```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

For example, the first image in the training set represents a coat:

```
>>> class_names[y_train[0]]
'Coat'
```

Figure 10-11 shows a few samples from the Fashion MNIST dataset:

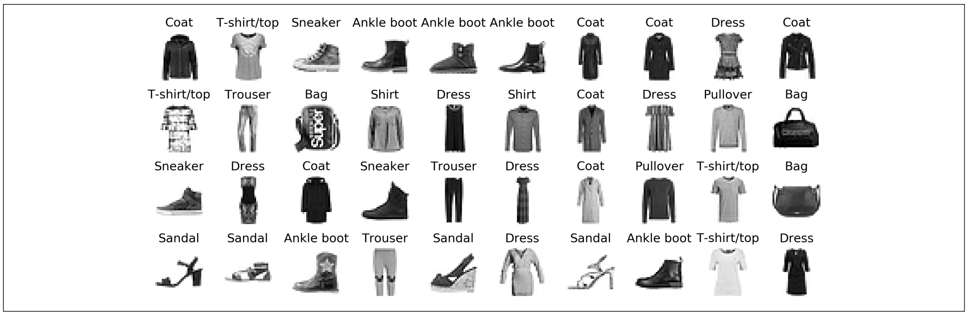

Figure 10-11. Samples from Fashion MNIST

Creating the Model Using the Sequential API

Now let's build the neural network! Here is a classification MLP with two hidden layers:

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

Let's go through this code line by line:

- The first line creates a `Sequential` model. This is the simplest kind of Keras model, for neural networks that are just composed of a single stack of layers, connected sequentially. This is called the sequential API.
- Next, we build the first layer and add it to the model. It is a `Flatten` layer whose role is simply to convert each input image into a 1D array: if it receives input data `X`, it computes `X.reshape(-1, 1)`. This layer does not have any parameters, it is just there to do some simple preprocessing. Since it is the first layer in the model, you should specify the `input_shape`: this does not include the batch size, only the shape of the instances. Alternatively, you could add a `keras.layers.InputLayer` as the first layer, setting `shape=[28,28]`.
- Next we add a `Dense` hidden layer with 300 neurons. It will use the ReLU activation function. Each `Dense` layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs. It also manages a vec-

tor of bias terms (one per neuron). When it receives some input data, it computes Equation 10-2.

- Next we add a second `Dense` hidden layer with 100 neurons, also using the ReLU activation function.
- Finally, we add a `Dense` output layer with 10 neurons (one per class), using the softmax activation function (because the classes are exclusive).

Specifying `activation="relu"` is equivalent to `activation=keras.activations.relu`. Other activation functions are available in the `keras.activations` package, we will use many of them in this book. See *https://keras.io/activations/* for the full list.

Instead of adding the layers one by one as we just did, you can pass a list of layers when creating the `Sequential` model:

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

Using Code Examples From keras.io

Code examples documented on keras.io will work fine with tf.keras, but you need to change the imports. For example, consider this keras.io code:

```
from keras.layers import Dense
output_layer = Dense(10)
```

You must change the imports like this:

```
from tensorflow.keras.layers import Dense
output_layer = Dense(10)
```

Or simply use full paths, if you prefer:

```
from tensorflow import keras
output_layer = keras.layers.Dense(10)
```

This is more verbose, but I use this approach in this book so you can easily see which packages to use, and to avoid confusion between standard classes and custom classes. In production code, I use the previous approach, as do most people.

The model's `summary()` method displays all the model's layers[13], including each layer's name (which is automatically generated unless you set it when creating the layer), its output shape (`None` means the batch size can be anything), and its number of parameters. The summary ends with the total number of parameters, including trainable and non-trainable parameters. Here we only have trainable parameters (we will see examples of non-trainable parameters in Chapter 11):

```
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_3 (Dense)              (None, 300)               235500
_________________________________________________________________
dense_4 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
```

Note that `Dense` layers often have a *lot* of parameters. For example, the first hidden layer has 784 × 300 connection weights, plus 300 bias terms, which adds up to 235,500 parameters! This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the risk of overfitting, especially when you do not have a lot of training data. We will come back to this later.

You can easily get a model's list of layers, to fetch a layer by its index, or you can fetch it by name:

```
>>> model.layers
[<tensorflow.python.keras.layers.core.Flatten at 0x132414e48>,
 <tensorflow.python.keras.layers.core.Dense at 0x1324149b0>,
 <tensorflow.python.keras.layers.core.Dense at 0x1356ba8d0>,
 <tensorflow.python.keras.layers.core.Dense at 0x13240d240>]
>>> model.layers[1].name
'dense_3'
>>> model.get_layer('dense_3').name
'dense_3'
```

All the parameters of a layer can be accessed using its `get_weights()` and `set_weights()` method. For a `Dense` layer, this includes both the connection weights and the bias terms:

13 You can also generate an image of your model using `keras.utils.plot_model()`.

```
>>> weights, biases = hidden1.get_weights()
>>> weights
array([[ 0.03854964, -0.04054524,  0.00599282, ...,  0.02566582,
         0.01032123,  0.06914985],
       ...,
       [ 0.02632413, -0.05105981, -0.00332005, ...,  0.04175945,
         0.0443138 , -0.05558084]], dtype=float32)
>>> weights.shape
(784, 300)
>>> biases
array([0., 0., 0., 0., 0., 0., 0., 0., 0., ...,  0., 0., 0.], dtype=float32)
>>> biases.shape
(300,)
```

Notice that the `Dense` layer initialized the connection weights randomly (which is needed to break symmetry, as we discussed earlier), and the biases were just initialized to zeros, which is fine. If you ever want to use a different initialization method, you can set `kernel_initializer` (*kernel* is another name for the matrix of connection weights) or `bias_initializer` when creating the layer. We will discuss initializers further in Chapter 11, but if you want the full list, see *https://keras.io/initializers/*.

The shape of the weight matrix depends on the number of inputs. This is why it is recommended to specify the `input_shape` when creating the first layer in a `Sequential` model. However, if you do not specify the input shape, it's okay: Keras will simply wait until it knows the input shape before it actually builds the model. This will happen either when you feed it actual data (e.g., during training), or when you call its `build()` method. Until the model is really built, the layers will not have any weights, and you will not be able to do certain things (such as print the model summary or save the model), so if you know the input shape when creating the model, it is best to specify it.

Compiling the Model

After a model is created, you must call its `compile()` method to specify the loss function and the optimizer to use. Optionally, you can also specify a list of extra metrics to compute during training and evaluation:

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

Using `loss="sparse_categorical_crossentropy"` is equivalent to `loss=keras.losses.sparse_categorical_crossentropy`. Similarly, `optimizer="sgd"` is equivalent to `optimizer=keras.optimizers.SGD()` and `metrics=["accuracy"]` is equivalent to `metrics=[keras.metrics.sparse_categorical_accuracy]` (when using this loss). We will use many other losses, optimizers and metrics in this book, but for the full lists see *https://keras.io/losses/*, *https://keras.io/optimizers/* and *https://keras.io/metrics/*.

This requires some explanation. First, we use the `"sparse_categorical_crossentropy"` loss because we have sparse labels (i.e., for each instance there is just a target class index, from 0 to 9 in this case), and the classes are exclusive. If instead we had one target probability per class for each instance (such as one-hot vectors, e.g. `[0., 0., 0., 1., 0., 0., 0., 0., 0., 0.]` to represent class 3), then we would need to use the `"categorical_crossentropy"` loss instead. If we were doing binary classification (with one or more binary labels), then we would use the `"sigmoid"` (i.e., logistic) activation function in the output layer instead of the `"softmax"` activation function, and we would use the `"binary_crossentropy"` loss.

If you want to convert sparse labels (i.e., class indices) to one-hot vector labels, you can use the `keras.utils.to_categorical()` function. To go the other way round, you can just use the `np.argmax()` function with `axis=1`.

Secondly, regarding the optimizer, `"sgd"` simply means that we will train the model using simple Stochastic Gradient Descent. In other words, Keras will perform the backpropagation algorithm described earlier (i.e., reverse-mode autodiff + Gradient Descent). We will discuss more efficient optimizers in Chapter 11 (they improve the Gradient Descent part, not the autodiff).

Finally, since this is a classifier, it's useful to measure its `"accuracy"` during training and evaluation.

Training and Evaluating the Model

Now the model is ready to be trained. For this we simply need to call its `fit()` method. We pass it the input features (`X_train`) and the target classes (`y_train`), as well as the number of epochs to train (or else it would default to just 1, which would definitely not be enough to converge to a good solution). We also pass a validation set (this is optional): Keras will measure the loss and the extra metrics on this set at the end of each epoch, which is very useful to see how well the model really performs: if the performance on the training set is much better than on the validation set, your model is probably overfitting the training set (or there is a bug, such as a data mismatch between the training set and the validation set):

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [==========] - 3s 55us/sample - loss: 1.4948     - acc: 0.5757
                                          - val_loss: 1.0042 - val_acc: 0.7166
Epoch 2/30
55000/55000 [==========] - 3s 55us/sample - loss: 0.8690     - acc: 0.7318
                                          - val_loss: 0.7549 - val_acc: 0.7616
[...]
Epoch 50/50
55000/55000 [==========] - 4s 72us/sample - loss: 0.3607     - acc: 0.8752
                                          - val_loss: 0.3706 - val_acc: 0.8728
```

And that's it! The neural network is trained. At each epoch during training, Keras displays the number of instances processed so far (along with a progress bar), the mean training time per sample, the loss and accuracy (or any other extra metrics you asked for), both on the training set and the validation set. You can see that the training loss went down, which is a good sign, and the validation accuracy reached 87.28% after 50 epochs, not too far from the training accuracy, so there does not seem to be much overfitting going on.

Instead of passing a validation set using the `validation_data` argument, you could instead set `validation_split` to the ratio of the training set that you want Keras to use for validation (e.g., 0.1).

If the training set was very skewed, with some classes being overrepresented and others underrepresented, it would be useful to set the `class_weight` argument when calling the `fit()` method, giving a larger weight to underrepresented classes, and a lower weight to overrepresented classes. These weights would be used by Keras when computing the loss. If you need per-instance weights instead, you can set the `sample_weight` argument (it supersedes `class_weight`). This could be useful for example if some instances were labeled by experts while others were labeled using a crowdsourcing platform: you might want to give more weight to the former. You can also provide sample weights (but not class weights) for the validation set by adding them as a third item in the `validation_data` tuple.

The `fit()` method returns a `History` object containing the training parameters (`history.params`), the list of epochs it went through (`history.epoch`), and most importantly a dictionary (`history.history`) containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if any). If you create a Pandas DataFrame using this dictionary and call its `plot()` method, you get the learning curves shown in Figure 10-12:

```
import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # set the vertical range to [0-1]
plt.show()
```

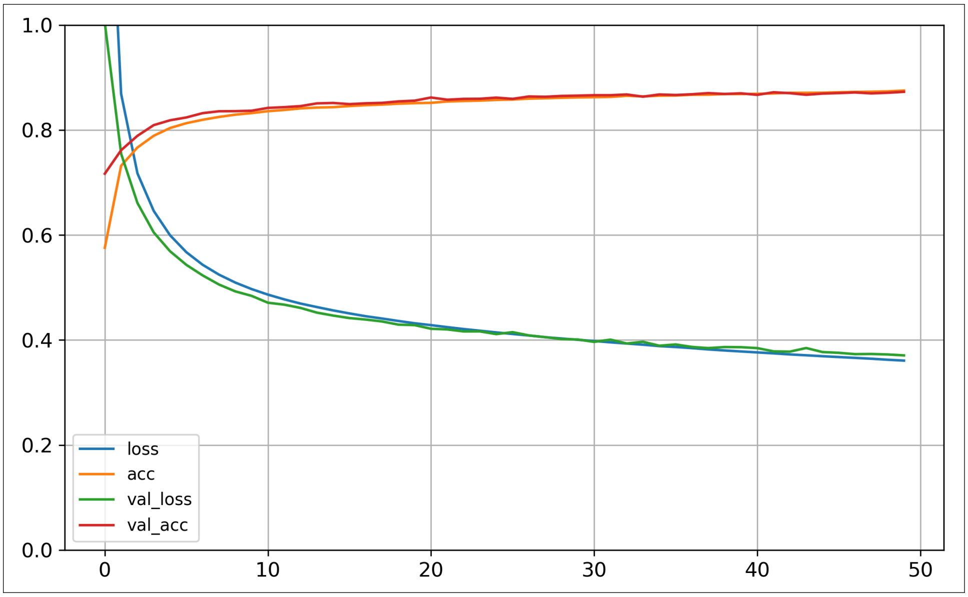

Figure 10-12. Learning Curves

You can see that both the training and validation accuracy steadily increase during training, while the training and validation loss decrease. Good! Moreover, the validation curves are quite close to the training curves, which means that there is not too much overfitting. In this particular case, the model performed better on the validation set than on the training set at the beginning of training: this sometimes happens by chance (especially when the validation set is fairly small). However, the training set performance ends up beating the validation performance, as is generally the case when you train for long enough. You can tell that the model has not quite converged yet, as the validation loss is still going down, so you should probably continue training. It's as simple as calling the `fit()` method again, since Keras just continues training where it left off (you should be able to reach close to 89% validation accuracy).

If you are not satisfied with the performance of your model, you should go back and tune the model's hyperparameters, for example the number of layers, the number of neurons per layer, the types of activation functions we use for each hidden layer, the number of training epochs, the batch size (it can be set in the `fit()` method using the `batch_size` argument, which defaults to 32). We will get back to hyperparameter tuning at the end of this chapter. Once you are satisfied with your model's validation accuracy, you should evaluate it on the test set to estimate the generalization error before you deploy the model to production. You can easily do this using the `evaluate()` method (it also supports several other arguments, such as `batch_size` or `sample_weight`, please check the documentation for more details):

```
>>> model.evaluate(X_test, y_test)
8832/10000 [==========================] - ETA: 0s - loss: 0.4074 - acc: 0.8540
[0.40738476498126985, 0.854]
```

As we saw in Chapter 2, it is common to get slightly lower performance on the test set than on the validation set, because the hyperparameters are tuned on the validation set, not the test set (however, in this example, we did not do any hyperparameter tuning, so the lower accuracy is just bad luck). Remember to resist the temptation to tweak the hyperparameters on the test set, or else your estimate of the generalization error will be too optimistic.

Using the Model to Make Predictions

Next, we can use the model's `predict()` method to make predictions on new instances. Since we don't have actual new instances, we will just use the first 3 instances of the test set:

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.09, 0.  , 0.12, 0.  , 0.79],
       [0.  , 0.  , 0.94, 0.  , 0.02, 0.  , 0.04, 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

As you can see, for each instance the model estimates one probability per class, from class 0 to class 9. For example, for the first image it estimates that the probability of class 9 (ankle boot) is 79%, the probability of class 7 (sneaker) is 12%, the probability of class 5 (sandal) is 9%, and the other classes are negligible. In other words, it "believes" it's footwear, probably ankle boots, but it's not entirely sure, it might be sneakers or sandals instead. If you only care about the class with the highest estimated probability (even if that probability is quite low) then you can use the `predict_classes()` method instead:

```
>>> y_pred = model.predict_classes(X_new)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

And the classifier actually classified all three images correctly:

```
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1])
```

Now you know how to build, train, evaluate and use a classification MLP using the Sequential API. But what about regression?

Building a Regression MLP Using the Sequential API

Let's switch to the California housing problem and tackle it using a regression neural network. For simplicity, we will use Scikit-Learn's `fetch_california_housing()` function to load the data: this dataset is simpler than the one we used in Chapter 2, since it contains only numerical features (there is no `ocean_proximity` feature), and there is no missing value. After loading the data, we split it into a training set, a validation set and a test set, and we scale all the features:

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()

X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_valid_scaled = scaler.transform(X_valid)
X_test_scaled = scaler.transform(X_test)
```

Building, training, evaluating and using a regression MLP using the Sequential API to make predictions is quite similar to what we did for classification. The main differences are the fact that the output layer has a single neuron (since we only want to predict a single value) and uses no activation function, and the loss function is the mean squared error. Since the dataset is quite noisy, we just use a single hidden layer with fewer neurons than before, to avoid overfitting:

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])
model.compile(loss="mean_squared_error", optimizer="sgd")
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3] # pretend these are new instances
y_pred = model.predict(X_new)
```

As you can see, the Sequential API is quite easy to use. However, although sequential models are extremely common, it is sometimes useful to build neural networks with more complex topologies, or with multiple inputs or outputs. For this purpose, Keras offers the Functional API.

Building Complex Models Using the Functional API

One example of a non-sequential neural network is a *Wide & Deep* neural network. This neural network architecture was introduced in a 2016 paper by Heng-Tze Cheng et al.[14]. It connects all or part of the inputs directly to the output layer, as shown in Figure 10-13. This architecture makes it possible for the neural network to learn both deep patterns (using the deep path) and simple rules (through the short path). In contrast, a regular MLP forces all the data to flow through the full stack of layers, thus simple patterns in the data may end up being distorted by this sequence of transformations.

14 "Wide & Deep Learning for Recommender Systems," Heng-Tze Cheng et al. (2016).