In [ ]:

```
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]:

```
'2.3.0'
```

Understanding recurrent neural networks

A first recurrent layer in Keras

The process we just naively implemented in Numpy corresponds to an actual Keras layer: the `SimpleRNN` layer:

In [3]:

```
from keras.layers import SimpleRNN
```

There is just one minor difference: `SimpleRNN` processes batches of sequences, like all other Keras layers, not just a single sequence like in our Numpy example. This means that it takes inputs of shape `(batch_size, timesteps, input_features)` , rather than `(timesteps, input_features)` .

Like all recurrent layers in Keras, `SimpleRNN` can be run in two different modes: it can return either the full sequences of successive outputs for each timestep (a 3D tensor of shape `(batch_size, timesteps, output_features)`), or it can return only the last output for each input sequence (a 2D tensor of shape `(batch_size, output_features)`). These two modes are controlled by the `return_sequences` constructor argument. Let's take a look at an example:

In [4]:

```
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN

model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32))
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_1 (Embedding)      (None, None, 32)          320000    
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 32)                2080      
=================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
_________________________________________________________________
```

In [5]:

```
model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32, return_sequences=True))
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_2 (Embedding)      (None, None, 32)          320000    
_________________________________________________________________
simple_rnn_2 (SimpleRNN)     (None, None, 32)          2080      
=================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
_________________________________________________________________
```

It is sometimes useful to stack several recurrent layers one after the other in order to increase the representational power of a network. In such a setup, you have to get all intermediate layers to return full sequences:

In [6]:

```
model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32))  # This last layer only returns the last outputs.
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_3 (Embedding)      (None, None, 32)          320000    
_________________________________________________________________
simple_rnn_3 (SimpleRNN)     (None, None, 32)          2080      
_________________________________________________________________
simple_rnn_4 (SimpleRNN)     (None, None, 32)          2080      
_________________________________________________________________
simple_rnn_5 (SimpleRNN)     (None, None, 32)          2080      
_________________________________________________________________
simple_rnn_6 (SimpleRNN)     (None, 32)                2080      
=================================================================
Total params: 328,320
Trainable params: 328,320
Non-trainable params: 0
_________________________________________________________________
```

Now let's try to use such a model on the IMDB movie review classification problem. First, let's preprocess the data:

In [7]:

```
from keras.datasets import imdb
from keras.preprocessing import sequence

max_features = 10000  # number of words to consider as features
maxlen = 500  # cut texts after this number of words (among top max_features mos
t common words)
batch_size = 32

print('Loading data...')
(input_train, y_train), (input_test, y_test) = imdb.load_data(num_words=max_feat
ures)
print(len(input_train), 'train sequences')
print(len(input_test), 'test sequences')

print('Pad sequences (samples x time)')
input_train = sequence.pad_sequences(input_train, maxlen=maxlen)
input_test = sequence.pad_sequences(input_test, maxlen=maxlen)
print('input_train shape:', input_train.shape)
print('input_test shape:', input_test.shape)
```

```
Loading data...
25000 train sequences
25000 test sequences
Pad sequences (samples x time)
input_train shape: (25000, 500)
input_test shape: (25000, 500)
```

Let's train a simple recurrent network using an `Embedding` layer and a `SimpleRNN` layer:

In [8]:

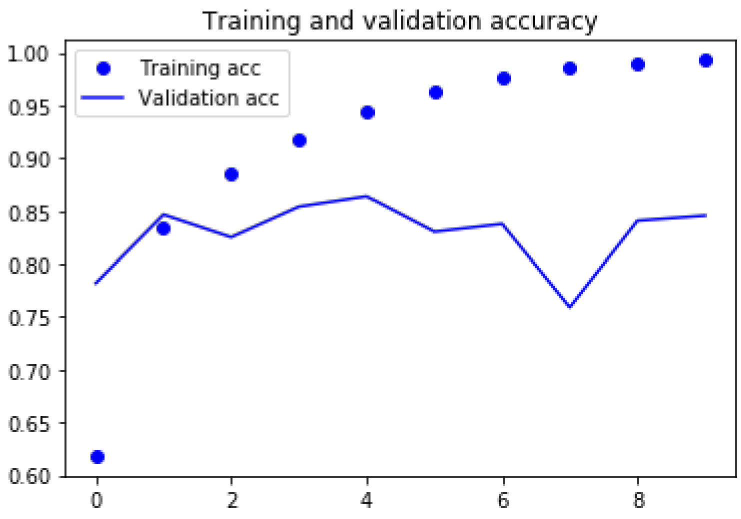
```python
from keras.layers import Dense

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

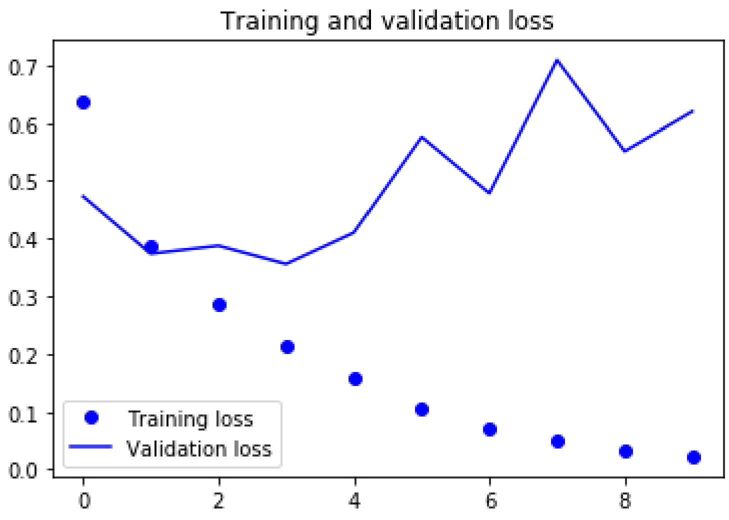
```
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 25s 1ms/step - loss:
0.6377 - acc: 0.6183 - val_loss: 0.4731 - val_acc: 0.7818
Epoch 2/10
20000/20000 [==============================] - 25s 1ms/step - loss:
0.3874 - acc: 0.8346 - val_loss: 0.3737 - val_acc: 0.8466
Epoch 3/10
20000/20000 [==============================] - 25s 1ms/step - loss:
0.2868 - acc: 0.8858 - val_loss: 0.3876 - val_acc: 0.8252
Epoch 4/10
20000/20000 [==============================] - 25s 1ms/step - loss:
0.2149 - acc: 0.9181 - val_loss: 0.3559 - val_acc: 0.8540
Epoch 5/10
20000/20000 [==============================] - 25s 1ms/step - loss:
0.1563 - acc: 0.9441 - val_loss: 0.4103 - val_acc: 0.8638
Epoch 6/10
20000/20000 [==============================] - 25s 1ms/step - loss:
0.1075 - acc: 0.9632 - val_loss: 0.5760 - val_acc: 0.8304
Epoch 7/10
20000/20000 [==============================] - 25s 1ms/step - loss:
0.0728 - acc: 0.9766 - val_loss: 0.4786 - val_acc: 0.8378
Epoch 8/10
20000/20000 [==============================] - 25s 1ms/step - loss:
0.0487 - acc: 0.9857 - val_loss: 0.7091 - val_acc: 0.7590
Epoch 9/10
20000/20000 [==============================] - 25s 1ms/step - loss:
0.0326 - acc: 0.9901 - val_loss: 0.5512 - val_acc: 0.8408
Epoch 10/10
20000/20000 [==============================] - 24s 1ms/step - loss:
0.0228 - acc: 0.9931 - val_loss: 0.6201 - val_acc: 0.8456
```

Let's display the training and validation loss and accuracy:

In [11]:

```python
import matplotlib.pyplot as plt

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

epochs = range(len(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.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show();
```





As a reminder, in chapter 3, our very first naive approach to this very dataset got us to 88% test accuracy. Unfortunately, our small recurrent network doesn't perform very well at all compared to this baseline (only up to 85% validation accuracy). Part of the problem is that our inputs only consider the first 500 words rather the full sequences -- hence our RNN has access to less information than our earlier baseline model. The remainder of the problem is simply that `SimpleRNN` isn't very good at processing long sequences, like text. Other types of recurrent layers perform much better. Let's take a look at some more advanced layers.

[...]

A concrete LSTM example in Keras

Now let's switch to more practical concerns: we will set up a model using a LSTM layer and train it on the IMDB data. Here's the network, similar to the one with `SimpleRNN` that we just presented. We only specify the output dimensionality of the LSTM layer, and leave every other argument (there are lots) to the Keras defaults. Keras has good defaults, and things will almost always "just work" without you having to spend time tuning parameters by hand.

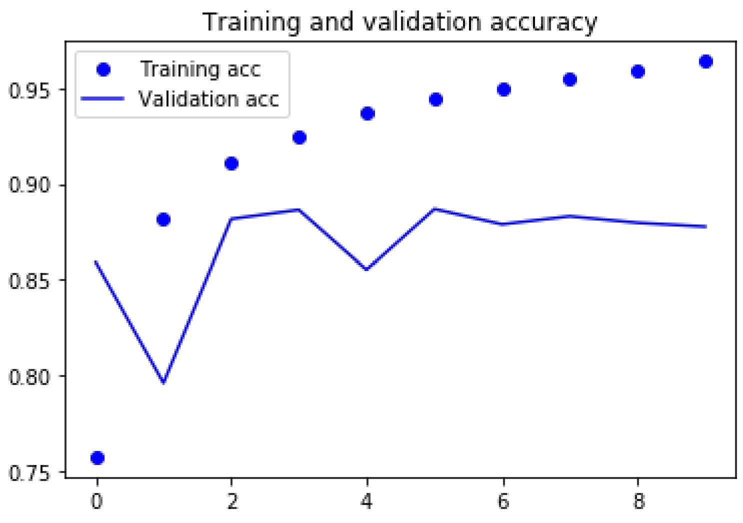
In [12]:

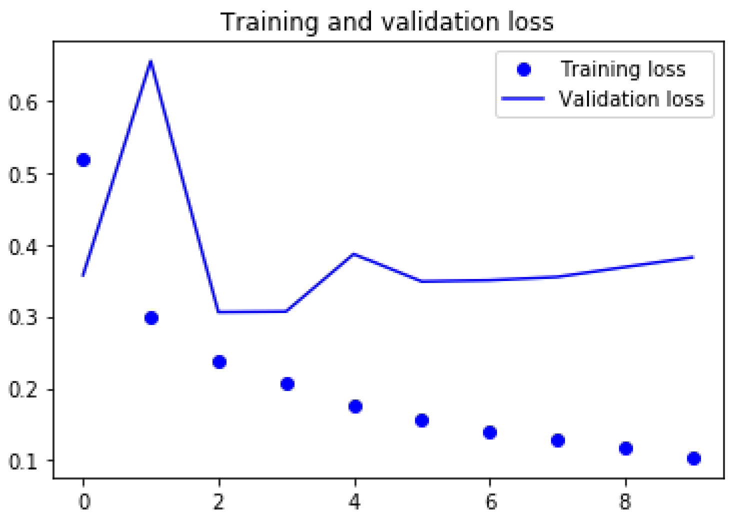
```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 53s 3ms/step - loss:
0.5178 - acc: 0.7574 - val_loss: 0.3573 - val_acc: 0.8590
Epoch 2/10
20000/20000 [==============================] - 53s 3ms/step - loss:
0.2998 - acc: 0.8819 - val_loss: 0.6546 - val_acc: 0.7960
Epoch 3/10
20000/20000 [==============================] - 53s 3ms/step - loss:
0.2366 - acc: 0.9111 - val_loss: 0.3057 - val_acc: 0.8816
Epoch 4/10
20000/20000 [==============================] - 53s 3ms/step - loss:
0.2073 - acc: 0.9251 - val_loss: 0.3066 - val_acc: 0.8864
Epoch 5/10
20000/20000 [==============================] - 53s 3ms/step - loss:
0.1754 - acc: 0.9372 - val_loss: 0.3865 - val_acc: 0.8552
Epoch 6/10
20000/20000 [==============================] - 53s 3ms/step - loss:
0.1556 - acc: 0.9445 - val_loss: 0.3484 - val_acc: 0.8868
Epoch 7/10
20000/20000 [==============================] - 53s 3ms/step - loss:
0.1395 - acc: 0.9502 - val_loss: 0.3501 - val_acc: 0.8788
Epoch 8/10
20000/20000 [==============================] - 54s 3ms/step - loss:
0.1291 - acc: 0.9554 - val_loss: 0.3547 - val_acc: 0.8830
Epoch 9/10
20000/20000 [==============================] - 54s 3ms/step - loss:
0.1192 - acc: 0.9592 - val_loss: 0.3684 - val_acc: 0.8796
Epoch 10/10
20000/20000 [==============================] - 54s 3ms/step - loss:
0.1043 - acc: 0.9643 - val_loss: 0.3820 - val_acc: 0.8776
```

In [13]:

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(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.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show();
```


Training and validation accuracy
● Training acc
Validation acc


Training and validation loss
● Training loss
Validation loss