In [1]:

```
import tensorflow as tf
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)


import numpy as np
load = np.load
np.load = lambda *a, **k: load(*a, **dict(k, allow_pickle=True))
```

In [2]:

```
import keras
keras.__version__
```

```
Using TensorFlow backend.
```

Out[2]:

```
'2.2.4'
```

Classifying newswires: a multi-class classification example

In the previous section we saw how to classify vector inputs into two mutually exclusive classes using a densely-connected neural network. But what happens when you have more than two classes?

In this section, we will build a network to classify Reuters newswires into 46 different mutually-exclusive topics. Since we have many classes, this problem is an instance of "multi-class classification", and since each data point should be classified into only one category, the problem is more specifically an instance of "single-label, multi-class classification". If each data point could have belonged to multiple categories (in our case, topics) then we would be facing a "multi-label, multi-class classification" problem.

The Reuters dataset

We will be working with the *Reuters dataset*, a set of short newswires and their topics, published by Reuters in 1986. It's a very simple, widely used toy dataset for text classification. There are 46 different topics; some topics are more represented than others, but each topic has at least 10 examples in the training set.

Like IMDB and MNIST, the Reuters dataset comes packaged as part of Keras. Let's take a look right away:

In [3]:

```
from keras.datasets import reuters

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

Like with the IMDB dataset, the argument `num_words=10000` restricts the data to the 10,000 most frequently occurring words found in the data.

We have 8,982 training examples and 2,246 test examples:

In [4]:

```
len(train_data)
```

Out[4]:

```
8982
```

In [5]:

```
len(test_data)
```

Out[5]:

```
2246
```

As with the IMDB reviews, each example is a list of integers (word indices):

In [7]:

```
train_data[0][:10]
```

Out[7]:

```
[1, 2, 2, 8, 43, 10, 447, 5, 25, 207]
```

Here's how you can decode it back to words, in case you are curious:

In [8]:

```
word_index = reuters.get_word_index()
reverse_word_index = {value:  key for key, value in word_index.items()}
# Note that our indices were offset by 3
# because 0, 1 and 2 are reserved indices for "padding", "start of sequence", an
d "unknown".
decoded_newswire = ' '.join([reverse_word_index.get(i - 3, '?') for i in train_d
ata[0]])
```

```
Downloading data from https://s3.amazonaws.com/text-datasets/reuters
_word_index.json
557056/550378 [==============================] - 1s 1us/step
```

In [9]:

```
decoded_newswire[:100]
```

Out[9]:

```
'? ? ? said as a result of its december acquisition of space co it e
xpects earnings per share in 1987'
```

The label associated with an example is an integer between 0 and 45: a topic index.

In [10]:

```
train_labels[10]
```

Out[10]:

```
3
```

Preparing the data

We can vectorize the data with the exact same code as in our previous example:

In [11]:

```
import numpy as np

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

# Our vectorized training data
x_train = vectorize_sequences(train_data)
# Our vectorized test data
x_test = vectorize_sequences(test_data)
```

To vectorize the labels, there are two possibilities: we could just cast the label list as an integer tensor, or we could use a "one-hot" encoding. One-hot encoding is a widely used format for categorical data, also called "categorical encoding". For a more detailed explanation of one-hot encoding, you can refer to Chapter 6, Section 1. In our case, one-hot encoding of our labels consists in embedding each label as an all-zero vector with a 1 in the place of the label index, e.g.:

In [17]:

```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results

# Our vectorized training labels
one_hot_train_labels = to_one_hot(train_labels)
# Our vectorized test labels
one_hot_test_labels = to_one_hot(test_labels)
```

Note that there is a built-in way to do this in Keras, which you have already seen in action in our MNIST example:

In [18]:

```
from keras.utils.np_utils import to_categorical

one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

Building our network

This topic classification problem looks very similar to our previous movie review classification problem: in both cases, we are trying to classify short snippets of text. There is however a new constraint here: the number of output classes has gone from 2 to 46, i.e. the dimensionality of the output space is much larger.

In a stack of `Dense` layers like what we were using, each layer can only access information present in the output of the previous layer. If one layer drops some information relevant to the classification problem, this information can never be recovered by later layers: each layer can potentially become an "information bottleneck". In our previous example, we were using 16-dimensional intermediate layers, but a 16-dimensional space may be too limited to learn to separate 46 different classes: such small layers may act as information bottlenecks, permanently dropping relevant information.

For this reason we will use larger layers. Let's go with 64 units:

In [19]:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

There are two other things you should note about this architecture:

- We are ending the network with a `Dense` layer of size 46. This means that for each input sample, our network will output a 46-dimensional vector. Each entry in this vector (each dimension) will encode a different output class.
- The last layer uses a `softmax` activation. You have already seen this pattern in the MNIST example. It means that the network will output a *probability distribution* over the 46 different output classes, i.e. for every input sample, the network will produce a 46-dimensional output vector where `output[i]` is the probability that the sample belongs to class `i`. The 46 scores will sum to 1.

The best loss function to use in this case is `categorical_crossentropy`. It measures the distance between two probability distributions: in our case, between the probability distribution output by our network, and the true distribution of the labels. By minimizing the distance between these two distributions, we train our network to output something as close as possible to the true labels.

In [20]:

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

Validating our approach

Let's set apart 1,000 samples in our training data to use as a validation set:

In [21]:

```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]

y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]
```

Now let's train our network for 20 epochs:

In [22]:

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

```
Train on 7982 samples, validate on 1000 samples
Epoch 1/20
7982/7982 [==============================] - 2s 206us/step - loss:
2.5270 - acc: 0.4981 - val_loss: 1.7191 - val_acc: 0.6100
Epoch 2/20
7982/7982 [==============================] - 1s 129us/step - loss:
1.4440 - acc: 0.6887 - val_loss: 1.3484 - val_acc: 0.7100
Epoch 3/20
7982/7982 [==============================] - 1s 131us/step - loss:
1.0962 - acc: 0.7656 - val_loss: 1.1725 - val_acc: 0.7430
Epoch 4/20
7982/7982 [==============================] - 1s 128us/step - loss:
0.8715 - acc: 0.8152 - val_loss: 1.0853 - val_acc: 0.7580
Epoch 5/20
7982/7982 [==============================] - 1s 128us/step - loss:
0.7048 - acc: 0.8492 - val_loss: 0.9856 - val_acc: 0.7800
Epoch 6/20
7982/7982 [==============================] - 1s 128us/step - loss:
0.5677 - acc: 0.8787 - val_loss: 0.9421 - val_acc: 0.8030
Epoch 7/20
7982/7982 [==============================] - 1s 130us/step - loss:
0.4596 - acc: 0.9039 - val_loss: 0.9101 - val_acc: 0.8010
Epoch 8/20
7982/7982 [==============================] - 1s 128us/step - loss:
0.3704 - acc: 0.9228 - val_loss: 0.9341 - val_acc: 0.7930
Epoch 9/20
7982/7982 [==============================] - 1s 130us/step - loss:
0.3037 - acc: 0.9316 - val_loss: 0.8904 - val_acc: 0.8050
Epoch 10/20
7982/7982 [==============================] - 1s 129us/step - loss:
0.2541 - acc: 0.9412 - val_loss: 0.9059 - val_acc: 0.8130
Epoch 11/20
7982/7982 [==============================] - 1s 129us/step - loss:
0.2185 - acc: 0.9474 - val_loss: 0.9167 - val_acc: 0.8100
Epoch 12/20
7982/7982 [==============================] - 1s 129us/step - loss:
0.1874 - acc: 0.9510 - val_loss: 0.9055 - val_acc: 0.8150
Epoch 13/20
7982/7982 [==============================] - 1s 129us/step - loss:
0.1698 - acc: 0.9520 - val_loss: 0.9322 - val_acc: 0.8090
Epoch 14/20
7982/7982 [==============================] - 1s 131us/step - loss:
0.1530 - acc: 0.9559 - val_loss: 0.9637 - val_acc: 0.8050
Epoch 15/20
7982/7982 [==============================] - 1s 129us/step - loss:
0.1387 - acc: 0.9557 - val_loss: 0.9686 - val_acc: 0.8120
Epoch 16/20
7982/7982 [==============================] - 1s 128us/step - loss:
0.1311 - acc: 0.9557 - val_loss: 1.0276 - val_acc: 0.8010
Epoch 17/20
7982/7982 [==============================] - 1s 130us/step - loss:
0.1216 - acc: 0.9577 - val_loss: 1.0274 - val_acc: 0.8000
Epoch 18/20
7982/7982 [==============================] - 1s 129us/step - loss:
0.1191 - acc: 0.9573 - val_loss: 1.0431 - val_acc: 0.8050
Epoch 19/20
7982/7982 [==============================] - 1s 129us/step - loss:
0.1135 - acc: 0.9597 - val_loss: 1.0947 - val_acc: 0.7970
Epoch 20/20
7982/7982 [==============================] - 1s 130us/step - loss:
0.1108 - acc: 0.9598 - val_loss: 1.0666 - val_acc: 0.7990
```

Let's display its loss and accuracy curves:

In [24]:

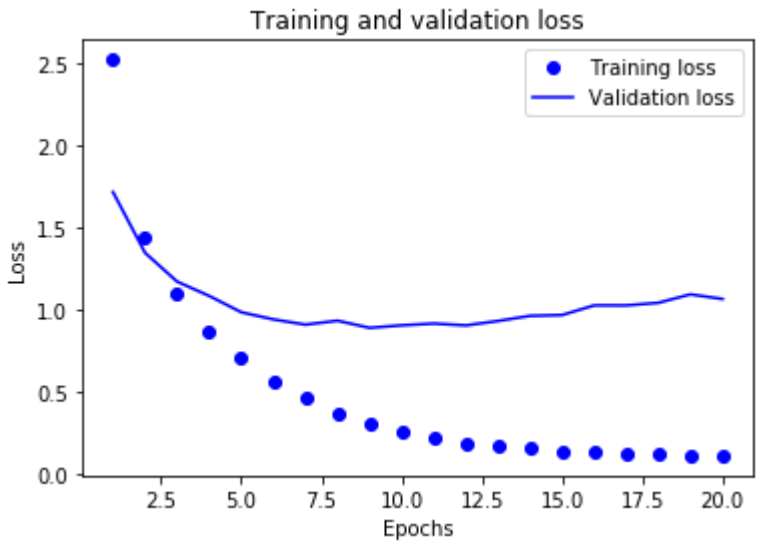
```
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['val_loss']

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

plt.plot(epochs, loss, 'bo', label='Training loss')
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();
```



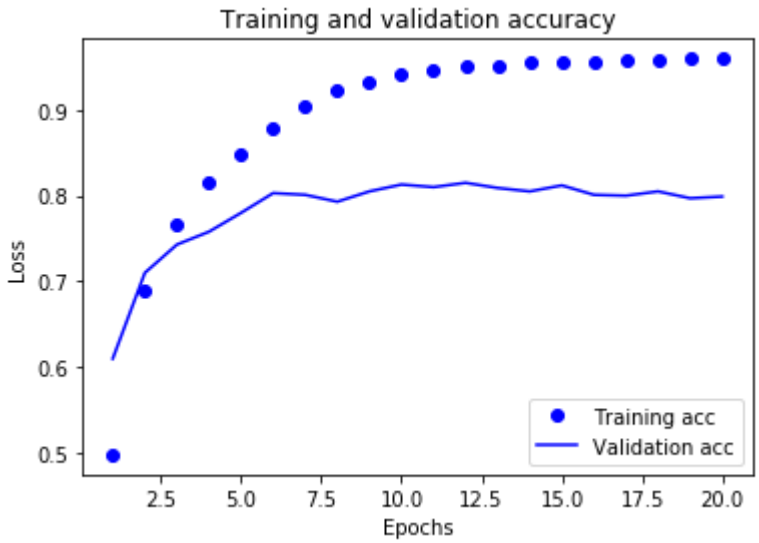
In [26]:

```
plt.clf()   # clear figure

acc = history.history['acc']
val_acc = history.history['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('Loss')
plt.legend()

plt.show();
```



It seems that the network starts overfitting after 8 epochs. Let's train a new network from scratch for 8 epochs, then let's evaluate it on the test set:

In [27]:

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=8,
          batch_size=512,
          validation_data=(x_val, y_val))
results = model.evaluate(x_test, one_hot_test_labels)
```

```
Train on 7982 samples, validate on 1000 samples
Epoch 1/8
7982/7982 [==============================] - 1s 182us/step - loss:
2.5399 - acc: 0.5226 - val_loss: 1.6795 - val_acc: 0.6530
Epoch 2/8
7982/7982 [==============================] - 1s 129us/step - loss:
1.3768 - acc: 0.7101 - val_loss: 1.2801 - val_acc: 0.7250
Epoch 3/8
7982/7982 [==============================] - 1s 129us/step - loss:
1.0206 - acc: 0.7783 - val_loss: 1.1355 - val_acc: 0.7490
Epoch 4/8
7982/7982 [==============================] - 1s 131us/step - loss:
0.8046 - acc: 0.8245 - val_loss: 1.0567 - val_acc: 0.7600
Epoch 5/8
7982/7982 [==============================] - 1s 130us/step - loss:
0.6453 - acc: 0.8619 - val_loss: 0.9776 - val_acc: 0.7960
Epoch 6/8
7982/7982 [==============================] - 1s 131us/step - loss:
0.5165 - acc: 0.8911 - val_loss: 0.9131 - val_acc: 0.8130
Epoch 7/8
7982/7982 [==============================] - 1s 130us/step - loss:
0.4156 - acc: 0.9143 - val_loss: 0.8970 - val_acc: 0.8240
Epoch 8/8
7982/7982 [==============================] - 1s 129us/step - loss:
0.3379 - acc: 0.9281 - val_loss: 0.8765 - val_acc: 0.8250
2246/2246 [==============================] - 0s 182us/step
```

In [28]:

```
results
```

Out[28]:

```
[0.9877414837121327, 0.7849510240427426]
```

Our approach reaches an accuracy of ~78%. With a balanced binary classification problem, the accuracy reached by a purely random classifier would be 50%, but in our case it is closer to 19%, so our results seem pretty good, at least when compared to a random baseline:

In [29]:

```
import copy

test_labels_copy = copy.copy(test_labels)
np.random.shuffle(test_labels_copy)

float(np.sum(np.array(test_labels) == np.array(test_labels_copy))) / len(test_la
bels)
```

Out[29]:

```
0.19679430097951914
```

Generating predictions on new data

We can verify that the `predict` method of our model instance returns a probability distribution over all 46 topics. Let's generate topic predictions for all of the test data:

In [30]:

```
predictions = model.predict(x_test)
```

Each entry in `predictions` is a vector of length 46:

In [31]:

```
predictions[0].shape
```

Out[31]:

```
(46,)
```

The coefficients in this vector sum to 1:

In [32]:

```
np.sum(predictions[0])
```

Out[32]:

```
1.0
```

The largest entry is the predicted class, i.e. the class with the highest probability:

In [33]:

```
np.argmax(predictions[0])
```

Out[33]:

```
3
```

A different way to handle the labels and the loss

We mentioned earlier that another way to encode the labels would be to cast them as an integer tensor, like such:

In [34]:

```
y_train = np.array(train_labels)
y_test = np.array(test_labels)
```

The only thing it would change is the choice of the loss function. Our previous loss, `categorical_crossentropy`, expects the labels to follow a categorical encoding. With integer labels, we should use `sparse_categorical_crossentropy`:

In [35]:

```
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metri
cs=['acc'])
```

This new loss function is still mathematically the same as `categorical_crossentropy`; it just has a different interface.

On the importance of having sufficiently large intermediate layers

We mentioned earlier that since our final outputs were 46-dimensional, we should avoid intermediate layers with much less than 46 hidden units. Now let's try to see what happens when we introduce an information bottleneck by having intermediate layers significantly less than 46-dimensional, e.g. 4-dimensional.

In [36]:

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=20,
          batch_size=128,
          validation_data=(x_val, y_val))
```

```
Train on 7982 samples, validate on 1000 samples
Epoch 1/20
7982/7982 [==============================] - 2s 240us/step - loss:
2.7073 - acc: 0.4409 - val_loss: 2.0153 - val_acc: 0.5930
Epoch 2/20
7982/7982 [==============================] - 1s 177us/step - loss:
1.7341 - acc: 0.6185 - val_loss: 1.6574 - val_acc: 0.6090
Epoch 3/20
7982/7982 [==============================] - 1s 174us/step - loss:
1.4679 - acc: 0.6358 - val_loss: 1.5689 - val_acc: 0.6080
Epoch 4/20
7982/7982 [==============================] - 1s 172us/step - loss:
1.3275 - acc: 0.6438 - val_loss: 1.4832 - val_acc: 0.6170
Epoch 5/20
7982/7982 [==============================] - 1s 179us/step - loss:
1.2171 - acc: 0.6505 - val_loss: 1.4549 - val_acc: 0.6240
Epoch 6/20
7982/7982 [==============================] - 1s 174us/step - loss:
1.1267 - acc: 0.6745 - val_loss: 1.4371 - val_acc: 0.6310
Epoch 7/20
7982/7982 [==============================] - 1s 175us/step - loss:
1.0522 - acc: 0.6941 - val_loss: 1.4525 - val_acc: 0.6430
Epoch 8/20
7982/7982 [==============================] - 1s 170us/step - loss:
0.9924 - acc: 0.7043 - val_loss: 1.4385 - val_acc: 0.6490
Epoch 9/20
7982/7982 [==============================] - 1s 178us/step - loss:
0.9423 - acc: 0.7176 - val_loss: 1.4755 - val_acc: 0.6520
Epoch 10/20
7982/7982 [==============================] - 1s 173us/step - loss:
0.8983 - acc: 0.7428 - val_loss: 1.5170 - val_acc: 0.6520
Epoch 11/20
7982/7982 [==============================] - 1s 172us/step - loss:
0.8612 - acc: 0.7557 - val_loss: 1.5041 - val_acc: 0.6630
Epoch 12/20
7982/7982 [==============================] - 1s 174us/step - loss:
0.8263 - acc: 0.7613 - val_loss: 1.5175 - val_acc: 0.6640
Epoch 13/20
7982/7982 [==============================] - 1s 169us/step - loss:
0.7977 - acc: 0.7684 - val_loss: 1.5438 - val_acc: 0.6700
Epoch 14/20
7982/7982 [==============================] - 1s 173us/step - loss:
0.7711 - acc: 0.7734 - val_loss: 1.5962 - val_acc: 0.6650
Epoch 15/20
7982/7982 [==============================] - 1s 171us/step - loss:
0.7466 - acc: 0.7816 - val_loss: 1.6369 - val_acc: 0.6680
Epoch 16/20
7982/7982 [==============================] - 1s 185us/step - loss:
0.7238 - acc: 0.7893 - val_loss: 1.7250 - val_acc: 0.6610
Epoch 17/20
7982/7982 [==============================] - 1s 178us/step - loss:
0.7062 - acc: 0.7965 - val_loss: 1.7563 - val_acc: 0.6650
Epoch 18/20
7982/7982 [==============================] - 1s 174us/step - loss:
0.6865 - acc: 0.8038 - val_loss: 1.7737 - val_acc: 0.6710
Epoch 19/20
7982/7982 [==============================] - 1s 172us/step - loss:
0.6688 - acc: 0.8076 - val_loss: 1.7709 - val_acc: 0.6730
Epoch 20/20
7982/7982 [==============================] - 1s 172us/step - loss:
0.6551 - acc: 0.8147 - val_loss: 1.8200 - val_acc: 0.6720
```

Out[36]:

```
<keras.callbacks.History at 0x2af2bea8eb8>
```

Our network now seems to peak at c. 80% test accuracy, a 8% absolute drop. This drop is mostly due to the fact that we are now trying to compress a lot of information (enough information to recover the separation hyperplanes of 46 classes) into an intermediate space that is too low-dimensional. The network is able to cram *most* of the necessary information into these 8-dimensional representations, but not all of it.

Further experiments

- Try using larger or smaller layers: 32 units, 128 units...
- We were using two hidden layers. Now try to use a single hidden layer, or three hidden layers.

Wrapping up

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

- If you are trying to classify data points between N classes, your network should end with a `Dense` layer of size N.
- In a single-label, multi-class classification problem, your network should end with a `softmax` activation, so that it will output a probability distribution over the N output classes.
- *Categorical crossentropy* is almost always the loss function you should use for such problems. It minimizes the distance between the probability distributions output by the network, and the true distribution of the targets.
- There are two ways to handle labels in multi-class classification: **Encoding the labels via "categorical encoding" (also known as "one-hot encoding") and using `categorical_crossentropy` as your loss function.** Encoding the labels as integers and using the `sparse_categorical_crossentropy` loss function.
- If you need to classify data into a large number of categories, then you should avoid creating information bottlenecks in your network by having intermediate layers that are too small.