Project: Sentiment Analysis Using Deep Learning

Working with text dataset

Text data can be understood as either a sequence of characters or a sequence of words, but we mostly work at the level of the words. The deep-learning techniques models we are using here can be used to produce a basic form of natural language understanding, enough for sentiment analysis.

However, we need to keep in mind that, none of these techniques can be compared to understand the text in a human intelligence; but these techniques can map the mathematical structure of written language, which is enough to solve many simple text tasks. Deep learning as other neural networks techniques works only with numeric tensors. Vectorizing text is the process of transforming text data into numeric tensors. A tensor is a container for data – almost always numerical data. So, it is a container for numbers.

Multiple ways of vectorizing text data:

- Segment text into words, and transform each word into a vector
- Segment text into characters, and transform each character into a vector
- Extract n-grams of words or characters transform each n-gram into a vector.

So, the different units into which we break down text are called tokens, and breaking text into such tokens is called tokenization. There are many ways to associate a vector with a token. i.e., one-hot encoding and token embedding, which is typically called word embedding. We will be using word embedding in our study.

Word embeddings:

- Dense
- Lower-dimensional
- Learned from data

One-hot encoding:

- spare
- High-dimensional
- Hardcoded

Word embeddings

In [1]:
```
import keras
keras.__version__
```

```
/anaconda/envs/py35/lib/python3.5/site-packages/h5py/__init__.py:36: FutureWa
rning: Conversion of the second argument of issubdtype from `float` to `np.fl
oating` is deprecated. In future, it will be treated as `np.float64 == np.dty
pe(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Out[1]:
```
'2.2.4'
```

Let's start with word embedding, which is one of the popular way to associate a vector with a word. "word embeddings" are low-dimensional floating point vectors. Unlike word vectors obtained via one-hot encoding, word embeddings are learned from data. It is not a suprise to see word embeddings that are 256, 512, or 1024-dimensional when dealing with very large vocabularies.

There are two approaches to using word embeddings:

- Learn word embeddings jointly with the document classification or sentiment prediction. Here, we can start with random word vectors, then learn our word vectors in the same way as we do learn the weights of a neural network.
- Use a pre-trained word embeddings that were pre-computed using a different task.

From raw text to word embeddings

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, we will start from scratch, by downloading the original text data.

Download the IMDB data as raw text

First, we need to get the data from the Stanford AI website at http://ai.stanford.edu/~amaas/data/sentiment/ (http://ai.stanford.edu/~amaas/data/sentiment/) and download the raw 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:

Processing the labels of the raw movie data

In [2]:
```
import os

movies_dir = '/data/home/dlvmadmin/notebooks/Deep_learning_python/data/aclImd
b/'
movies_train_dir = os.path.join(movies_dir, 'train')

movie_labels = []
movie_texts = []

for label_type in ['neg', 'pos']:
    movie_dir_name = os.path.join(movies_train_dir, label_type)
    for fname in os.listdir(movie_dir_name):
        if fname[-4:] == '.txt':
            movie_file = open(os.path.join(movie_dir_name, fname))
            movie_texts.append(movie_file.read())
            movie_file.close()
            if label_type == 'neg':
                movie_labels.append(0)
            else:
                movie_labels.append(1)
```

Tokenizing the text of the raw movie data

Let's vectorize the texts we downloaded, and prepare a training and validation split. Pre-trained word embeddings are meant to be particularly useful on problems where little training data is available but, task-specific embeddings are likely to outperform them.

In [3]:
```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

# We'll cut reviews after 200 words
maxlen = 200

# We'll be training on 12500 samples
movie_training_samples = 12500

# We'll be validating on 15000 samples
movie_validation_samples = 15000

# We'll only consider the top 15000 words in the dataset
max_words = 15000

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

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

movie_data = pad_sequences(sequences, maxlen=maxlen)

movie_labels = np.asarray(movie_labels)
print('Tensor shape for data:', movie_data.shape)
print('Tensor shape for label:', movie_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 positive).
indices = np.arange(movie_data.shape[0])
np.random.shuffle(indices)
movie_data = movie_data[indices]
movie_labels = movie_labels[indices]

movie_data_train = movie_data[:movie_training_samples]

movie_labels_train = movie_labels[:movie_training_samples]

movie_data_val = movie_data[movie_training_samples: movie_training_samples +
                            movie_validation_samples]
movie_labels_val = movie_labels[movie_training_samples: movie_training_samples
+
                                movie_validation_samples]
```

```
Found 88582 unique tokens.
Tensor shape for data: (25000, 200)
Tensor shape for label: (25000,)
```

Download the GloVe word embeddings

"GloVe project is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from the 2014 English Wikipedia corpus, and the resulting representations showcase interesting linear substructures of the word vector space." Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014.

We can learn more about the algorithm by visiting: https://nlp.stanford.edu/projects/glove/ (https://nlp.stanford.edu/projects/glove/) It's a 822MB zip file named glove.6B.zip, containing 100-dimensional embedding vectors for 400,000 words (or non-word tokens).

Pre-process the embeddings

Let's parse the file to build an index mapping words (as strings) to their vector representation (as number vectors).

In [4]:
```
glove_word_dir = '/data/home/dlvmadmin/notebooks/Deep_learning_python/data/'

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

print('There are %s word vectors.' % len(word_embeddings_index))
```

```
There are 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).

Preparing the GloVe word-embeddings matrix

In [5]:
```
word_embedding_dim = 100

word_embedding_matrix = np.zeros((max_words, word_embedding_dim))
for word, i in word_index.items():
    word_embedding_vector = word_embeddings_index.get(word)
    if i < max_words:
        if word_embedding_vector is not None:
            # Words not found in embedding index will be all-zeros.
            word_embedding_matrix[i] = word_embedding_vector
```

Define a model

We'll be using the following model architecture:

In [6]:
```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

word_embedding_model = Sequential()
word_embedding_model.add(Embedding(max_words, word_embedding_dim, input_length
=maxlen))
word_embedding_model.add(Flatten())
word_embedding_model.add(Dense(32, activation='relu'))
word_embedding_model.add(Dense(1, activation='sigmoid'))
word_embedding_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 200, 100)          1500000
_________________________________________________________________
flatten_1 (Flatten)          (None, 20000)             0
_________________________________________________________________
dense_1 (Dense)              (None, 32)                640032
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 33
=================================================================
Total params: 2,140,065
Trainable params: 2,140,065
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 is the word vector meant to be associated with an index. We can load load the GloVe matrix we prepared into our Embedding layer, the first layer in our model.

Also, we freeze the embedding layer by setting its trainable attribute to False, because, the pre-trained parts should not be updated during training to avoid forgetting what the model already know.

In [7]:
```
word_embedding_model.layers[0].set_weights([word_embedding_matrix])
word_embedding_model.layers[0].trainable = False
```

Train and evaluate

Let's compile our model and train it:

In [8]:
```
word_embedding_model.compile(optimizer='rmsprop',
             loss='binary_crossentropy',
             metrics=['acc'])

history = word_embedding_model.fit(movie_data_train, movie_labels_train,
                   epochs=10,
                   batch_size=32,
                   validation_data=(movie_data_val, movie_labels_val))

word_embedding_model.save_weights('pre_trained_glove_model.h5')
```

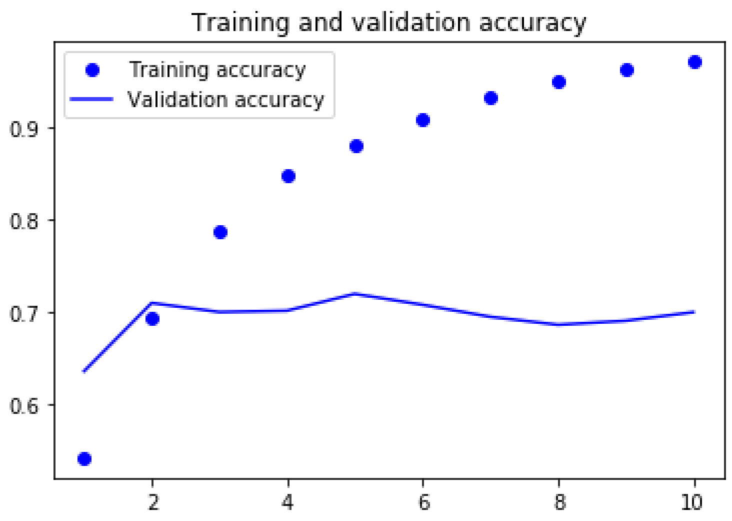
```
Train on 12500 samples, validate on 12500 samples
Epoch 1/10
12500/12500 [==============================] - 3s 209us/step - loss: 0.7365 -
acc: 0.5417 - val_loss: 0.6504 - val_acc: 0.6357
Epoch 2/10
12500/12500 [==============================] - 2s 189us/step - loss: 0.5870 -
acc: 0.6935 - val_loss: 0.5610 - val_acc: 0.7098
Epoch 3/10
12500/12500 [==============================] - 2s 189us/step - loss: 0.4639 -
acc: 0.7862 - val_loss: 0.5936 - val_acc: 0.6999
Epoch 4/10
12500/12500 [==============================] - 2s 189us/step - loss: 0.3643 -
acc: 0.8470 - val_loss: 0.6327 - val_acc: 0.7014
Epoch 5/10
12500/12500 [==============================] - 2s 188us/step - loss: 0.2897 -
acc: 0.8811 - val_loss: 0.6140 - val_acc: 0.7196
Epoch 6/10
12500/12500 [==============================] - 2s 189us/step - loss: 0.2313 -
acc: 0.9086 - val_loss: 0.7310 - val_acc: 0.7078
Epoch 7/10
12500/12500 [==============================] - 2s 189us/step - loss: 0.1801 -
acc: 0.9337 - val_loss: 0.8626 - val_acc: 0.6947
Epoch 8/10
12500/12500 [==============================] - 2s 191us/step - loss: 0.1350 -
acc: 0.9502 - val_loss: 0.9268 - val_acc: 0.6861
Epoch 9/10
12500/12500 [==============================] - 2s 189us/step - loss: 0.1064 -
acc: 0.9638 - val_loss: 1.0378 - val_acc: 0.6904
Epoch 10/10
12500/12500 [==============================] - 2s 190us/step - loss: 0.0828 -
acc: 0.9713 - val_loss: 1.0301 - val_acc: 0.6998
```

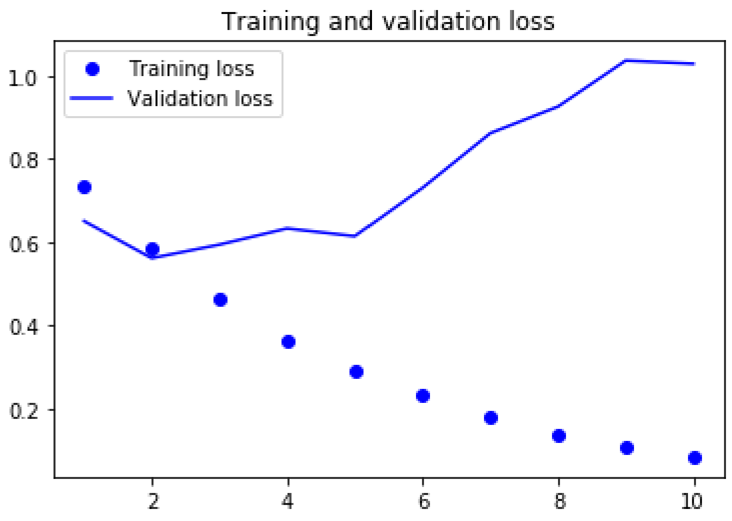
Let's plot its performance over time:

Plotting the results

In [9]:
```
# this is to render graphs in the notebook
%matplotlib inline
```

In [10]:
```
import matplotlib.pyplot as plt

accuracy = history.history['acc']
validation_accuracy = history.history['val_acc']
loss = history.history['loss']
validation_loss = history.history['val_loss']

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

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

plt.figure()

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

plt.show()
```





The model quickly starts overfitting. Validation accuracy has high variance for the same reason, but seems to reach high 70s.

We can also try to train the same model without loading the pre-trained word embeddings and without freezing the embedding layer.

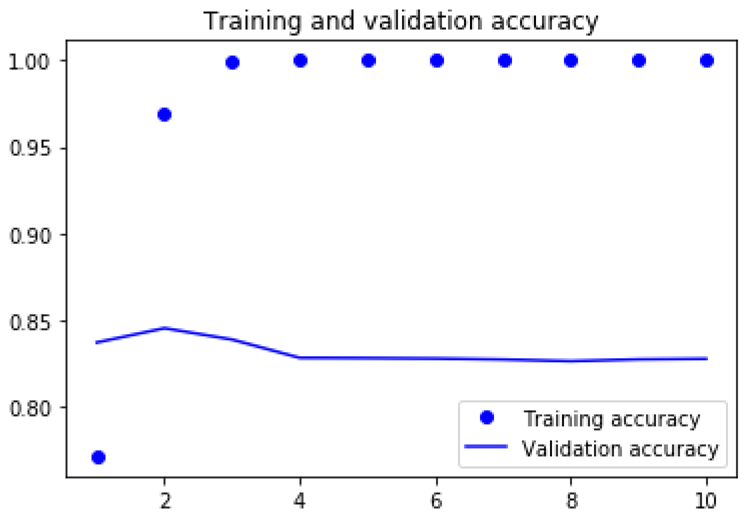
Training the same model without pretrained word embeddings

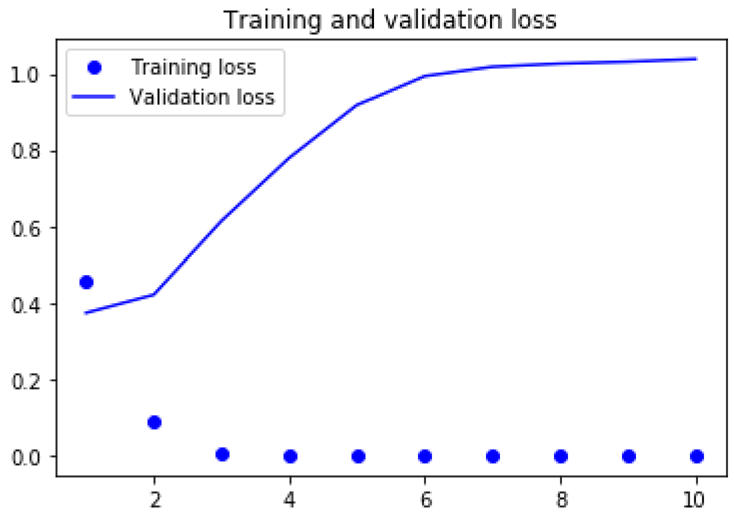
In [11]:
```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

no_pretrained_model = Sequential()
no_pretrained_model.add(Embedding(max_words,
                                  word_embedding_dim, input_length=maxlen))
no_pretrained_model.add(Flatten())
no_pretrained_model.add(Dense(32, activation='relu'))
no_pretrained_model.add(Dense(1, activation='sigmoid'))
no_pretrained_model.summary()

no_pretrained_model.compile(optimizer='rmsprop',
             loss='binary_crossentropy',
             metrics=['acc'])

history = no_pretrained_model.fit(movie_data_train, movie_labels_train,
                   epochs=10,
                   batch_size=32,
                   validation_data=(movie_data_val, movie_labels_val))
no_pretrained_model.save_weights('no_pretrained_model.h5')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_2 (Embedding)      (None, 200, 100)          1500000   
_________________________________________________________________
flatten_2 (Flatten)          (None, 20000)             0         
_________________________________________________________________
dense_3 (Dense)              (None, 32)                640032    
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 33        
=================================================================
Total params: 2,140,065
Trainable params: 2,140,065
Non-trainable params: 0
_________________________________________________________________
Train on 12500 samples, validate on 12500 samples
Epoch 1/10
12500/12500 [==============================] - 4s 352us/step - loss: 0.4557 -
acc: 0.7716 - val_loss: 0.3737 - val_acc: 0.8373
Epoch 2/10
12500/12500 [==============================] - 4s 324us/step - loss: 0.0910 -
acc: 0.9688 - val_loss: 0.4212 - val_acc: 0.8455
Epoch 3/10
12500/12500 [==============================] - 4s 326us/step - loss: 0.0055 -
acc: 0.9989 - val_loss: 0.6136 - val_acc: 0.8390
Epoch 4/10
12500/12500 [==============================] - 4s 325us/step - loss: 4.0626e-
04 - acc: 0.9998 - val_loss: 0.7788 - val_acc: 0.8284
Epoch 5/10
12500/12500 [==============================] - 4s 324us/step - loss: 1.1024e-
05 - acc: 1.0000 - val_loss: 0.9176 - val_acc: 0.8282
Epoch 6/10
12500/12500 [==============================] - 4s 325us/step - loss: 3.6742e-
07 - acc: 1.0000 - val_loss: 0.9938 - val_acc: 0.8280
Epoch 7/10
12500/12500 [==============================] - 4s 326us/step - loss: 1.3510e-
07 - acc: 1.0000 - val_loss: 1.0183 - val_acc: 0.8274
Epoch 8/10
12500/12500 [==============================] - 4s 326us/step - loss: 1.1051e-
07 - acc: 1.0000 - val_loss: 1.0266 - val_acc: 0.8265
Epoch 9/10
12500/12500 [==============================] - 4s 326us/step - loss: 1.1019e-
07 - acc: 1.0000 - val_loss: 1.0315 - val_acc: 0.8275
Epoch 10/10
12500/12500 [==============================] - 4s 325us/step - loss: 1.0970e-
07 - acc: 1.0000 - val_loss: 1.0386 - val_acc: 0.8278
```

Plotting the results

In [12]:
```
accuracy = history.history['acc']
validation_accuracy = history.history['val_acc']
loss = history.history['loss']
validation_loss = history.history['val_loss']

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

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

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, validation_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 we increase the number of training samples, this will quickly stop being the case.

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

Tokenizing the data of the test set

In [13]:
```
movie_test_dir = os.path.join(movies_dir, 'test')

movie_labels = []
movie_texts = []

for label_type in ['neg', 'pos']:
    movie_dir_name = os.path.join(movie_test_dir, label_type)
    for fname in sorted(os.listdir(movie_dir_name)):
        if fname[-4:] == '.txt':
            movie_file = open(os.path.join(movie_dir_name, fname))
            movie_texts.append(movie_file.read())
            movie_file.close()
            if label_type == 'neg':
                movie_labels.append(0)
            else:
                movie_labels.append(1)

sequences = tokenizer.texts_to_sequences(movie_texts)
movie_data_test = pad_sequences(sequences, maxlen=maxlen)
movie_labels_test = np.asarray(movie_labels)
```

And let's load and evaluate the first model:

Evaluating the model on the test set

In [14]:
```
word_embedding_model.load_weights('pre_trained_glove_model.h5')
word_embedding_model.evaluate(movie_data_test, movie_labels_test)
```

```
25000/25000 [==============================] - 1s 57us/step
```

Out[14]:
```
[1.0236201507616043, 0.70368]
```

We get a not so good test accuracy of 70%.