read this.

With the Data API, TFRecords, the Features API and TF Transform, you can build highly scalable input pipelines for training, and also benefit from fast and portable data preprocessing in production.

But what if you just wanted to use a standard dataset? Well in that case, things are much simpler: just use TFDS!

The TensorFlow Datasets (TFDS) Project

The TensorFlow Datasets project makes it trivial to download common datasets, from small ones like MNIST or Fashion MNIST, to huge datasets like ImageNet[11] (you will need quite a bit of disk space!). The list includes image datasets, text datasets (including translation datasets), audio and video datasets, and more. You can visit *https://homl.info/tfds* to view the full list, along with a description of each dataset.

TFDS is not bundled with TensorFlow, so you need to install the `tensorflow-datasets` library (e.g., using pip). Then all you need to do is call the `tfds.load()` function, and it will download the data you want (unless it was already downloaded earlier), and return the data as a dictionary of `Datasets` (typically one for training,

11 At the time of writing, TFDS requires you to download a few files manually for ImageNet (for legal reasons), but this will hopefully get resolved soon.

and one for testing, but this depends on the dataset you choose). For example, let's download MNIST:

```
import tensorflow_datasets as tfds

dataset = tfds.load(name="mnist")
mnist_train, mnist_test = dataset["train"], dataset["test"]
```

You can then apply any transformation you want (typically repeating, batching and prefetching), and you're ready to train your model. Here is a simple example:

```
mnist_train = mnist_train.repeat(5).batch(32).prefetch(1)
for item in mnist_train:
    images = item["image"]
    labels = item["label"]
    [...]
```

In general, `load()` returns a shuffled training set, so there's no need to shuffle it some more.

Note that each item in the dataset is a dictionary containing both the features and the labels. But Keras expects each item to be a tuple containing 2 elements (again, the features and the labels). You could transform the dataset using the `map()` method, like this:

```
mnist_train = mnist_train.repeat(5).batch(32)
mnist_train = mnist_train.map(lambda items: (items["image"], items["label"]))
mnist_train = mnist_train.prefetch(1)
```

Or you can just ask the `load()` function to do this for you by setting `as_supervised=True` (obviously this works only for labeled datasets). You can also specify the batch size if you want. Then the dataset can be passed directly to your tf.keras model:

```
dataset = tfds.load(name="mnist", batch_size=32, as_supervised=True)
mnist_train = dataset["train"].repeat().prefetch(1)
model = keras.models.Sequential([...])
model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd")
model.fit(mnist_train, steps_per_epoch=60000 // 32, epochs=5)
```

This was quite a technical chapter, and you may feel that it is a bit far from the abstract beauty of neural networks, but the fact is deep learning often involves large amounts of data, and knowing how to load, parse and preprocess it efficiently is a crucial skill to have. In the next chapter, we will look at Convolutional Neural Networks, which are among the most successful neural net architectures for image processing, and many other applications.

CHAPTER 14

Deep Computer Vision Using Convolutional Neural Networks

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 14 in the final release of the book.

Although IBM's Deep Blue supercomputer beat the chess world champion Garry Kasparov back in 1996, it wasn't until fairly recently that computers were able to reliably perform seemingly trivial tasks such as detecting a puppy in a picture or recognizing spoken words. Why are these tasks so effortless to us humans? The answer lies in the fact that perception largely takes place outside the realm of our consciousness, within specialized visual, auditory, and other sensory modules in our brains. By the time sensory information reaches our consciousness, it is already adorned with high-level features; for example, when you look at a picture of a cute puppy, you cannot choose *not* to see the puppy, or *not* to notice its cuteness. Nor can you explain *how* you recognize a cute puppy; it's just obvious to you. Thus, we cannot trust our subjective experience: perception is not trivial at all, and to understand it we must look at how the sensory modules work.

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and they have been used in image recognition since the 1980s. In the last few years, thanks to the increase in computational power, the amount of available training data, and the tricks presented in Chapter 11 for training deep nets, CNNs have managed to achieve superhuman performance on some complex visual tasks. They power image search services, self-driving cars, automatic video classification systems, and more. Moreover, CNNs are not restricted to visual perception: they are also successful

at many other tasks, such as *voice recognition* or *natural language processing* (NLP); however, we will focus on visual applications for now.

In this chapter we will present where CNNs came from, what their building blocks look like, and how to implement them using TensorFlow and Keras. Then we will discuss some of the best CNN architectures, and discuss other visual tasks, including *object detection* (classifying multiple objects in an image and placing bounding boxes around them) and *semantic segmentation* (classifying each pixel according to the class of the object it belongs to).

The Architecture of the Visual Cortex

David H. Hubel and Torsten Wiesel performed a series of experiments on cats in 1958[1] and 1959[2] (and a few years later on monkeys[3]), giving crucial insights on the structure of the visual cortex (the authors received the Nobel Prize in Physiology or Medicine in 1981 for their work). In particular, they showed that many neurons in the visual cortex have a small *local receptive field*, meaning they react only to visual stimuli located in a limited region of the visual field (see Figure 14-1, in which the local receptive fields of five neurons are represented by dashed circles). The receptive fields of different neurons may overlap, and together they tile the whole visual field. Moreover, the authors showed that some neurons react only to images of horizontal lines, while others react only to lines with different orientations (two neurons may have the same receptive field but react to different line orientations). They also noticed that some neurons have larger receptive fields, and they react to more complex patterns that are combinations of the lower-level patterns. These observations led to the idea that the higher-level neurons are based on the outputs of neighboring lower-level neurons (in Figure 14-1, notice that each neuron is connected only to a few neurons from the previous layer). This powerful architecture is able to detect all sorts of complex patterns in any area of the visual field.

1 "Single Unit Activity in Striate Cortex of Unrestrained Cats," D. Hubel and T. Wiesel (1958).

2 "Receptive Fields of Single Neurones in the Cat's Striate Cortex," D. Hubel and T. Wiesel (1959).

3 "Receptive Fields and Functional Architecture of Monkey Striate Cortex," D. Hubel and T. Wiesel (1968).

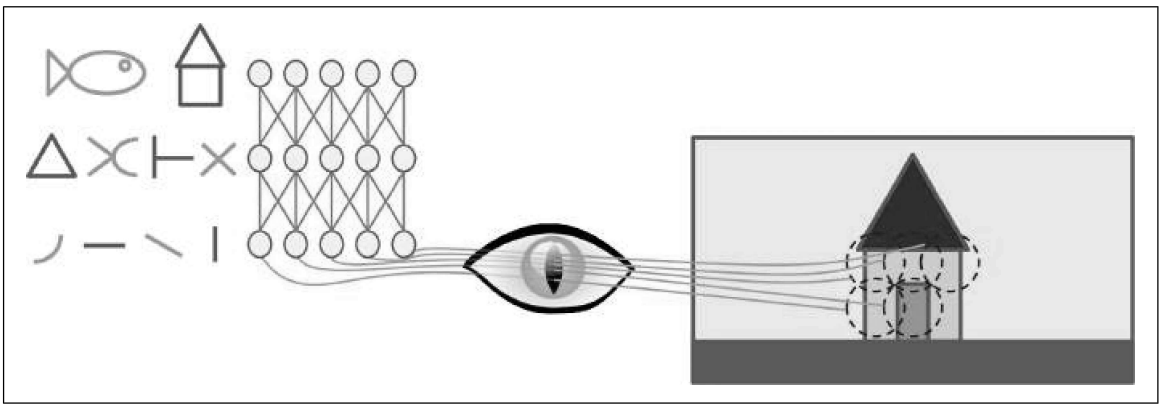

Figure 14-1. Local receptive fields in the visual cortex

These studies of the visual cortex inspired the neocognitron, introduced in 1980,[4] which gradually evolved into what we now call *convolutional neural networks*. An important milestone was a 1998 paper[5] by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, which introduced the famous *LeNet-5* architecture, widely used to recognize handwritten check numbers. This architecture has some building blocks that you already know, such as fully connected layers and sigmoid activation functions, but it also introduces two new building blocks: *convolutional layers* and *pooling layers*. Let's look at them now.

Why not simply use a regular deep neural network with fully connected layers for image recognition tasks? Unfortunately, although this works fine for small images (e.g., MNIST), it breaks down for larger images because of the huge number of parameters it requires. For example, a 100 × 100 image has 10,000 pixels, and if the first layer has just 1,000 neurons (which already severely restricts the amount of information transmitted to the next layer), this means a total of 10 million connections. And that's just the first layer. CNNs solve this problem using partially connected layers and weight sharing.

4 "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position," K. Fukushima (1980).

5 "Gradient-Based Learning Applied to Document Recognition," Y. LeCun et al. (1998).

Convolutional Layer

The most important building block of a CNN is the *convolutional layer*:[6] neurons in the first convolutional layer are not connected to every single pixel in the input image (like they were in previous chapters), but only to pixels in their receptive fields (see Figure 14-2). In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer. This architecture allows the network to concentrate on small low-level features in the first hidden layer, then assemble them into larger higher-level features in the next hidden layer, and so on. This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition.

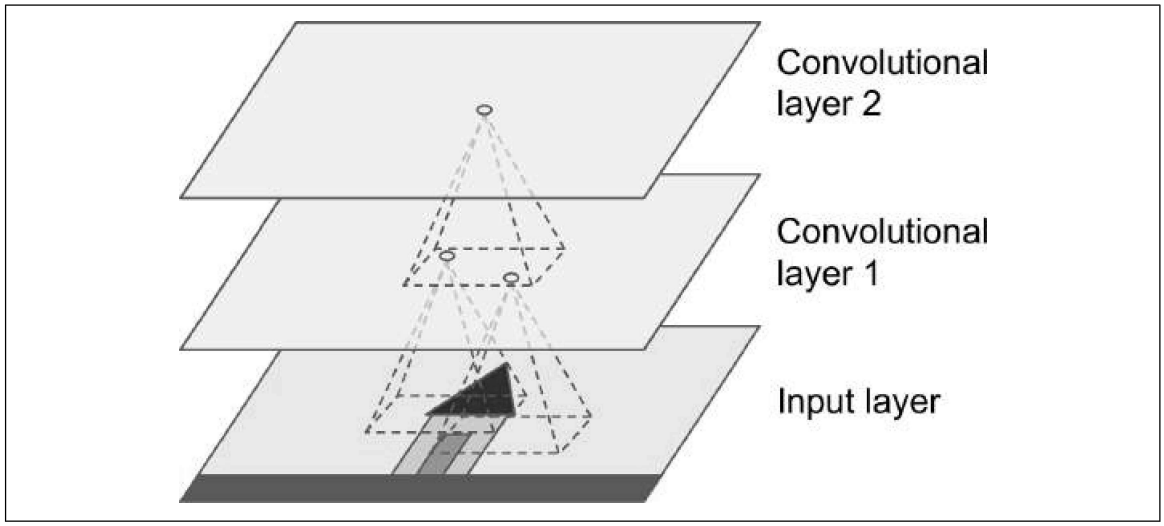

Figure 14-2. CNN layers with rectangular local receptive fields

Until now, all multilayer neural networks we looked at had layers composed of a long line of neurons, and we had to flatten input images to 1D before feeding them to the neural network. Now each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs.

A neuron located in row i, column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$, where f_h and f_w are the height and width of the receptive field (see Figure 14-3). In order for a layer to have the same height and width as the previous layer, it is com-

6 A convolution is a mathematical operation that slides one function over another and measures the integral of their pointwise multiplication. It has deep connections with the Fourier transform and the Laplace transform, and is heavily used in signal processing. Convolutional layers actually use cross-correlations, which are very similar to convolutions (see *https://homl.info/76* for more details).