In [27]:

```
import warnings
warnings.warn = lambda *a, **k: ...
```

In [28]:

```
import keras
keras.__version__
```

Out[28]:

```
'2.2.4'
```

A first look at a neural network

We will now take a look at a first concrete example of a neural network, which makes use of the Python library Keras to learn to classify hand-written digits. Unless you already have experience with Keras or similar libraries, you will not understand everything about this first example right away. You probably haven't even installed Keras yet. Don't worry, that is perfectly fine. In the next chapter, we will review each element in our example and explain them in detail. So don't worry if some steps seem arbitrary or look like magic to you! We've got to start somewhere.

The problem we are trying to solve here is to classify grayscale images of handwritten digits (28 pixels by 28 pixels), into their 10 categories (0 to 9). The dataset we will use is the MNIST dataset, a classic dataset in the machine learning community, which has been around for almost as long as the field itself and has been very intensively studied. It's a set of 60,000 training images, plus 10,000 test images, assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s. You can think of "solving" MNIST as the "Hello World" of deep learning -- it's what you do to verify that your algorithms are working as expected. As you become a machine learning practitioner, you will see MNIST come up over and over again, in scientific papers, blog posts, and so on.

The MNIST dataset comes pre-loaded in Keras, in the form of a set of four Numpy arrays:

In [14]:

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

`train_images` and `train_labels` form the "training set", the data that the model will learn from. The model will then be tested on the "test set", `test_images` and `test_labels`. Our images are encoded as Numpy arrays, and the labels are simply an array of digits, ranging from 0 to 9. There is a one-to-one correspondence between the images and the labels.

Let's have a look at the training data:

In [15]:

```
train_images.shape
```

Out[15]:

```
(60000, 28, 28)
```

In [16]:

```
len(train_labels)
```

Out[16]:

```
60000
```

In [17]:

```
train_labels
```

Out[17]:

```
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

Let's have a look at the test data:

In [18]:

```
test_images.shape
```

Out[18]:

```
(10000, 28, 28)
```

In [19]:

```
len(test_labels)
```

Out[19]:

```
10000
```

In [20]:

```
test_labels
```

Out[20]:

```
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

Our workflow will be as follow: first we will present our neural network with the training data, `train_images` and `train_labels`. The network will then learn to associate images and labels. Finally, we will ask the network to produce predictions for `test_images`, and we will verify if these predictions match the labels from `test_labels`.

Let's build our network -- again, remember that you aren't supposed to understand everything about this example just yet.

In [29]:

```
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

The core building block of neural networks is the "layer", a data-processing module which you can conceive as a "filter" for data. Some data comes in, and comes out in a more useful form. Precisely, layers extract *representations* out of the data fed into them -- hopefully representations that are more meaningful for the problem at hand. Most of deep learning really consists of chaining together simple layers which will implement a form of progressive "data distillation". A deep learning model is like a sieve for data processing, made of a succession of increasingly refined data filters -- the "layers".

Here our network consists of a sequence of two `Dense` layers, which are densely-connected (also called "fully-connected") neural layers. The second (and last) layer is a 10-way "softmax" layer, which means it will return an array of 10 probability scores (summing to 1). Each score will be the probability that the current digit image belongs to one of our 10 digit classes.

To make our network ready for training, we need to pick three more things, as part of "compilation" step:

- A loss function: the is how the network will be able to measure how good a job it is doing on its training data, and thus how it will be able to steer itself in the right direction.
- An optimizer: this is the mechanism through which the network will update itself based on the data it sees and its loss function.
- Metrics to monitor during training and testing. Here we will only care about accuracy (the fraction of the images that were correctly classified).

The exact purpose of the loss function and the optimizer will be made clear throughout the next two chapters.

In [30]:

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

Before training, we will preprocess our data by reshaping it into the shape that the network expects, and scaling it so that all values are in the `[0, 1]` interval. Previously, our training images for instance were stored in an array of shape `(60000, 28, 28)` of type `uint8` with values in the `[0, 255]` interval. We transform it into a `float32` array of shape `(60000, 28 * 28)` with values between 0 and 1.

In [31]:

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

We also need to categorically encode the labels, a step which we explain in chapter 3:

In [32]:

```
from keras.utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

We are now ready to train our network, which in Keras is done via a call to the `fit` method of the network: we "fit" the model to its training data.

In [33]:

```
network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

```
WARNING:tensorflow:From C:\Users\Michael\Anaconda3\lib\site-packages
\tensorflow\python\ops\math_grad.py:1250: add_dispatch_support.<loca
ls>.wrapper (from tensorflow.python.ops.array_ops) is deprecated and
will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
WARNING:tensorflow:From C:\Users\Michael\Anaconda3\lib\site-packages
\keras\backend\tensorflow_backend.py:986: The name tf.assign_add is
deprecated. Please use tf.compat.v1.assign_add instead.

Epoch 1/5
60000/60000 [==============================] - 5s 76us/step - loss:
0.2560 - acc: 0.9251
Epoch 2/5
60000/60000 [==============================] - 4s 66us/step - loss:
0.1029 - acc: 0.9694
Epoch 3/5
60000/60000 [==============================] - 4s 68us/step - loss:
0.0679 - acc: 0.9798
Epoch 4/5
60000/60000 [==============================] - 4s 67us/step - loss:
0.0494 - acc: 0.9852
Epoch 5/5
60000/60000 [==============================] - 4s 66us/step - loss:
0.0371 - acc: 0.9888
```

Out[33]:

```
<keras.callbacks.History at 0x26dca5b4588>
```

Two quantities are being displayed during training: the "loss" of the network over the training data, and the accuracy of the network over the training data.

We quickly reach an accuracy of 0.989 (i.e. 98.9%) on the training data. Now let's check that our model performs well on the test set too:

In [34]:

```
test_loss, test_acc = network.evaluate(test_images, test_labels)
```

```
10000/10000 [==============================] - 0s 48us/step
```

In [35]:

```
print('test_acc:', test_acc)
```

```
test_acc: 0.9807
```

Our test set accuracy turns out to be c. 98% -- that's quite a bit lower than the training set accuracy. This gap between training accuracy and test accuracy is an example of "overfitting", the fact that machine learning models tend to perform worse on new data than on their training data. Overfitting will be a central topic in chapter 3.

This concludes our very first example -- you just saw how we could build and a train a neural network to classify handwritten digits, in less than 20 lines of Python code. In the next chapter, we will go in detail over every moving piece we just previewed, and clarify what is really going on behind the scenes. You will learn about "tensors", the data-storing objects going into the network, about tensor operations, which layers are made of, and about gradient descent, which allows our network to learn from its training examples.