7. Workshop 2-2 : IMDB - NLP - Word Embedding

> **[Reference] :**
>
> François Chollet, **Deep Learning with Python**, Chapter 6, Section 1, Manning, 2018. http://www.deeplearningitalia.com/wp-content/uploads/2017/12/Dropbox_Chollet.pdf (http://www.deeplearningitalia.com/wp-content/uploads/2017/12/Dropbox_Chollet.pdf)

Another popular and powerful way to associate a vector with a word is the use of dense "word vectors", also called "word embeddings". While the vectors obtained through one-hot encoding are binary, sparse (mostly made of zeros) and very high-dimensional (same dimensionality as the number of words in the vocabulary), "word embeddings" are low-dimensional floating point vectors (i.e. "dense" vectors, as opposed to sparse vectors). Unlike word vectors obtained via one-hot encoding, word embeddings are learned from data. It is common to see word embeddings that are 256-dimensional, 512-dimensional, or 1024-dimensional when dealing with very large vocabularies. On the other hand, one-hot encoding words generally leads to vectors that are 20,000-dimensional or higher (capturing a vocabulary of 20,000 token in this case). So, word embeddings pack more information into far fewer dimensions.

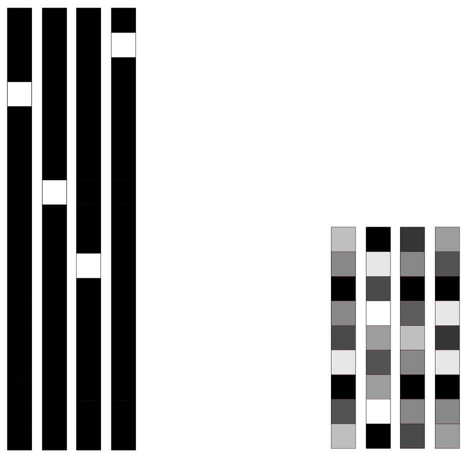
In [1]:

```
import keras
keras.__version__
```

```
/Users/macmini1/anaconda3/lib/python3.6/site-packages/h
5py/__init__.py:36: FutureWarning: Conversion of the se
cond argument of issubdtype from `float` to `np.floatin
g` is deprecated. In future, it will be treated as `np.
float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_c
onverters
Using TensorFlow backend.
```

Out[1]:

```
'2.2.4'
```



There are two ways to obtain word embeddings:

- Learn word embeddings jointly with the main task you care about (e.g. document classification or sentiment prediction). In this setup, you would start with random word vectors, then learn your word vectors in the same way that you learn the weights of a neural network.
- Load into your model word embeddings that were pre-computed using a different machine learning task than the one you are trying to solve. These are called "pre-trained word embeddings".

Let's take a look at both.

Learning word embeddings with the `Embedding` layer

The simplest way to associate a dense vector to a word would be to pick the vector at random. The problem with this approach is that the resulting embedding space would have no structure: for instance, the words "accurate" and "exact" may end up with completely different embeddings, even though they are interchangeable in most sentences. It would be very difficult for a deep neural network to make sense of such a noisy, unstructured embedding space.

To get a bit more abstract: the geometric relationships between word vectors should reflect the semantic relationships between these words. Word embeddings are meant to map human language into a geometric space. For instance, in a reasonable embedding space, we would expect synonyms to be embedded into similar word vectors, and in general we would expect the geometric distance (e.g. L2 distance) between any two word vectors to relate to the semantic distance of the associated words (words meaning very different things would be embedded to points far away from each other, while related words would be closer). Even beyond mere distance, we may want specific **directions** in the embedding space to be meaningful.

In real-world word embedding spaces, common examples of meaningful geometric transformations are "gender vectors" and "plural vector". For instance, by adding a "female vector" to the vector "king", one obtain the vector "queen". By adding a "plural vector", one obtain "kings". Word embedding spaces typically feature thousands of such interpretable and potentially useful vectors.

Is there some "ideal" word embedding space that would perfectly map human language and could be used for any natural language processing task? Possibly, but in any case, we have yet to compute anything of the sort. Also, there isn't such a thing as "human language", there are many different languages and they are not isomorphic, as a language is the reflection of a specific culture and a specific context. But more pragmatically, what makes a good word embedding space depends heavily on your task: the perfect word embedding space for an English-language movie review sentiment analysis model may look very different from the perfect embedding space for an English-language legal document classification model, because the importance of certain semantic relationships varies from task to task.

It is thus reasonable to **learn** a new embedding space with every new task. Thankfully, backpropagation makes this really easy, and Keras makes it even easier. It's just about learning the weights of a layer: the `Embedding` layer.

In [2]:

```
from keras.layers import Embedding

# The Embedding layer takes at least two arguments:
# the number of possible tokens, here 1000 (1 + maximum word ind
# and the dimensionality of the embeddings, here 64.
embedding_layer = Embedding(1000, 64)
```

The `Embedding` layer is best understood as a dictionary mapping integer indices (which stand for specific words) to dense vectors. It takes as input integers, it looks up these integers into an internal dictionary, and it returns the associated vectors. It's effectively a dictionary lookup.

The `Embedding` layer takes as input a 2D tensor of integers, of shape `(samples, sequence_length)`, where each entry is a sequence of integers. It can embed sequences of variable lengths, so for instance we could feed into our embedding layer above batches that could have shapes `(32, 10)` (batch of 32 sequences of length 10) or `(64, 15)` (batch of 64 sequences of length 15). All sequences in a batch must have the same length, though (since we need to pack them into a single tensor), so sequences that are shorter than others should be padded with zeros, and sequences that are longer should be truncated.

This layer returns a 3D floating point tensor, of shape `(samples, sequence_length, embedding_dimensionality)`. Such a 3D tensor can then be processed by a RNN layer or a 1D convolution layer (both will be introduced in the next sections).

When you instantiate an `Embedding` layer, its weights (its internal dictionary of token vectors) are initially random, just like with any other layer. During training, these word vectors will be gradually adjusted via backpropagation, structuring the space into something that the downstream model can exploit. Once fully trained, your embedding space will show a lot of structure -- a kind of structure specialized for the specific problem you were training your model for.

Let's apply this idea to the IMDB movie review sentiment prediction task that you are already familiar with. Let's quickly prepare the data. We will restrict the movie reviews to the top 10,000 most common words (like we did the first time we worked with this dataset), and cut the reviews after only 20 words. Our network will simply learn 8-dimensional embeddings for each of the 10,000 words, turn the input integer sequences (2D integer tensor) into embedded sequences (3D float tensor), flatten the tensor to 2D, and train a single `Dense` layer on top for classification.

In [3]:

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

# Number of words to consider as features
max_features = 10000
# Cut texts after this number of words
# (among top max_features most common words)
maxlen = 20

# Load the data as lists of integers.
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=

# This turns our lists of integers
# into a 2D integer tensor of shape `(samples, maxlen)`
x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=m
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=max
```

In [4]:

```
from keras.models import Sequential
from keras.layers import Flatten, Dense

model = Sequential()
# We specify the maximum input length to our Embedding layer
# so we can later flatten the embedded inputs
model.add(Embedding(10000, 8, input_length=maxlen))
# After the Embedding layer,
# our activations have shape `(samples, maxlen, 8)`.

# We flatten the 3D tensor of embeddings
# into a 2D tensor of shape `(samples, maxlen * 8)`
model.add(Flatten())

# We add the classifier on top
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', m
model.summary()

history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_split=0.2)
```

```
_______________________________________________________________
__________
Layer (type)                 Output Shape
Param #
=======================================================================
==========
embedding_2 (Embedding)      (None, 20, 8)
80000
_______________________________________________________________
__________
flatten_1 (Flatten)          (None, 160)
0
_______________________________________________________________
__________
dense_1 (Dense)              (None, 1)
161
=======================================================================
==========
Total params: 80,161
Trainable params: 80,161
Non-trainable params: 0
_______________________________________________________________
__________
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 1s 74us/
step - loss: 0.6759 - acc: 0.6048 - val_loss: 0.6400 -
val_acc: 0.6806
Epoch 2/10
20000/20000 [==============================] - 1s 59us/
step - loss: 0.5661 - acc: 0.7431 - val_loss: 0.5471 -
```

```
val_acc: 0.7198
Epoch 3/10
20000/20000 [==============================] - 1s 59us/
step - loss: 0.4754 - acc: 0.7808 - val_loss: 0.5116 -
val_acc: 0.7382
Epoch 4/10
20000/20000 [==============================] - 1s 57us/
step - loss: 0.4264 - acc: 0.8078 - val_loss: 0.5010 -
val_acc: 0.7448
Epoch 5/10
20000/20000 [==============================] - 1s 58us/
step - loss: 0.3928 - acc: 0.8261 - val_loss: 0.4982 -
val_acc: 0.7530
Epoch 6/10
20000/20000 [==============================] - 1s 58us/
step - loss: 0.3665 - acc: 0.8400 - val_loss: 0.5015 -
val_acc: 0.7526
Epoch 7/10
20000/20000 [==============================] - 1s 58us/
step - loss: 0.3430 - acc: 0.8539 - val_loss: 0.5054 -
val_acc: 0.7524
Epoch 8/10
20000/20000 [==============================] - 1s 59us/
step - loss: 0.3217 - acc: 0.8658 - val_loss: 0.5135 -
val_acc: 0.7478
Epoch 9/10
20000/20000 [==============================] - 1s 61us/
step - loss: 0.3016 - acc: 0.8769 - val_loss: 0.5218 -
val_acc: 0.7484
Epoch 10/10
20000/20000 [==============================] - 1s 62us/
step - loss: 0.2832 - acc: 0.8857 - val_loss: 0.5308 -
val_acc: 0.7470
```

We get to a validation accuracy of ~76%, which is pretty good considering that we only look at the first 20 words in every review. But note that merely flattening the embedded sequences and training a single `Dense` layer on top leads to a model that treats each word in the input sequence separately, without considering inter-word relationships and structure sentence (e.g. it would likely treat both "*this movie is shit*" and "*this movie is the shit*" as being negative "reviews"). It would be much better to add recurrent layers or 1D convolutional layers on top of the embedded sequences to learn features that take into account each sequence as a whole. That's what we will focus on in the next few sections.

Using pre-trained word embeddings

Sometimes, you have so little training data available that could never use your data alone to learn an appropriate task-specific embedding of your vocabulary. What to do then?

Instead of learning word embeddings jointly with the problem you want to solve, you could be loading embedding vectors from a pre-computed embedding space known to be highly structured and to exhibit useful properties -- that captures generic aspects of language structure. The rationale behind using pre-trained word embeddings in natural language processing is very much the same as for using pre-trained convnets in image classification: we don't have enough data available to learn truly powerful features on our own, but we expect the features that we need to be fairly generic, i.e. common visual features or semantic features. In this case it makes sense to reuse features learned on a different problem.

Such word embeddings are generally computed using word occurrence statistics (observations about what words co-occur in sentences or documents), using a variety of techniques, some involving neural networks, others not. The idea of a dense, low-dimensional embedding space for words, computed in an unsupervised way, was initially explored by Bengio et al. in the early 2000s, but it only started really taking off in research and industry applications after the release of one of the most famous and successful word embedding scheme: the Word2Vec algorithm, developed by Mikolov at Google in 2013. Word2Vec dimensions capture specific semantic properties, e.g. gender.

There are various pre-computed databases of word embeddings that can download and start using in a Keras `Embedding` layer. Word2Vec is one of them. Another popular one is called "GloVe", developed by Stanford researchers in 2014. It stands for "Global Vectors for Word Representation", and it is an embedding technique based on factorizing a matrix of word co-occurrence statistics. Its developers have made available pre-computed embeddings for millions of English tokens, obtained from Wikipedia data or from Common Crawl data.

Let's take a look at how you can get started using GloVe embeddings in a Keras model. The same method will of course be valid for Word2Vec embeddings or any other word embedding database that you can download. We will also use this example to refresh the text tokenization techniques we introduced a few paragraphs ago: we will start from raw text, and work our way up.

Putting it all together: from raw text to word embeddings

We will be using a model similar to the one we just went over -- embedding sentences in sequences of vectors, flattening them and training a `Dense` layer on top. But we will do it using pre-trained word embeddings, and instead of using the pre-tokenized IMDB data packaged in Keras, we will start from scratch, by downloading the original text data.

Download the IMDB data as raw text

First, head to `http://ai.stanford.edu/~amaas/data/sentiment/` and download the raw IMDB dataset (if the URL isn't working anymore, just Google "IMDB dataset"). Uncompress it.

Now let's collect the individual training reviews into a list of strings, one string per review, and let's also collect the review labels (positive / negative) into a `labels` list:

In [5]:

```
import os

imdb_dir = './aclImdb' # directory for file folder : aclImdb
train_dir = os.path.join(imdb_dir, 'train')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(train_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname), encoding='ut
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)
```

Tokenize the data

Let's vectorize the texts we collected, and prepare a training and validation split. We will merely be using the concepts we introduced earlier in this section.

Because pre-trained word embeddings are meant to be particularly useful on problems where little training data is available (otherwise, task-specific embeddings are likely to outperform them), we will add the following twist: we restrict the training data to its first 200 samples. So we will be learning to classify movie reviews after looking at just 200 examples...

In [6]:

```python
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

maxlen = 100  # We will cut reviews after 100 words
training_samples = 200  # We will be training on 200 samples
validation_samples = 10000  # We will be validating on 10000 sam
max_words = 10000  # We will only consider the top 10,000 words

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)

# Split the data into a training set and a validation set
# But first, shuffle the data, since we started from data
# where sample are ordered (all negative first, then all positiv
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]

x_train = data[:training_samples]
y_train = labels[:training_samples]
x_val = data[training_samples: training_samples + validation_sam
y_val = labels[training_samples: training_samples + validation_s
```

```
Found 88582 unique tokens.
Shape of data tensor: (25000, 100)
Shape of label tensor: (25000,)
```

Download the GloVe word embeddings

Head to `https://nlp.stanford.edu/projects/glove/` (where you can learn more about the GloVe algorithm), and download the pre-computed embeddings from 2014 English Wikipedia. It's a 822MB zip file named `glove.6B.zip`, containing 100-dimensional embedding vectors for 400,000 words (or non-word tokens). Un-zip it.

Pre-process the embeddings

Let's parse the un-zipped file (it's a `txt` file) to build an index mapping words (as strings) to their vector representation (as number vectors).

In [7]:

```python
glove_dir = './glove.6B'

embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))
```

```
Found 400000 word vectors.
```

Now let's build an embedding matrix that we will be able to load into an `Embedding` layer. It must be a matrix of shape `(max_words, embedding_dim)` , where each entry `i` contains the `embedding_dim` -dimensional vector for the word of index `i` in our reference word index (built during tokenization). Note that the index `0` is not supposed to stand for any word or token -- it's a placeholder.

In [8]:

```python
embedding_dim = 100

embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if i < max_words:
        if embedding_vector is not None:
            # Words not found in embedding index will be all-zer
            embedding_matrix[i] = embedding_vector
```

Define a model

We will be using the same model architecture as before:

In [9]:

```python
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxle
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

```
_______________________________________________________
__________
Layer (type)                 Output Shape
Param #
=======================================================
==========
embedding_3 (Embedding)      (None, 100, 100)
1000000
_______________________________________________________
__________
flatten_2 (Flatten)          (None, 10000)
0
_______________________________________________________
__________
dense_2 (Dense)              (None, 32)
320032
_______________________________________________________
__________
dense_3 (Dense)              (None, 1)
33
=======================================================
==========
Total params: 1,320,065
Trainable params: 1,320,065
Non-trainable params: 0
_______________________________________________________
__________
```

Load the GloVe embeddings in the model

The `Embedding` layer has a single weight matrix: a 2D float matrix where each entry `i` is the word vector meant to be associated with index `i`. Simple enough. Let's just load the GloVe matrix we prepared into our `Embedding` layer, the first layer in our model:

In [10]:

```python
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
```

Additionally, we freeze the embedding layer (we set its `trainable` attribute to `False`), following the same rationale as what you are already familiar with in the context of pre-trained convnet features: when parts of a model are pre-trained (like our `Embedding` layer), and parts are randomly initialized (like our classifier), the pre-trained parts should not be updated during training to avoid forgetting what they already know. The large gradient update triggered by the randomly initialized layers would be very disruptive to the already learned features.

Train and evaluate

Let's compile our model and train it:

In [11]:

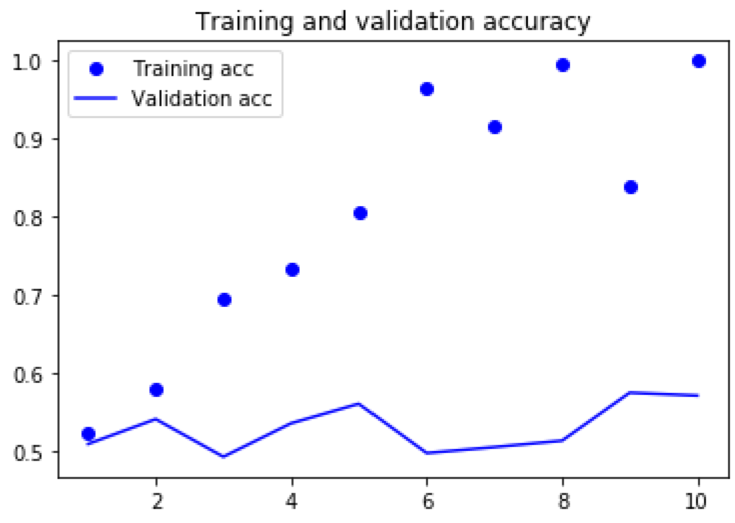
```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_data=(x_val, y_val))
model.save_weights('pre_trained_glove_model.h5')
```

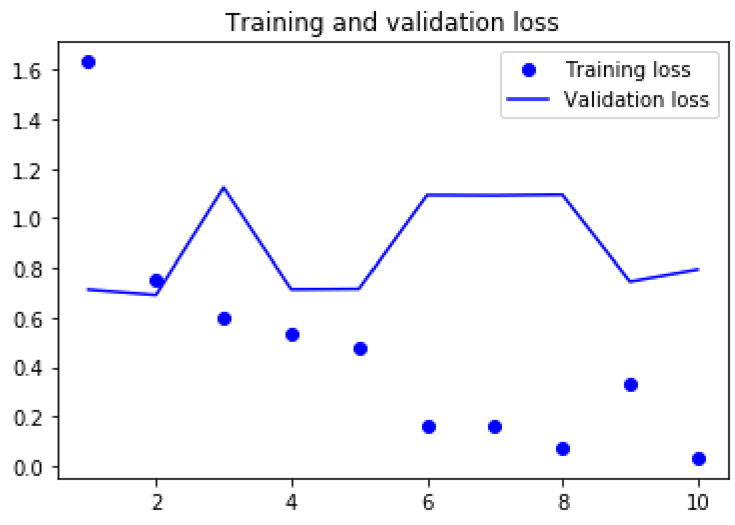
```
Train on 200 samples, validate on 10000 samples
Epoch 1/10
200/200 [==============================] - 1s 5ms/step
- loss: 1.6337 - acc: 0.5250 - val_loss: 0.7130 - val_a
cc: 0.5100
Epoch 2/10
200/200 [==============================] - 1s 3ms/step
- loss: 0.7565 - acc: 0.5800 - val_loss: 0.6910 - val_a
cc: 0.5418
Epoch 3/10
200/200 [==============================] - 1s 3ms/step
- loss: 0.5956 - acc: 0.6950 - val_loss: 1.1244 - val_a
cc: 0.4936
Epoch 4/10
200/200 [==============================] - 1s 3ms/step
- loss: 0.5342 - acc: 0.7350 - val_loss: 0.7134 - val_a
cc: 0.5364
Epoch 5/10
200/200 [==============================] - 1s 4ms/step
- loss: 0.4746 - acc: 0.8050 - val_loss: 0.7156 - val_a
cc: 0.5614
Epoch 6/10
200/200 [==============================] - 1s 3ms/step
- loss: 0.1649 - acc: 0.9650 - val_loss: 1.0934 - val_a
cc: 0.4983
Epoch 7/10
200/200 [==============================] - 1s 3ms/step
- loss: 0.1640 - acc: 0.9150 - val_loss: 1.0923 - val_a
cc: 0.5060
Epoch 8/10
200/200 [==============================] - 1s 3ms/step
- loss: 0.0760 - acc: 0.9950 - val_loss: 1.0953 - val_a
cc: 0.5142
Epoch 9/10
200/200 [==============================] - 1s 3ms/step
- loss: 0.3276 - acc: 0.8400 - val_loss: 0.7451 - val_a
cc: 0.5756
Epoch 10/10
200/200 [==============================] - 1s 3ms/step
- loss: 0.0326 - acc: 1.0000 - val_loss: 0.7939 - val_a
cc: 0.5719
```

Let's plot its performance over time:

In [13]:

```
import matplotlib.pyplot as plt
%matplotlib inline

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)

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()
```





The model quickly starts overfitting, unsurprisingly given the small number of training samples. Validation accuracy has high variance for the same reason, but seems to reach high 50s.

Note that your mileage may vary: since we have so few training samples, performance is heavily dependent on which exact 200 samples we picked, and we picked them at random. If it worked really poorly for you, try picking a different random set of 200 samples, just for the sake of the exercise (in real life you don't get to pick your training data).

We can also try to train the same model without loading the pre-trained word embeddings and without freezing the embedding layer. In that case, we would be learning a task-specific embedding of our input tokens, which is generally more powerful than pre-trained word embeddings when lots of data is available. However, in our case, we have only 200 training samples. Let's try it:

In [14]:

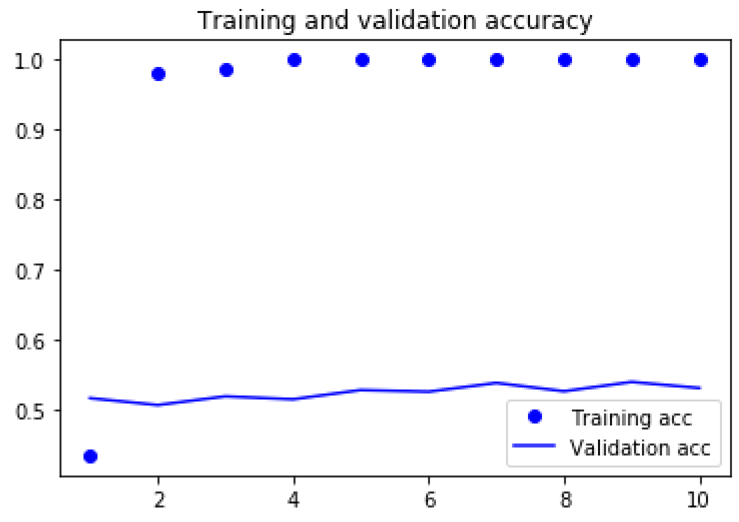
```python
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxle
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_data=(x_val, y_val))
```

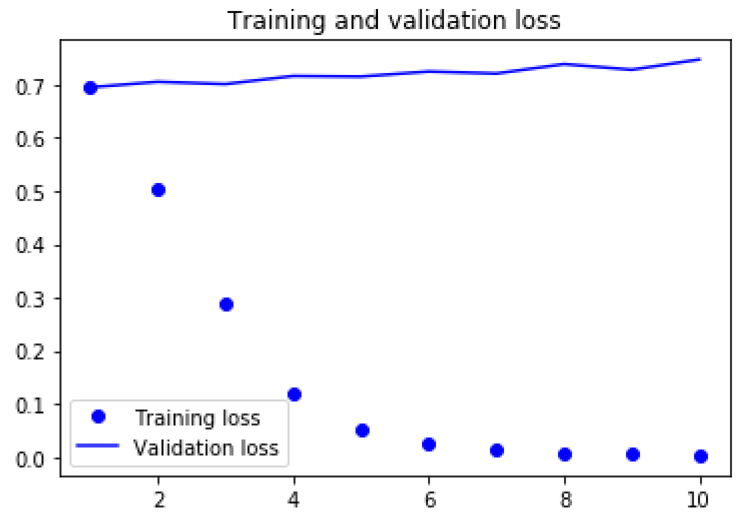
```
_______________________________________________________
__________
Layer (type)                 Output Shape
Param #
=======================================================
==========
embedding_4 (Embedding)      (None, 100, 100)
1000000
_______________________________________________________
__________
flatten_3 (Flatten)          (None, 10000)
0
_______________________________________________________
__________
dense_4 (Dense)              (None, 32)
320032
_______________________________________________________
__________
dense_5 (Dense)              (None, 1)
33
=======================================================
==========
Total params: 1,320,065
Trainable params: 1,320,065
Non-trainable params: 0
_______________________________________________________
__________
Train on 200 samples, validate on 10000 samples
Epoch 1/10
200/200 [==============================] - 1s 6ms/step
- loss: 0.6951 - acc: 0.4350 - val_loss: 0.6950 - val_a
cc: 0.5167
Epoch 2/10
200/200 [==============================] - 1s 4ms/step
- loss: 0.5027 - acc: 0.9800 - val_loss: 0.7055 - val_a
cc: 0.5069
Epoch 3/10
```

```
200/200 [==============================] - 1s 4ms/step
- loss: 0.2893 - acc: 0.9850 - val_loss: 0.7012 - val_a
cc: 0.5193
Epoch 4/10
200/200 [==============================] - 1s 4ms/step
- loss: 0.1181 - acc: 1.0000 - val_loss: 0.7164 - val_a
cc: 0.5151
Epoch 5/10
200/200 [==============================] - 1s 4ms/step
- loss: 0.0522 - acc: 1.0000 - val_loss: 0.7153 - val_a
cc: 0.5283
Epoch 6/10
200/200 [==============================] - 1s 3ms/step
- loss: 0.0261 - acc: 1.0000 - val_loss: 0.7249 - val_a
cc: 0.5260
Epoch 7/10
200/200 [==============================] - 1s 4ms/step
- loss: 0.0141 - acc: 1.0000 - val_loss: 0.7211 - val_a
cc: 0.5385
Epoch 8/10
200/200 [==============================] - 1s 3ms/step
- loss: 0.0082 - acc: 1.0000 - val_loss: 0.7388 - val_a
cc: 0.5266
Epoch 9/10
200/200 [==============================] - 1s 3ms/step
- loss: 0.0049 - acc: 1.0000 - val_loss: 0.7282 - val_a
cc: 0.5399
Epoch 10/10
200/200 [==============================] - 1s 4ms/step
- loss: 0.0030 - acc: 1.0000 - val_loss: 0.7475 - val_a
cc: 0.5312
```

In [15]:

```
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)

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()
```





Validation accuracy stalls in the low 50s. So in our case, pre-trained word embeddings does outperform jointly learned embeddings. If you increase the number of training samples, this will quickly stop being the case -- try it as an exercise.

Finally, let's evaluate the model on the test data. First, we will need to tokenize the test data:

In [16]:

```
test_dir = os.path.join(imdb_dir, 'test')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(test_dir, label_type)
    for fname in sorted(os.listdir(dir_name)):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)

sequences = tokenizer.texts_to_sequences(texts)
x_test = pad_sequences(sequences, maxlen=maxlen)
y_test = np.asarray(labels)
```

And let's load and evaluate the first model:

In [17]:

```
model.load_weights('pre_trained_glove_model.h5')
model.evaluate(x_test, y_test)
```

```
25000/25000 [==============================] - 1s 53us/
step
```

Out[17]:

```
[0.7961385548210144, 0.56976]
```

We get an appalling test accuracy of 54%. Working with just a handful of training samples is hard!