# Project: Sentiment Analysis Using Deep Learning

# RNN - SimpleRNN

### Working with text dataset

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'
```

### 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

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 = 125000

# 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,)
```

## A first recurrent layer - SimpleRNN

In [4]:
```python
from keras.layers import SimpleRNN
```

SimpleRNN processes batches of sequences. This means that it takes inputs of shape (batch_size, timesteps, input_features), rather than (timesteps, input_features). SimpleRNN can be run in two different modes: it can return either the full sequences of successive outputs for each timestep, or it can return only the last output for each input sequence. These two modes are controlled by the return_sequences constructor argument.

In [5]:
```python
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN

simple_rnn_model = Sequential()
simple_rnn_model.add(Embedding(20000, 32))
simple_rnn_model.add(SimpleRNN(32))
simple_rnn_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, None, 32)          640000
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 32)                2080
=================================================================
Total params: 642,080
Trainable params: 642,080
Non-trainable params: 0
_________________________________________________________________
```

In [6]:
```python
simple_rnn_model = Sequential()
simple_rnn_model.add(Embedding(20000, 32))
simple_rnn_model.add(SimpleRNN(32, return_sequences=True))
simple_rnn_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, None, 32)          640000
_________________________________________________________________
simple_rnn_2 (SimpleRNN)     (None, None, 32)          2080
=================================================================
Total params: 642,080
Trainable params: 642,080
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 [7]:
```
simple_rnn_model = Sequential()
simple_rnn_model.add(Embedding(20000, 32))
simple_rnn_model.add(SimpleRNN(32, return_sequences=True))
simple_rnn_model.add(SimpleRNN(32, return_sequences=True))
simple_rnn_model.add(SimpleRNN(32, return_sequences=True))
simple_rnn_model.add(SimpleRNN(32))  # This last layer only returns the last o
utputs.
simple_rnn_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_3 (Embedding)      (None, None, 32)          640000    
_________________________________________________________________
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: 648,320
Trainable params: 648,320
Non-trainable params: 0
_________________________________________________________________
```

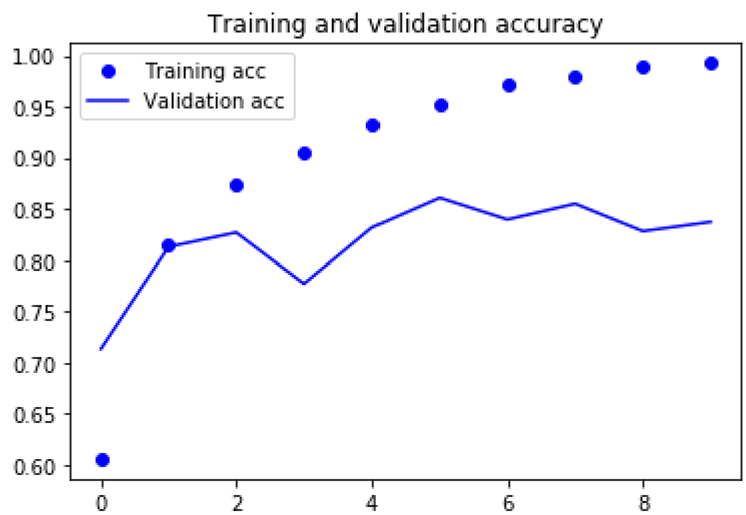
Now let's try to use such a model on the movie review sentiment analysis problem. Let's train a simple recurrent network using an Embedding layer and a SimpleRNN layer:

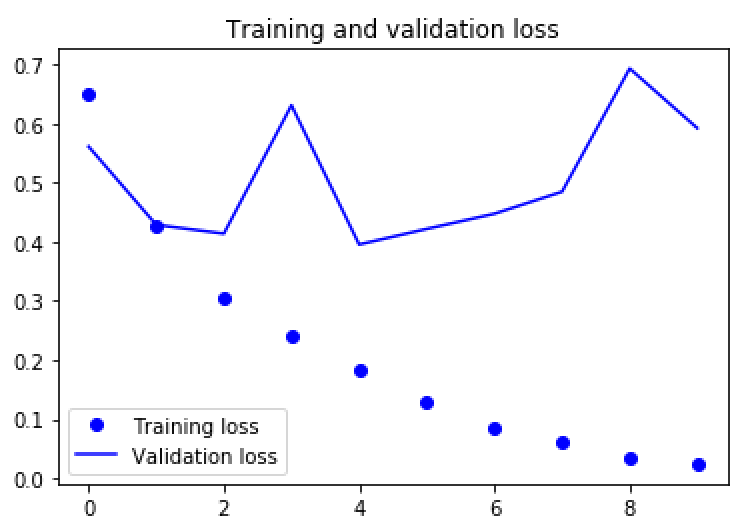
In [8]:
```
from keras.layers import Dense

simple_rnn_model = Sequential()
simple_rnn_model.add(Embedding(max_words, 32))
simple_rnn_model.add(SimpleRNN(32))
simple_rnn_model.add(Dense(1, activation='sigmoid'))

simple_rnn_model.compile(optimizer='rmsprop', loss='binary_crossentropy', metr
ics=['acc'])
history = simple_rnn_model.fit(movie_data_train, movie_labels_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
simple_rnn_model.save_weights('simple_rnn_model.h5')
```

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 7s 337us/step - loss: 0.6483 -
acc: 0.6055 - val_loss: 0.5611 - val_acc: 0.7132
Epoch 2/10
20000/20000 [==============================] - 6s 318us/step - loss: 0.4282 -
acc: 0.8143 - val_loss: 0.4287 - val_acc: 0.8130
Epoch 3/10
20000/20000 [==============================] - 6s 316us/step - loss: 0.3049 -
acc: 0.8740 - val_loss: 0.4138 - val_acc: 0.8270
Epoch 4/10
20000/20000 [==============================] - 6s 314us/step - loss: 0.2413 -
acc: 0.9050 - val_loss: 0.6309 - val_acc: 0.7768
Epoch 5/10
20000/20000 [==============================] - 6s 313us/step - loss: 0.1830 -
acc: 0.9325 - val_loss: 0.3954 - val_acc: 0.8318
Epoch 6/10
20000/20000 [==============================] - 6s 316us/step - loss: 0.1302 -
acc: 0.9527 - val_loss: 0.4217 - val_acc: 0.8608
Epoch 7/10
20000/20000 [==============================] - 6s 316us/step - loss: 0.0838 -
acc: 0.9713 - val_loss: 0.4474 - val_acc: 0.8396
Epoch 8/10
20000/20000 [==============================] - 6s 315us/step - loss: 0.0601 -
acc: 0.9805 - val_loss: 0.4846 - val_acc: 0.8550
Epoch 9/10
20000/20000 [==============================] - 6s 316us/step - loss: 0.0359 -
acc: 0.9892 - val_loss: 0.6930 - val_acc: 0.8282
Epoch 10/10
20000/20000 [==============================] - 6s 317us/step - loss: 0.0246 -
acc: 0.9929 - val_loss: 0.5921 - val_acc: 0.8372
```

In [9]:
```
%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(len(accuracy))

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

plt.show()
```



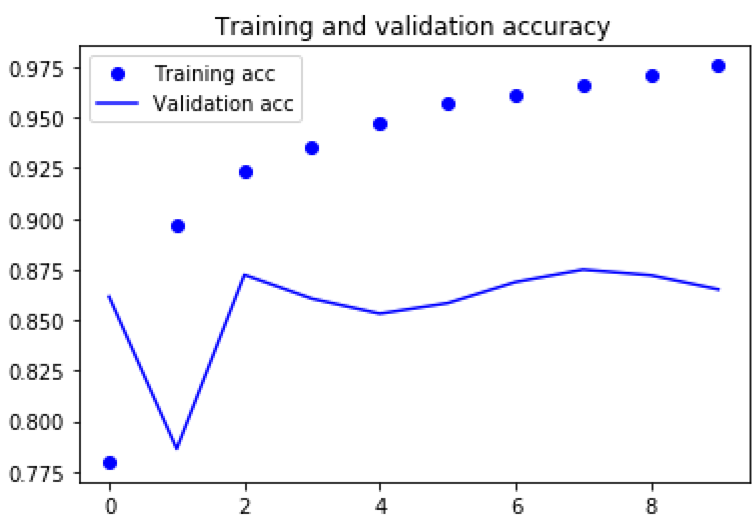
# Using Long-short memory (LSTM)

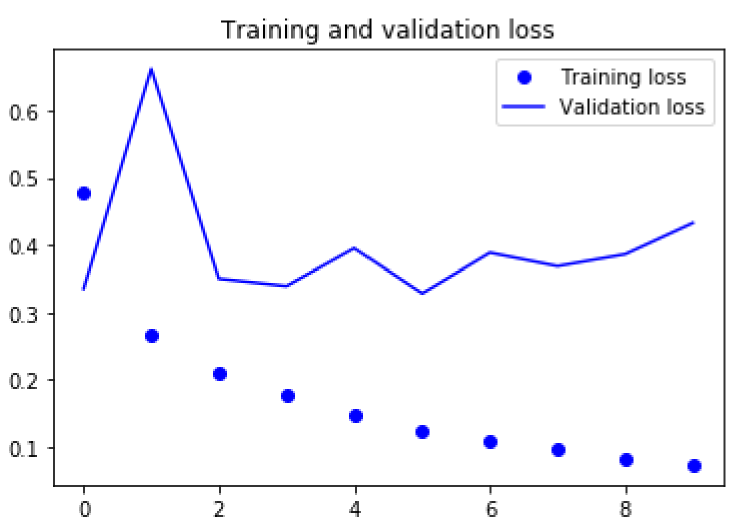
We will set up a model using a LSTM layer and train it on the movie data. The network is similar to the one with SimpleRNN. We only specify the output dimensionality of the LSTM layer, and use the default arguments.

In [11]:
```
from keras.layers import LSTM

lstm_model = Sequential()
lstm_model.add(Embedding(max_words, 32))
lstm_model.add(LSTM(32))
lstm_model.add(Dense(1, activation='sigmoid'))

lstm_model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = lstm_model.fit(movie_data_train, movie_labels_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
lstm_model.save_weights('lstm_model.h5')
```

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 19s 953us/step - loss: 0.4780
- acc: 0.7798 - val_loss: 0.3354 - val_acc: 0.8616
Epoch 2/10
20000/20000 [==============================] - 18s 915us/step - loss: 0.2677
- acc: 0.8968 - val_loss: 0.6618 - val_acc: 0.7862
Epoch 3/10
20000/20000 [==============================] - 18s 904us/step - loss: 0.2094
- acc: 0.9239 - val_loss: 0.3503 - val_acc: 0.8724
Epoch 4/10
20000/20000 [==============================] - 18s 892us/step - loss: 0.1773
- acc: 0.9355 - val_loss: 0.3394 - val_acc: 0.8606
Epoch 5/10
20000/20000 [==============================] - 18s 880us/step - loss: 0.1482
- acc: 0.9473 - val_loss: 0.3954 - val_acc: 0.8532
Epoch 6/10
20000/20000 [==============================] - 17s 875us/step - loss: 0.1247
- acc: 0.9568 - val_loss: 0.3283 - val_acc: 0.8584
Epoch 7/10
20000/20000 [==============================] - 17s 871us/step - loss: 0.1103
- acc: 0.9611 - val_loss: 0.3888 - val_acc: 0.8688
Epoch 8/10
20000/20000 [==============================] - 17s 868us/step - loss: 0.0960
- acc: 0.9664 - val_loss: 0.3689 - val_acc: 0.8750
Epoch 9/10
20000/20000 [==============================] - 17s 862us/step - loss: 0.0836
- acc: 0.9709 - val_loss: 0.3865 - val_acc: 0.8722
Epoch 10/10
20000/20000 [==============================] - 17s 873us/step - loss: 0.0737
- acc: 0.9758 - val_loss: 0.4328 - val_acc: 0.8652
```

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

epochs = range(len(accuracy))

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

plt.show()
```



## Reversed-order LSTM

In [13]:
```
from keras.preprocessing import sequence
from keras import layers
from keras.models import Sequential

# Reverse sequences
movie_data_train = [x[::-1] for x in movie_data_train]
movie_data_val = [x[::-1] for x in movie_data_val]

# Pad sequences
movie_data_train = sequence.pad_sequences(movie_data_train, maxlen=maxlen)
movie_data_val = sequence.pad_sequences(movie_data_val, maxlen=maxlen)

reverse_order_lstm_model = Sequential()
reverse_order_lstm_model.add(layers.Embedding(max_words, 128))
reverse_order_lstm_model.add(layers.LSTM(32))
reverse_order_lstm_model.add(layers.Dense(1, activation='sigmoid'))

reverse_order_lstm_model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = reverse_order_lstm_model.fit(movie_data_train, movie_labels_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
reverse_order_lstm_model.save_weights('reverse_order_lstm_model.h5')
```

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 23s 1ms/step - loss: 0.6264 -
acc: 0.6336 - val_loss: 0.5758 - val_acc: 0.7408
Epoch 2/10
20000/20000 [==============================] - 23s 1ms/step - loss: 0.4537 -
acc: 0.8256 - val_loss: 0.3891 - val_acc: 0.8634
Epoch 3/10
20000/20000 [==============================] - 23s 1ms/step - loss: 0.3555 -
acc: 0.8788 - val_loss: 0.3887 - val_acc: 0.8556
Epoch 4/10
20000/20000 [==============================] - 23s 1ms/step - loss: 0.2993 -
acc: 0.9002 - val_loss: 0.3808 - val_acc: 0.8694
Epoch 5/10
20000/20000 [==============================] - 23s 1ms/step - loss: 0.2553 -
acc: 0.9172 - val_loss: 0.4009 - val_acc: 0.8506
Epoch 6/10
20000/20000 [==============================] - 23s 1ms/step - loss: 0.2172 -
acc: 0.9312 - val_loss: 0.5023 - val_acc: 0.7786
Epoch 7/10
20000/20000 [==============================] - 23s 1ms/step - loss: 0.2096 -
acc: 0.9342 - val_loss: 0.3432 - val_acc: 0.8712
Epoch 8/10
20000/20000 [==============================] - 23s 1ms/step - loss: 0.1944 -
acc: 0.9405 - val_loss: 0.3803 - val_acc: 0.8740
Epoch 9/10
20000/20000 [==============================] - 23s 1ms/step - loss: 0.1738 -
acc: 0.9466 - val_loss: 0.3902 - val_acc: 0.8688
Epoch 10/10
20000/20000 [==============================] - 23s 1ms/step - loss: 0.1591 -
acc: 0.9526 - val_loss: 0.4293 - val_acc: 0.8654
```

## Bidirectional layer

In [14]:
```
from keras import backend as K
K.clear_session()
```

In [15]:
```
bidirectional_model = Sequential()
bidirectional_model.add(layers.Embedding(max_words, 32))
bidirectional_model.add(layers.Bidirectional(layers.LSTM(32)))
bidirectional_model.add(layers.Dense(1, activation='sigmoid'))

bidirectional_model.compile(optimizer='rmsprop',
                            loss='binary_crossentropy', metrics=['acc'])
history = bidirectional_model.fit(movie_data_train, movie_labels_train,
                    epochs=10, batch_size=128, validation_split=0.2)

bidirectional_model.save_weights('bidirectional_model.h5')
```

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 23s 1ms/step - loss: 0.5092 -
acc: 0.7523 - val_loss: 0.4005 - val_acc: 0.8222
Epoch 2/10
20000/20000 [==============================] - 21s 1ms/step - loss: 0.2857 -
acc: 0.8869 - val_loss: 0.4412 - val_acc: 0.8402
Epoch 3/10
20000/20000 [==============================] - 21s 1ms/step - loss: 0.2189 -
acc: 0.9193 - val_loss: 0.4087 - val_acc: 0.8422
Epoch 4/10
20000/20000 [==============================] - 21s 1ms/step - loss: 0.1800 -
acc: 0.9354 - val_loss: 0.3332 - val_acc: 0.8588
Epoch 5/10
20000/20000 [==============================] - 21s 1ms/step - loss: 0.1540 -
acc: 0.9451 - val_loss: 0.5021 - val_acc: 0.8162
Epoch 6/10
20000/20000 [==============================] - 21s 1ms/step - loss: 0.1392 -
acc: 0.9515 - val_loss: 0.5105 - val_acc: 0.8028
Epoch 7/10
20000/20000 [==============================] - 21s 1ms/step - loss: 0.1186 -
acc: 0.9573 - val_loss: 0.3593 - val_acc: 0.8416
Epoch 8/10
20000/20000 [==============================] - 21s 1ms/step - loss: 0.1037 -
acc: 0.9643 - val_loss: 0.3997 - val_acc: 0.8696
Epoch 9/10
20000/20000 [==============================] - 21s 1ms/step - loss: 0.0911 -
acc: 0.9694 - val_loss: 0.4554 - val_acc: 0.8648
Epoch 10/10
20000/20000 [==============================] - 21s 1ms/step - loss: 0.0794 -
acc: 0.9736 - val_loss: 0.4148 - val_acc: 0.8584
```

It performs slightly better than the regular LSTM we tried in the previous section, going above ~86% validation accuracy.