In [1]:
```
# Import libraries
from tensorflow import keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import RMSprop
```

In [2]:
```
import keras
keras.__version__
```

Out[2]:
```
'2.4.3'
```

In [5]:
```
# Load the dataset
from keras.datasets import imdb
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
s/imdb.npz
17465344/17464789 [==============================] - 1s 0us/step
<__array_function__ internals>:5: VisibleDeprecationWarning: Creating an ndarray
from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or nda
rrays with different lengths or shapes) is deprecated. If you meant to do this,
you must specify 'dtype=object' when creating the ndarray
/opt/conda/lib/python3.8/site-packages/tensorflow/python/keras/datasets/imdb.py:
159: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths
or shapes) is deprecated. If you meant to do this, you must specify 'dtype=objec
t' when creating the ndarray
  x_train, y_train = np.array(xs[:idx]), np.array(labels[:idx])
/opt/conda/lib/python3.8/site-packages/tensorflow/python/keras/datasets/imdb.py:
160: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths
or shapes) is deprecated. If you meant to do this, you must specify 'dtype=objec
t' when creating the ndarray
  x_test, y_test = np.array(xs[idx:]), np.array(labels[idx:])
```

The argument num_words=10000 means that we will only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows us to work with vector data of manageable size. The variables train_data and test_data are lists of reviews, each review being 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”:

In [6]:
```
train_data[0]
```

Out[6]:
```
[1,
 14,
 22,
 16,
 43,
 530,
 973,
 1622,
 1385,
 65,
458,
4468,
66,
3941,
4,
173,
36,
256,
5,
25,
100,
43,
838,
112,
50,
670,
2,
9,
35,
480,
284,
5,
150,
4,
172,
112,
167,
2,
336,
385,
39,
4,
172,
4536,
1111,
17,
546,
38,
13,
447,
4,
192,
50,
16,
6,
147,
2025,
19,
14,
22,
4,
1920,
4613,
469,
4,
22,
71,
87,
12,
16,
43,
530,
38,
76,
15,
13,
1247,
4,
22,
17,
515,
17,
12,
16,
626,
18,
2,
5,
62,
386,
12,
8,
316,
8,
106,
5,
4,
2223,
5244,
16,
480,
66,
3785,
33,
4,
130,
12,
16,
38,
619,
5,
25,
124,
51,
36,
135,
48,
25,
1415,
33,
6,
22,
12,
215,
28,
77,
52,
5,
14,
407,
16,
82,
2,
8,
4,
107,
117,
5952,
15,
256,
4,
2,
7,
3766,
5,
723,
36,
71,
43,
530,
476,
26,
400,
317,
46,
7,
4,
2,
1029,
13,
104,
88,
4,
381,
15,
297,
98,
32,
2071,
56,
26,
141,
6,
194,
7486,
18,
4,
226,
22,
21,
134,
476,
26,
480,
5,
144,
30,
5535,
18,
51,
36,
28,
224,
92,
25,
104,
4,
226,
65,
16,
38,
1334,
88,
12,
16,
283,
5,
16,
4472,
113,
103,
32,
15,
16,
5345,
19,
178,
32]
```

In [7]:
```
train_labels[0]
```

Out[7]:
```
1
```

In [8]:
```
#Since we restricted ourselves to the top 10,000 most frequent words, no word in
max([max(sequence) for sequence in train_data])
```

Out[8]:
```
9999
```

Lastly, we need to pick a loss function and an optimizer. Since we are facing a binary classification problem and the output of our network is a probability (we end our network with a single-unit layer with a sigmoid activation), is it 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 are 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 our case, between the ground-truth distribution and our predictions. Here's the step where we configure our model with the rmsprop optimizer and the binary_crossentropy loss function. Note that we will also monitor accuracy during training.

In [10]:
```
# word_index is a dictionary mapping words to an integer index
word_index = imdb.get_word_index()
# We reverse it, mapping integer indices to words
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
# We decode the review; note that our indices were offset by 3
# because 0, 1 and 2 are reserved indices for "padding", "start of sequence", an
decoded_review = ' '.join([reverse_word_index.get(i - 3, '?') for i in train_dat
decoded_review
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
s/imdb_word_index.json
1646592/1641221 [==============================] - 0s 0us/step
```

Out[10]:
```
"? this film was just brilliant casting location scenery story direction everyon
e's really suited the part they played and you could just imagine being there ro
bert ? is an amazing actor and now the same being director ? father came from th
e same scottish island as myself so i loved the fact there was a real connection
with this film the witty remarks throughout the film were great it was just bril
liant so much that i bought the film as soon as it was released for ? and would
recommend it to everyone to watch and the fly fishing was amazing really cried a
t the end it was so sad and you know what they say if you cry at a film it must
have been good and this definitely was also ? to the two little boy's that playe
d the ? of norman and paul they were just brilliant children are often left out
```

```
of the ? list i think because the stars that play them all grown up are such a b
ig profile for the whole film but these children are amazing and should be prais
ed for what they have done don't you think the whole story was so lovely because
it was true and was someone's life after all that was shared with us all"
```

Preparing the data We cannot feed lists of integers into a neural network. We have to turn our lists into tensors. There are two ways we could do that: We could pad our lists so that they all have the same length, and turn them into an integer tensor of shape (samples, word_indices), then use as first layer in our network a layer capable of handling such integer tensors (the Embedding layer, which we will cover in detail later in the book). We could one-hot-encode our lists to turn them into vectors of 0s and 1s. Concretely, this would mean for instance turning the sequence [3, 5] into a 10,000-dimensional vector that would be all-zeros except for indices 3 and 5, which would be ones. Then we could use as first layer in our network a Dense layer, capable of handling floating point vector data. We will go with the latter solution. Let's vectorize our data, which we will do manually for maximum clarity:

In [12]:
```python
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
# Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1. # set specific indices of results[i] to 1s
    return results
# Our vectorized training data
x_train = vectorize_sequences(train_data)
# Our vectorized test data
x_test = vectorize_sequences(test_data)
```

In [13]:
```python
# Print sample
x_train[0]
```

Out[13]:
```
array([0., 1., 1., ..., 0., 0., 0.])
```

In [14]:
```python
# Our vectorized labels
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

Now our data is ready to be fed into a neural network. ##### Building our network The input data is simply vectors, and the labels are scalars (1s and 0s): this is the easiest setup you will ever encounter. A type of network that performs well on such a problem would be a simple stack of fullyconnected (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. What's a hidden unit? It's a dimension in the representation space of the layer. You may remember from the previous chapter 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 that the weight matrix W will have shape (input_dimension, 16), i.e. the dot product with W will project the input data onto a 16-dimensional representation space (and then we would 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 are allowing the network to have when learning internal representations". Having more hidden units (a higherdimensional representation space) allows your network to learn more complex representations, but it makes your 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 stack of dense layers: How many layers to use. How many "hidden units" to chose for each layer.

In the next chapter, you will learn formal principles to guide you in making these choices. For the time being, you will have to trust us with the following architecture choice: two intermediate layers with 16 hidden units each, and a third layer which 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", i.e. how likely the review is to be positive). A relu (rectified linear unit) is a function meant to zero-out negative values, while a sigmoid "squashes" arbitrary values into the [0, 1] interval, thus outputting something that can be interpreted as a probability.

In [15]:

```
 #The Keras implementation
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'))
```

Lastly, we need to pick a loss function and an optimizer. Since we are facing a binary classification problem and the output of our network is a probability (we end our network with a single-unit layer with a sigmoid activation), is it 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 are 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 our case, between the ground-truth distribution and our predictions. Here's the step where we configure our model with the rmsprop optimizer and the binary_crossentropy loss function. Note that we will also monitor accuracy during training.

In [16]:

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

We are passing our 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. This former can be done by passing an optimizer class instance as the optimizer argument:

In [17]:

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

Validating our approach In order to monitor during training the accuracy of the model on data that it has never seen before, we will create a “validation set” by setting apart 10,000 samples from the original training data:

In [18]:

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

We will now train our model for 20 epochs (20 iterations over all samples in the x_train and y_train tensors), in mini-batches of 512 samples. At this same time we will monitor loss and accuracy on the 10,000 samples that we set apart. This is done by passing the validation data as the validation_data argument:

In [19]:

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

```
Epoch 1/20
30/30 [==============================] - 3s 61ms/step - loss: 0.5896 - binary_ac
curacy: 0.6966 - val_loss: 0.3877 - val_binary_accuracy: 0.8690
Epoch 2/20
30/30 [==============================] - 0s 13ms/step - loss: 0.3270 - binary_ac
curacy: 0.8994 - val_loss: 0.3082 - val_binary_accuracy: 0.8879
Epoch 3/20
30/30 [==============================] - 0s 12ms/step - loss: 0.2300 - binary_ac
curacy: 0.9317 - val_loss: 0.2924 - val_binary_accuracy: 0.8826
Epoch 4/20
30/30 [==============================] - 0s 14ms/step - loss: 0.1816 - binary_ac
curacy: 0.9443 - val_loss: 0.2919 - val_binary_accuracy: 0.8820
Epoch 5/20
30/30 [==============================] - 0s 12ms/step - loss: 0.1424 - binary_ac
curacy: 0.9544 - val_loss: 0.2918 - val_binary_accuracy: 0.8826
Epoch 6/20
30/30 [==============================] - 0s 13ms/step - loss: 0.1152 - binary_ac
curacy: 0.9659 - val_loss: 0.2932 - val_binary_accuracy: 0.8837
Epoch 7/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0935 - binary_ac
curacy: 0.9745 - val_loss: 0.3382 - val_binary_accuracy: 0.8744
Epoch 8/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0819 - binary_ac
curacy: 0.9788 - val_loss: 0.3386 - val_binary_accuracy: 0.8778
Epoch 9/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0647 - binary_ac
curacy: 0.9834 - val_loss: 0.3585 - val_binary_accuracy: 0.8784
Epoch 10/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0500 - binary_ac
```

```
curacy: 0.9894 - val_loss: 0.3940 - val_binary_accuracy: 0.8716
Epoch 11/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0412 - binary_ac
curacy: 0.9924 - val_loss: 0.4084 - val_binary_accuracy: 0.8719
Epoch 12/20
30/30 [==============================] - 0s 13ms/step - loss: 0.0300 - binary_ac
curacy: 0.9952 - val_loss: 0.4425 - val_binary_accuracy: 0.8705
Epoch 13/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0229 - binary_ac
curacy: 0.9963 - val_loss: 0.4636 - val_binary_accuracy: 0.8720
Epoch 14/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0162 - binary_ac
curacy: 0.9988 - val_loss: 0.5063 - val_binary_accuracy: 0.8678
Epoch 15/20
30/30 [==============================] - 0s 15ms/step - loss: 0.0143 - binary_ac
curacy: 0.9984 - val_loss: 0.5355 - val_binary_accuracy: 0.8679
Epoch 16/20
30/30 [==============================] - 1s 19ms/step - loss: 0.0093 - binary_ac
curacy: 0.9996 - val_loss: 0.5828 - val_binary_accuracy: 0.8639
Epoch 17/20
30/30 [==============================] - 0s 15ms/step - loss: 0.0076 - binary_ac
curacy: 0.9996 - val_loss: 0.6096 - val_binary_accuracy: 0.8678
Epoch 18/20
30/30 [==============================] - 0s 13ms/step - loss: 0.0055 - binary_ac
curacy: 0.9996 - val_loss: 0.6483 - val_binary_accuracy: 0.8658
Epoch 19/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0045 - binary_ac
curacy: 0.9995 - val_loss: 0.6728 - val_binary_accuracy: 0.8656
Epoch 20/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0025 - binary_ac
curacy: 0.9998 - val_loss: 0.7014 - val_binary_accuracy: 0.8658
```

On CPU, this will take less than two 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 take a look at it:

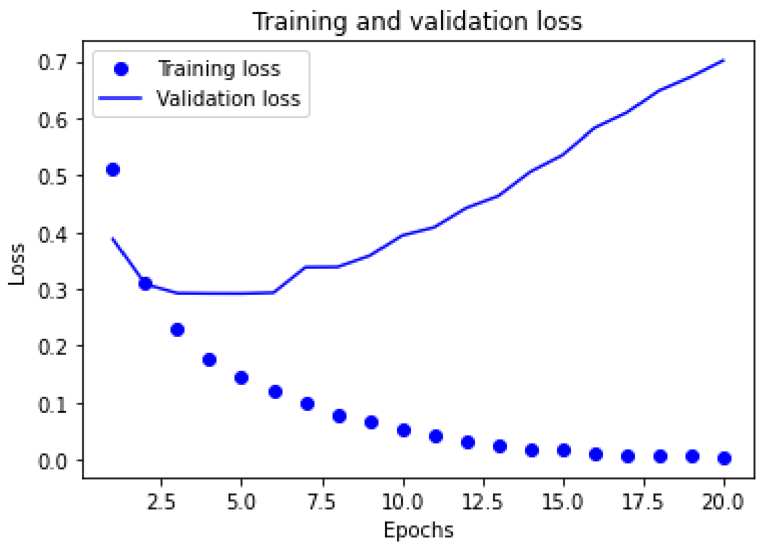
```
In [20]:
history_dict = history.history
history_dict.keys()
#dict_keys(['loss', 'val_loss', 'binary_accuracy', 'val_binary_accuracy'])
#history_dict = history.history
print(history_dict)
```

```
{'loss': [0.5123856067657471, 0.3122806251049042, 0.2284364402294159, 0.17730687
55865097, 0.1445995867252349 9, 0.1197182387113571 2, 0.0974646732211113, 0.078708
4773182869, 0.0665047615766525 3, 0.0519377142190933 2, 0.0421204194426536 56, 0.03
287658840417862, 0.0258494392037391 66, 0.0172231532633304 6, 0.0182574335485696 8,
0.0095374844968318 94, 0.0079089142382144 93, 0.0060765622183680 534, 0.00571954296
9018221, 0.0042051179334521 29], 'binary_accuracy': [0.7807999849319458, 0.897800
0283241272, 0.9264666438102722, 0.9437333345413208, 0.9519333243370056, 0.962999
9995231628, 0.9708666801452637, 0.9793333411216736, 0.9819999933242798, 0.987733
3045005798, 0.9911999702453613, 0.9932000041007996, 0.9946666955947876, 0.998266
6373252869, 0.996999979019165, 0.9994666576385498, 0.9994666576385498, 0.9995333
552360535, 0.9992666840553284, 0.9994666576385498], 'val_loss': [0.3876972794532
776, 0.3082281649112701 4, 0.2923900783061981, 0.2918720245361328, 0.29175585083
4656, 0.2932302057743072 5, 0.338207930326461 8, 0.3386070430278778, 0.35846075415
611267, 0.3940198719501495 4, 0.4084009528160095, 0.4425483047962188 7, 0.46361055
970191956, 0.5063155889511108, 0.5354788303375244, 0.5827764868736267, 0.6095966
100692749, 0.6483209729194641, 0.6727501749992371, 0.7013818621635437], 'val_bin
ary_accuracy': [0.8690000176429749, 0.8878999948501587, 0.8826000094413757, 0.88
20000290870667, 0.8826000094413757, 0.8837000131607056, 0.8744000196456909, 0.87
77999877929688, 0.8784000277519226, 0.8715999722480774, 0.8719000220298767, 0.87
05000281333923, 0.871999979019165, 0.8677999973297119, 0.867900013923645, 0.8639
000058174133, 0.8677999973297119, 0.8658000230789185, 0.8655999898910522, 0.8658
000230789185]}
```

It contains 4 entries: one per metric that was being monitored, during training and during validation. Let's use Matplotlib to plot the training and validation loss side by side, as well as the training and validation accuracy:

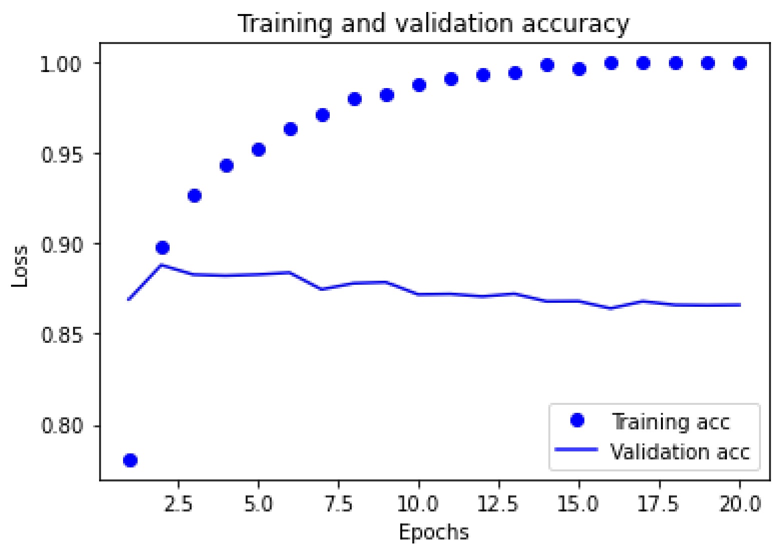
In [21]:

```python
import matplotlib.pyplot as plt
acc = history.history['binary_accuracy']
val_acc = history.history['val_binary_accuracy']
#acc = history.history['acc']
#val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
# "bo" is for "blue dot"
plt.plot(epochs, loss, 'bo', label='Training loss')
# b is for "solid blue line"
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



In [22]:

```python
plt.clf() # clear figure
#acc_values = history_dict['acc']
#val_acc_values = history_dict['val_acc']
acc = history.history['binary_accuracy']
val_acc = history.history['val_binary_accuracy']
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('Loss')
plt.legend()
plt.show()
```



The dots are the training loss and accuracy, while the solid lines are the validation loss and accuracy. Note that your own results may vary slightly due to a different random initialization of your network. 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 are trying to minimize should get lower 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 were warning 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 are seeing is "overfitting": after the second epoch, we are over-optimizing on the training data, and we ended up learning representations that are specific to the training data and do not generalize to data outside of the training set. In this case, to prevent overfitting, we could simply stop training after three epochs. In general, there is a range of techniques you can leverage to mitigate overfitting, which we will cover in the next chapter. Let's train a new network from scratch for four epochs, then evaluate it on our test data:

In [23]:

```
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)
```

```
Epoch 1/4
49/49 [==============================] - 1s 11ms/step - loss: 0.5418 - accuracy:
0.7406
Epoch 2/4
49/49 [==============================] - 0s 9ms/step - loss: 0.2661 - accuracy:
0.9120
Epoch 3/4
49/49 [==============================] - 0s 9ms/step - loss: 0.2003 - accuracy:
0.9277
Epoch 4/4
49/49 [==============================] - 0s 8ms/step - loss: 0.1591 - accuracy:
```

```
0.9446
782/782 [==============================] - 2s 2ms/step - loss: 0.3110 - accurac
y: 0.8787
```

In [24]:

```
results
```

Out[24]:

```
[0.3109950125217438, 0.8786799907684326]
```

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

Using a trained network to generate predictions on new data After having trained a network, you will want to use it in a practical setting. You can generate the likelihood of reviews being positive by using the predict method:

In [25]:

```
model.predict(x_test)
```

Out[25]:

```
array([[0.18303666],
       [0.9999194 ],
       [0.971292  ],
       ...,
       [0.15059358],
       [0.11548319],
       [0.78961325]], dtype=float32)
```

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

Conclusion There’s usually quite a bit of preprocessing you need to do on your raw data in order to be able to feed it – as tensors – into a neural network. In the case of sequences of words, they 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 will likely use them frequently. In a binary classification problem (two output classes), your network should end with a Dense layer with 1 unit and a sigmoid activation, i.e. 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 of optimizer, whatever your problem.

In [ ]:

```

```