

the step-by-step guides in the appendixes, and look online if you need further help. There is no shortage of tutorials on how to install Keras and common deep-learning dependencies.

We can now dive into practical Keras examples.

### 3.4. Classifying movie reviews: a binary classification example

Two-class classification, or binary classification, may be the most widely applied kind of machine-learning problem. In this example, you'll learn to classify movie reviews as positive or negative, based on the text content of the reviews.

#### 3.4.1. The IMDB dataset

You'll work with the IMDB dataset: a set of 50,000 highly polarized reviews from the Internet Movie Database. They're split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews.

Why use separate training and test sets? Because you should never test a machine-learning model on the same data that you used to train it! Just because a model performs well on its training data doesn't mean it will perform well on data it has never seen; and what you care about is your model's performance on new data (because you already know the labels of your training data—obviously you don't need your model to predict those). For instance, it's possible that your model could end up merely *memorizing* a mapping between your training samples and their targets, which would be useless for the task of predicting targets for data the model has never seen before. We'll go over this point in much more detail in the next chapter.

Just like the MNIST dataset, the IMDB dataset comes packaged with Keras. It has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary.

The following code will load the dataset (when you run it the first time, about 80 MB of data will be downloaded to your machine).

### Listing 3.1. Loading the IMDB dataset

```
from keras.datasets import imdb

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(
    num_words=10000)
```

The argument `num_words=10000` means you'll only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows you to work with vector data of manageable size.

The variables `train_data` and `test_data` are lists of reviews; each review is a list of word indices (encoding a sequence of words). `train_labels` and `test_labels` are lists of 0s and 1s, where 0 stands for *negative* and 1 stands for *positive*:

```
>>> train_data[0]
[1, 14, 22, 16, ... 178, 32]

>>> train_labels[0]
1
```

Because you're restricting yourself to the top 10,000 most frequent words, no word index will exceed 10,000:

```
>>> max([max(sequence) for sequence in train_data])
9999
```

For kicks, here's how you can quickly decode one of these reviews back to English words:

```
word_index = imdb.get_word_index()
reverse_word_index = dict(
    [(value, key) for (key, value) in word_index.items()])
decoded_review = ' '.join(
    [reverse_word_index.get(i - 3, '?') for i in train_data[0]])
```

❶

❷

❸

❶ `word_index` is a dictionary mapping words to an integer index.

❷ Reverses it, mapping integer indices to words

③ Decodes the review. Note that the indices are offset by 3 because 0, 1, and 2 are reserved indices for “padding,” “start of sequence,” and “unknown.”

### 3.4.2. Preparing the data

You can’t feed lists of integers into a neural network. You have to turn your lists into tensors. There are two ways to do that:

- Pad your lists so that they all have the same length, turn them into an integer tensor of shape `(samples, word_indices)`, and then use as the first layer in your network a layer capable of handling such integer tensors (the Embedding layer, which we’ll cover in detail later in the book).
- One-hot encode your lists to turn them into vectors of 0s and 1s. This would mean, for instance, turning the sequence `[3, 5]` into a 10,000-dimensional vector that would be all 0s except for indices 3 and 5, which would be 1s. Then you could use as the first layer in your network a Dense layer, capable of handling floating-point vector data.

Let’s go with the latter solution to vectorize the data, which you’ll do manually for maximum clarity.

Listing 3.2. Encoding the integer sequences into a binary matrix

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))  ❶
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.  ❷
    return results

x_train = vectorize_sequences(train_data)  ❸
x_test = vectorize_sequences(test_data)  ❹
```

❶ Creates an all-zero matrix of shape `(len(sequences), dimension)`

❷ Sets specific indices of `results[i]` to 1s

❸ Vectorized training data

❹ Vectorized test data

Here’s what the samples look like now:

```
>>> x_train[0]
array([ 0.,  1.,  1., ...,  0.,  0.,  0.]
```

You should also vectorize your labels, which is straightforward:

```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

Now the data is ready to be fed into a neural network.

### 3.4.3. Building your network

The input data is vectors, and the labels are scalars (1s and 0s): this is the easiest setup you'll ever encounter. A type of network that performs well on such a problem is a simple stack of fully connected (Dense) layers with relu activations: `Dense(16, activation='relu')`.

The argument being passed to each Dense layer (16) is the number of hidden units of the layer. A *hidden unit* is a dimension in the representation space of the layer. You may remember from [chapter 2](#) that each such Dense layer with a relu activation implements the following chain of tensor operations:

```
output = relu(dot(W, input) + b)
```

Having 16 hidden units means the weight matrix  $w$  will have shape `(input_dimension, 16)`: the dot product with  $w$  will project the input data onto a 16-dimensional representation space (and then you'll add the bias vector  $b$  and apply the relu operation). You can intuitively understand the dimensionality of your representation space as “how much freedom you're allowing the network to have when learning internal representations.” Having more hidden units (a higher-dimensional representation space) allows your network to learn more-complex representations, but it makes the network more computationally expensive and may lead to learning unwanted patterns (patterns that will improve performance on the training data but not on the test data).

There are two key architecture decisions to be made about such a stack of Dense layers:

- How many layers to use
- How many hidden units to choose for each layer

In [chapter 4](#), you'll learn formal principles to guide you in making these choices. For the time being, you'll have to trust me with the following architecture choice:

- Two intermediate layers with 16 hidden units each
- A third layer that will output the scalar prediction regarding the sentiment of the current review

The intermediate layers will use `relu` as their activation function, and the final layer will use a sigmoid activation so as to output a probability (a score between 0 and 1, indicating how likely the sample is to have the target “1”: how likely the review is to be positive). A `relu` (rectified linear unit) is a function meant to zero out negative values (see [figure 3.4](#)), whereas a sigmoid “squashes” arbitrary values into the  $[0, 1]$  interval (see [figure 3.5](#)), outputting something that can be interpreted as a probability.

Figure 3.4. The rectified linear unit function

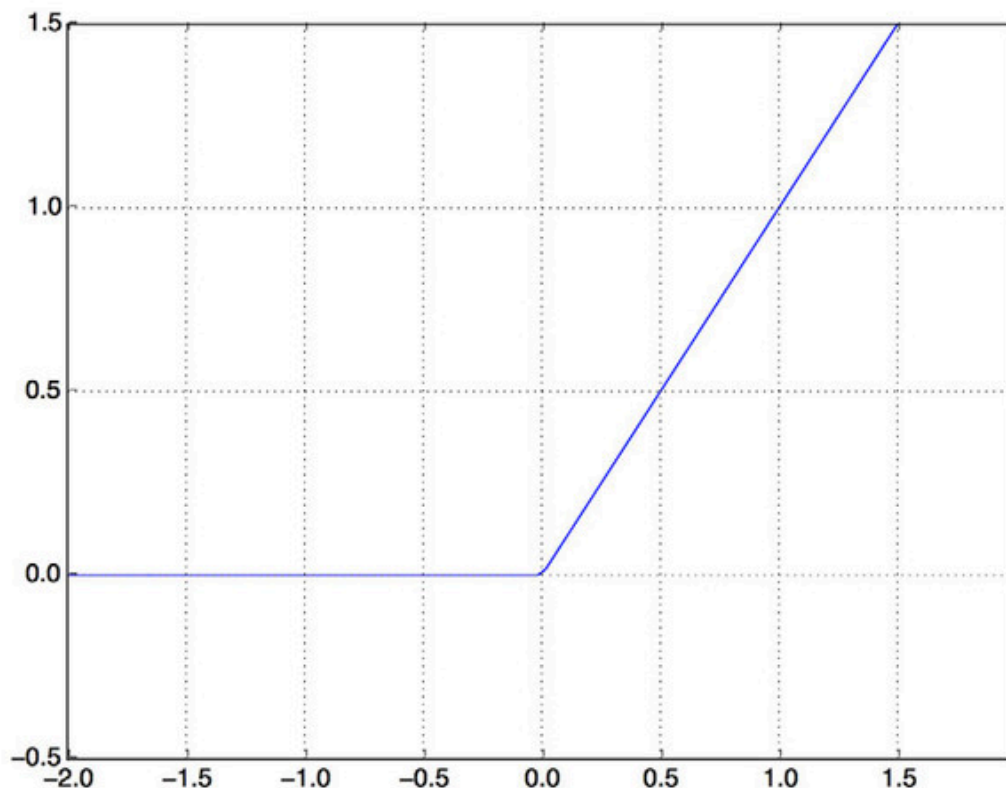
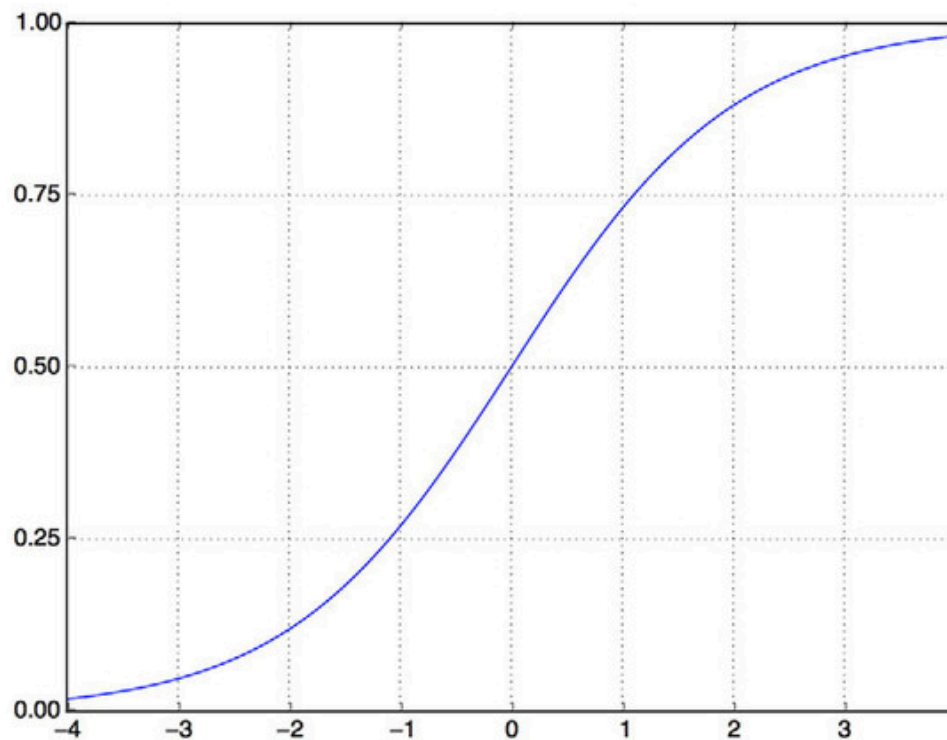
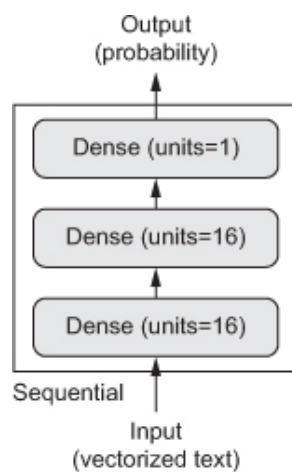


Figure 3.5. The sigmoid function



**Figure 3.6** shows what the network looks like. And here's the Keras implementation, similar to the MNIST example you saw previously.

Figure 3.6. The three-layer network



Listing 3.3. The model definition

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

---

## What are activation functions, and why are they necessary?

Without an activation function like `relu` (also called a *non-linearity*), the Dense layer would consist of two linear operations—a dot product and an addition:

```
output = dot(W, input) + b
```

So the layer could only learn *linear transformations* (affine transformations) of the input data: the *hypothesis space* of the layer would be the set of all possible linear transformations of the input data into a 16-dimensional space. Such a hypothesis space is too restricted and wouldn't benefit from multiple layers of representations, because a deep stack of linear layers would still implement a linear operation: adding more layers wouldn't extend the hypothesis space.

In order to get access to a much richer hypothesis space that would benefit from deep representations, you need a non-linearity, or activation function. `relu` is the most popular activation function in deep learning, but there are many other candidates, which all come with similarly strange names: `prelu`, `elu`, and so on.

---

Finally, you need to choose a loss function and an optimizer. Because you're facing a binary classification problem and the output of your network is a probability (you end your network with a single-unit layer with a sigmoid activation), it's best to use the `binary_crossentropy` loss. It isn't the only viable choice: you could use, for instance, `mean_squared_error`. But crossentropy is usually the best choice when you're dealing with models that output probabilities. *Crossentropy* is a quantity from the field of Information Theory that measures the distance between probability distributions or, in this case, between the ground-truth distribution and your predictions.

Here's the step where you configure the model with the `rmsprop` optimizer and the `binary_crossentropy` loss function. Note that you'll also monitor accuracy during training.

Listing 3.4. Compiling the model

```
model.compile(optimizer='rmsprop',  
              loss='binary_crossentropy',  
              metrics=['accuracy'])
```

You're passing your optimizer, loss function, and metrics as strings, which is possible because `rmsprop`, `binary_crossentropy`, and `accuracy` are packaged as part of Keras. Sometimes you may want to configure the parameters of your optimizer or pass a custom loss function or metric function. The former can be done by passing an optimizer class instance as the optimizer argument, as shown in [listing 3.5](#); the latter can be done by passing function objects as the loss and/or metrics arguments, as shown in [listing 3.6](#).

Listing 3.5. Configuring the optimizer

```
from keras import optimizers  
  
model.compile(optimizer=optimizers.RMSprop(lr=0.001),  
              loss='binary_crossentropy',  
              metrics=['accuracy'])
```

Listing 3.6. Using custom losses and metrics

```
from keras import losses  
from keras import metrics  
  
model.compile(optimizer=optimizers.RMSprop(lr=0.001),  
              loss=losses.binary_crossentropy,  
              metrics=[metrics.binary_accuracy])
```

### 3.4.4. Validating your approach

In order to monitor during training the accuracy of the model on data it has never seen before, you'll create a validation set by setting apart 10,000 samples from the original training data.

Listing 3.7. Setting aside a validation set

```
x_val = x_train[:10000]  
partial_x_train = x_train[10000:]
```



```
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

You'll now train the model for 20 epochs (20 iterations over all samples in the `x_train` and `y_train` tensors), in mini-batches of 512 samples. At the same time, you'll monitor loss and accuracy on the 10,000 samples that you set apart. You do so by passing the validation data as the `validation_data` argument.

#### Listing 3.8. Training your model

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit(partial_x_train,
                   partial_y_train,
                   epochs=20,
                   batch_size=512,
                   validation_data=(x_val, y_val))
```

On CPU, this will take less than 2 seconds per epoch—training is over in 20 seconds. At the end of every epoch, there is a slight pause as the model computes its loss and accuracy on the 10,000 samples of the validation data.

Note that the call to `model.fit()` returns a `History` object. This object has a member `history`, which is a dictionary containing data about everything that happened during training. Let's look at it:

```
>>> history_dict = history.history
>>> history_dict.keys()
[u'acc', u'loss', u'val_acc', u'val_loss']
```

The dictionary contains four entries: one per metric that was being monitored during training and during validation. In the following two listing, let's use Matplotlib to plot the training and validation loss side by side (see [figure 3.7](#)), as well as the training and validation accuracy (see [figure 3.8](#)). Note that your own results may vary slightly due to a different random initialization of your network.

### Listing 3.9 Plotting the training and validation loss

```
import matplotlib.pyplot as plt

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

epochs = range(1, len(loss_values) + 1)

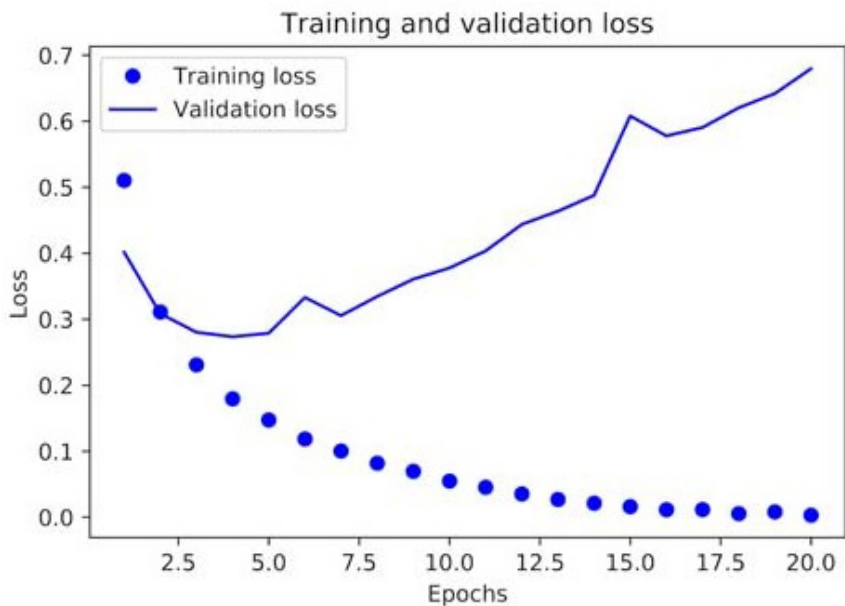
plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

❶ “bo” is for “blue dot.”

❷ “b” is for “solid blue line.”

Figure 3.7. Training and validation loss



### Listing 3.10. Plotting the training and validation accuracy

```
plt.clf()
acc = history_dict['acc']
```

❶

```

val_acc = history_dict['val_acc']

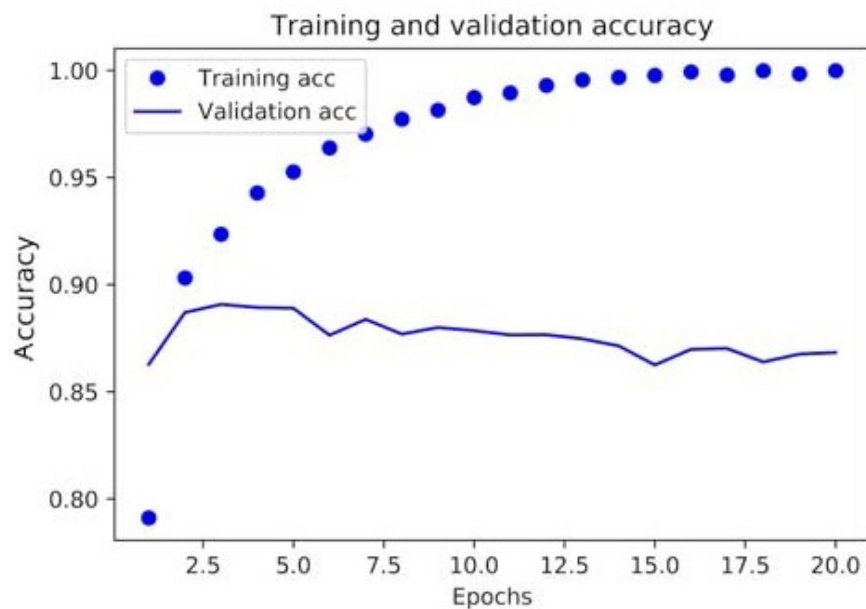
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()

```

❶ Clears the figure

Figure 3.8. Training and validation accuracy



Listing 3.9. Plotting the training and validation loss

As you can see, the training loss decreases with every epoch, and the training accuracy increases with every epoch. That’s what you would expect when running gradient-descent optimization—the quantity you’re trying to minimize should be less with every iteration. But that isn’t the case for the validation loss and accuracy: they seem to peak at the fourth epoch. This is an example of what we warned against earlier: a model that performs better on the training data isn’t necessarily a model that will do better on data it has never seen before. In precise terms, what you’re seeing is *overfitting*: after the second epoch, you’re overoptimizing on the training data, and you end up learning representations that are specific to the training data and don’t generalize to data outside of the training set.

In this case, to prevent overfitting, you could stop training after three epochs. In general, you can use a range of techniques to mitigate overfitting, which we'll cover in [chapter 4](#).

Let's train a new network from scratch for four epochs and then evaluate it on the test data.

Listing 3.11. Retraining a model from scratch

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

The final results are as follows:

```
>>> results
[0.2929924130630493, 0.8832799999999995]
```

This fairly naive approach achieves an accuracy of 88%. With state-of-the-art approaches, you should be able to get close to 95%.

### 3.4.5. Using a trained network to generate predictions on new data

After having trained a network, you'll want to use it in a practical setting. You can generate the likelihood of reviews being positive by using the `predict` method:

```
>>> model.predict(x_test)
array([[ 0.98006207]
       [ 0.99758697]
       [ 0.99975556]
       ...,
       [ 0.82167041])
```

```
[ 0.02885115]  
[ 0.65371346]], dtype=float32)
```

As you can see, the network is confident for some samples (0.99 or more, or 0.01 or less) but less confident for others (0.6, 0.4).

### 3.4.6. Further experiments

The following experiments will help convince you that the architecture choices you've made are all fairly reasonable, although there's still room for improvement:

- You used two hidden layers. Try using one or three hidden layers, and see how doing so affects validation and test accuracy.
- Try using layers with more hidden units or fewer hidden units: 32 units, 64 units, and so on.
- Try using the `mse` loss function instead of `binary_crossentropy`.
- Try using the `tanh` activation (an activation that was popular in the early days of neural networks) instead of `relu`.

### 3.4.7. Wrapping up

Here's what you should take away from this example:

- You usually need to do quite a bit of preprocessing on your raw data in order to be able to feed it—as tensors—into a neural network. Sequences of words can be encoded as binary vectors, but there are other encoding options, too.
- Stacks of Dense layers with `relu` activations can solve a wide range of problems (including sentiment classification), and you'll likely use them frequently.
- In a binary classification problem (two output classes), your network should end with a Dense layer with one unit and a `sigmoid` activation: the output of your network should be a scalar between 0 and 1, encoding a probability.
- With such a scalar sigmoid output on a binary classification problem, the loss function you should use is `binary_crossentropy`.
- The `rmsprop` optimizer is generally a good enough choice, whatever your problem. That's one less thing for you to worry about.