Introduction to Machine Learning

Project Report

Sentiment Analysis on Yelp data with Neural Networks

By: Amir Suhail (1410110043)

Introduction

Sentiment analysis is an important application of natural language processing, as it makes it possible to predict what a person thinks given the text she has written. The yelp dataset contains over 6 million text reviews from users on businesses, as well as their rating. This dataset is interesting because it is large enough to train advanced machine learning models. It is also large enough to be fairly challenging to process.

Index

Part 1: Gathering System Requirements

Before doing any large scale data analysis, you need to know how much resources are available on your computer. The resources we care about are:

- The amount of RAM (Random Access Memory). The data stored in the RAM is accessed directly by the processor.
- The amount of disk space . Disk space is necessary to store your samples.
- The number of cores in the processor . The more cores you have, the more processes you'll be able to run simultaneously

The laptop I’m working on has 4 cores and 8GB of RAM with a 2 GB Nvidia GT740 GPU.

2. Text Preprocessing

We are going to convert the yelp text data to a numpy array that can fit in memory and that is suitable to machine learning.
We ultimately want to train our machine to predict whether the review is positive or negative given only the text, but we have two problems to solve:

> To feed the review text to a neural network, for example, we need to convert it to an array of numbers in some way (encoding) ;
> Loading the full dataset in memory is barely doable on a computer with 16 GB of RAM.

The important things we are focusing on are:

- organizing a python package in a modular way.
- Considerations in algorithmic complexity.
- Efficient memory management.
- Multiprocessing.

Our dataset is in a 5 GB file JSON Lines file, review.json, which contains lines like:

{"review_id":"Q1sbwvVQXV2734tPgoKj4Q","user_id":"hG7b0MtEbXx5QzbzE6C_VA","business_id":"ujmEBvifdJM6h6RLv4wQIg","stars":1.0,"useful":6,"funny":1,"cool":0,"text":"Total bill for this horrible service? Over $8Gs. These crooks actually had the nerve to charge us $69 for 3 pills. I checked online the pills can be had for 19 cents EACH! Avoid Hospital ERs at all costs.","date":"2013-05-07 04:34:36"}

Processing this data is going to be a bit challenging, both in terms of RAM and CPU.

However, given the size of this JSON file, I was confident that it would be possible to find a way to transform this data into a structure that would entirely fit in memory.

2.1 Divide and Conquer

To do that, I decided to adopt a tiered processing approach. In this approach, we will process the data in subsequent steps. For each review, we will take the following steps:

- extract the words from the review text
- encode words into integers
- convert the lists of integers to a numpy array suitable to machine learning

The first two steps are CPU intensive, and the last one needs a lot of RAM.

CPU intensive tasks can be performed in parallel on a multicore processor. On the other hand, tasks than require a lot of RAM may not be parallelized on a single machine, since all cores share the same RAM. For these reasons, I decided to isolate CPU- and RAM-intensive tasks in different processes.

Now, to be able to perform CPU-intensive tasks in parallel on different cores, we need to split our sample into chunks, so that each core can take care of a chunk. (we have four cores, so we will make 4 chunks of 16671475 lines each).

split -l 340000 review.json

We can check the contents of any one of them:

>>head -n 1 xab

{"review_id":"srnRzrX0sWEigqfyV_3BVQ","user_id":"4eT43qWNh-9Xdy0_TTU1qw","business_id":"9mCX2MZvZP9KgnOUCVod0Q","stars":4.0,"useful":0,"funny":0,"cool":0,"text":"Came within 24 hours of request. Came at the scheduled time. Quickly loaded the items. Polite. A bit expensive, but they provide a useful service. I would call them again.","date":"2016-04-18 00:43:51"

2.2 Text parsing with nltk

In this section, we will process our chunks to extract words from the review text, which is a string. For that, we will use the nltk package. But first, let's try and do it in bare python. To perform the tokenization (the task of extracting the words), we will instead use the nltk package. The module, yelp_tokenize.py, as well as the supporting modules parallelize and base, can be found inside the repository of the project.

An interesting thing to note: there are two tasks that are going to be done in several scripts: parallelization to several cores, and the definition of some of the command line arguments. Indeed, these scripts have the following in common:

- they read several input files from a directory
- they perform some task on each file, possibly returning results

That's why I decided to create a unified interface for the parallelization, and to define the command line arguments that are in common at a single place. I did that in the parallelize and base modules, respectively.

python yelp_tokenize.py -d <path_to_our_yelp_dataset> 'xa?' -p

We can check the content of this newly created file:

>> head -n 1 xaa_tok.json

{"review_id": "Q1sbwvVQXV2734tPgoKj4Q", "user_id": "hG7b0MtEbXx5QzbzE6C_VA", "business_id": "ujmEBvifdJM6h6RLv4wQIg", "stars": 1.0, "useful": 6, "funny": 1, "cool": 0, "text": ["total", "bill", "for", "this", "horrible", "service", "?", "over", "$", "8gs", ".", "these", "crooks", "actually", "had", "the", "nerve", "to", "charge", "us", "$", "69", "for", "3", "pills", ".", "i", "checked", "online", "the", "pills", "can", "be", "had", "for", "19", "cents", "each", "!", "avoid", "hospital", "ers", "at", "all", "costs", "."], "date": "2013-05-07 04:34:36"}

As we see, all review texts have been split into words.

2.3 Building the vocabulary

We need to build a vocabulary before encoding, which is an ordered list of all possible words in all reviews. Then, a word can be encoded as a number giving its position in the vocabulary.

In the code I used for **yelp_vocabulary.py**, we have a separate vocabulary class which we import

- the index is a dictionary because:
    - we want to insert a word in the vocabulary only if it's not already there. So we need to search the index at each word during the construction of the vocabulary, and we need to do that fast.
    - after the vocabulary is built for the whole yelp dataset, we will encode all reviews. For that, we will look up the integer corresponding to each word in the index. That's a massive number of searches, so it must be fast.
- the vocabulary is a list because:
    - it must be ordered, since the position in the vocabulary represents a word.
    - at decoding time, it's good to have O(1) complexity, though we're not going to decode much in practice.

The reviews that are too long will be truncated, and the ones that are too small will be padded with 0.

Then we ran the script to build the index in parallel:

python yelp_vocabulary.py -d <your_data_dir> 'xa?_tok.json' -p

We will get a file called index.pck in our data directory. It contains the vocabulary object.

We're now ready to encode the whole yelp dataset using our vocabulary.

2.4 Encoding the yelp dataset

We will process a review JSON lines file and count the words in all reviews and return the counter, which will be used to find the most frequent words in our **yelp_encode.py** script.
Now we run it to encode the whole yelp dataset in parallel:

python yelp_encode.py -d <your_data_dir> 'xa?_tok.json' -p

We will get new files in our data directory, with a name ending with _enc.json. Further we can check the contents of a file:

>>head -n 1 xaa_enc.json

{"review_id": "Q1sbwvVQXV2734tPgoKj4Q", "user_id": "hG7b0MtEbXx5QzbzE6C_VA", "business_id": "ujmEBvifdJM6h6RLv4wQIg", "stars": 1.0, "useful": 6, "funny": 1, "cool": 0, "text": [805, 548, 1, 5, 528, 28, 65, 103, 55, 1, 1, 238, 8079, 264, 11, 1, 4895, 1, 537, 62, 55, 10651, 1, 182, 9011, 1, 1, 737, 753, 1, 9011, 56, 1, 11, 1, 3754, 2964, 276, 2, 978, 1865, 1, 14, 24, 1767, 1], "date": "2013-05-07 04:34:36"}

Note that the unknown words encoded by a 1. There is a good fraction of them, but most often, they are simply stop words, like "the".

2.5 Converting the encoded dataset to a numpy array

- A numpy array is represented in memory by a contiguous section of memory.It is good to consider that an array has a fixed size along all its axes. Expanding an array is in fact possible, but this operation might involve a copy of the array data to a new area of memory to find enough space to store the data contiguously. I actually never do that.

- In our case, we need to have an array with two axes, like an excel table. The first axis (rows) will index the examples (the reviews) and the second axis (columns) will contain the rating (stars) followed by all encoded words in the review text.

- Now, we know that the number of lines is fixed to the number of lines in our input JSON files. But the review text currently has a variable length: some reviews have a lot of words, and some others only a few.

- To deal with this issue, we will impose a maximum review size nwords. Reviews with more than nwords will be truncated , meaning that the last words in the review will just be dropped. If nwords is large enough, that's probably not a big issue since the user had enough space to give her opinion already. Reviews with less than nwords will be padded : all remaining slots on this line of the array will simply be filled with a 0.

- Why 0 and not another number? Simply because the neural network will not see this value. Indeed, remember that a neural network is simply a function of its input values . If one of the values is 0, it will not flow through the network. It is the same as setting all weights multiplying this value to 0.

So we want a numpy array containing the whole yelp review data, that fits entirely in memory. Let's estimate the size of this array.

- we have about 7 million reviews, so the same number of lines
- we could limit the size of each review to 500 words, and we need a few more

columns to store the number of stars and other features. Still, that's about 500 columns
- each number takes size in memory, and we need to know how much. To find out, let's do a simple test

By default, every number in a numpy array is a 64-bits float, which takes 8 bytes of memory (since 1 byte = 8 bits). Then, the size of our numpy array in memory will be 7e6 * 500 * 8 = 28 GB!!! Clearly, this is too large for the RAM of the vast majority of computers. And that's anyway not reasonable to use so much RAM if there are solutions to use less. Let's see what we can do.

First, we want to store integers, not floats. If we code an integer on 64 bits, we can have values of up to 2 at the power 64 = 18446744073709551616. We certainly do not need that: the star rating goes up to 5, and the encoded words go up to the vocabulary size (~20,000 by default). With 8 bits, we can have 256 different integer values, and with 16 bits, 65536. So 16 bits is the right size. With respect to 64 bits, we gain a factor 4.

Apart from a small constant overhead of 96 bytes, we do gain a factor 4. For the whole yelp dataset, we would go down to 28 / 4 = 7 GB. That's still a bit too much.To gain another factor of two and go down to 3.5 GB, we are just going to limit the review size to 250 words. And now we can talk.

Eventually we have **yelp_fillarray.py** that converts the encoded dataset to a numpy array. Here, we don't run it in parallel mode because this step requires quite a bit of RAM.

python yelp_fillarray.py -d <your_data_dir> 'xa*_enc.json'

This script creates a file called data.h5 in our data directory, with a size of 3.2 GB, as estimated.

3. <u>**Exploring the data using text mining**</u>

When we open the newly created h5py file, we will have the data as expected:

 6685900 reviews, and for each of them, 254 values:

- rating
- useful
- funny
- cool
- 250 encoded words.

We can have a look at the first entry as given below. It is a positive one (5 stars in the first column) and one person found it useful. The review text is relatively short, and followed by the padding zero values.

```
In [4]: print(d[0])

[    5     1     0     0   696    26    39  3348    26  1523    44    3
    64    14   153  5179  2731    24    72   172  4377   125   257   36
  6568 10127  8410     3    33   277   219   501  8900     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0]
```

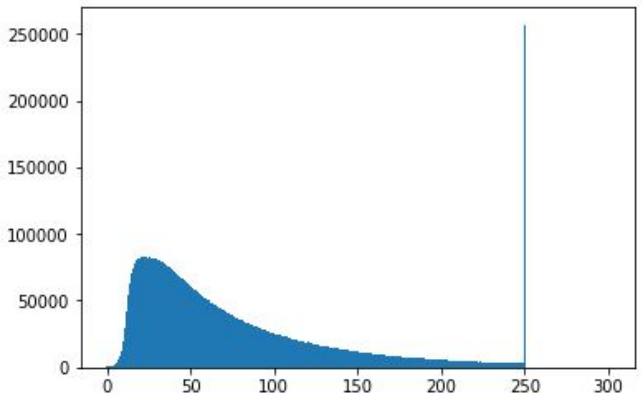
To decode this review, we are going to use our Vocabulary class.

In [5]:
```
from vocabulary import Vocabulary
vocab = Vocabulary.load(datadir+'/index')
rev1 = d[0,1:]
rev1 = rev1[rev1!=0]
' '.join(vocab.decode(rev1))
```

Out[5]:
```
'<UNK> helped out when locked out apartment he quick got at price lowe
```

Let's plot the distribution of words in the reviews:

In [7]:
```
reviews = d[:,4:]
word_counts = np.count_nonzero(reviews, axis=1)
# the assignment to _ avoids a long useless printout
_ = plt.hist(word_counts,range=(-0.5,300.5),bins=301)
```

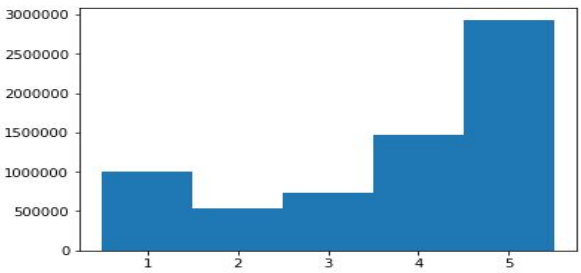


Since we have truncated our reviews at a maximum length of 250, all reviews which had more words end up in the last bin.

Now let's plot the rating distribution:

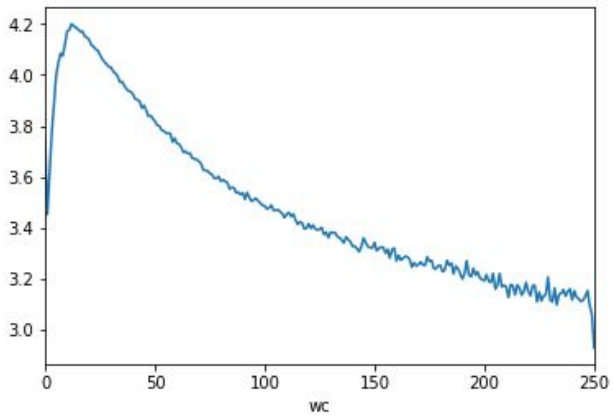
In [8]:
```
plt.hist(d[:,0], range=(0.5,5.5), bins=5)
```

Out[8]:
```
(array([1002159.,  542394.,  739280., 1468985., 2933082.]),
 array([0.5, 1.5, 2.5, 3.5, 4.5, 5.5]),
 <a list of 5 Patch objects>)
```



Let's see how the average rating evolves with the number of words:

```
In [75]: means['stars'].plot()
Out[75]: <matplotlib.axes._subplots.AxesSubplot at 0x7f0e5824c748>
```

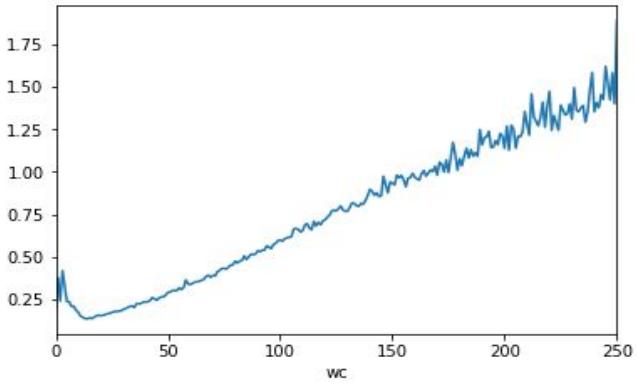


After a peak around 15 words, the average rating decreases steadily with the number of words. It looks like the more angry people are, the more they write.

Around 250, there is a sharp drop. This is simply due to the fact that we have truncated our ratings: over 250 words, the average rating keeps dropping steadily.

Now let's look at the average of the 'funny' score:

```
In [76]: means['funny'].plot()
Out[76]: <matplotlib.axes._subplots.AxesSubplot at 0x7f0e58204668>
```



Long reviews are more funny than the short ones. But actually, some short reviews are funny too, and I'm not too sure why.

4. Sentiment analysis using Neural Networks

Now let's open our dataset file. This is an hdf5 file, so we use the h5py package to open it.

```
In [3]: datadir = '/home/hades/Videos/yelp'
        h5 = h5py.File(datadir+'/data.h5')
        d = h5['reviews']
        d.shape
Out[3]: (6685900, 254)

In [4]: h5.keys()
Out[4]: <KeysViewHDF5 ['reviews']>

In [5]: data=h5['reviews']

In [6]: data.shape
Out[6]: (6685900, 254)
```

We can use the dataset already as a numpy array. h5py will load in memory only the data you need to complete a given operation. Let's check the first line:

```
In [6]: data.shape
Out[6]: (6685900, 254)

In [7]: data[0]
Out[7]: array([    5,     1,     0,     0,   696,    26,    39,  3348,    26,
                1523,    44,   336,    64,    14,   153,  5179,  2731,    24,
                  72,   172,  4377,   125,   257,  3044,  6568, 10127,  8410,
                   3,    33,   277,   219,   501,  8900,     0,     0,     0,
                   0,     0,     0,     0,     0,     0,     0,     0,     0,
                   0,     0,     0,     0,     0,     0,     0,     0,     0,
                   0,     0,     0,     0,     0,     0,     0,     0,     0,
                   0,     0,     0,     0,     0,     0,     0,     0,     0,
                   0,     0,     0,     0,     0,     0,     0,     0,     0,
                   0,     0,     0,     0,     0,     0,     0,     0,     0,
                   0,     0,     0,     0,     0,     0,     0,     0,     0,
                   0,     0,     0,     0,     0,     0,     0,     0,     0,
                   0,     0,     0,     0,     0,     0,     0,     0,     0,
                   0,     0,     0,     0,     0,     0,     0,     0,     0,
                   0,     0,     0,     0,     0,     0,     0,     0,     0,
                   0,     0,     0,     0,     0,     0,     0,     0,     0,
```

At preprocessing stage, when I created this array, I decided to reserve the first four slots on each line for:

- the number of stars;
- the number of "useful" votes;
- the number of "funny" votes;
- the number of "cool" votes.

The reviewer gave 5 stars (the maximum rating) to the company above, and somebody considered his review helpful.

After the first four slots come the codes for the review text. I allocated 250 slots for the reviews. If the review contains more than 250 words, it's truncated. If it contains less that 250 words, as is the case here, the unused slots are filled with zeros.

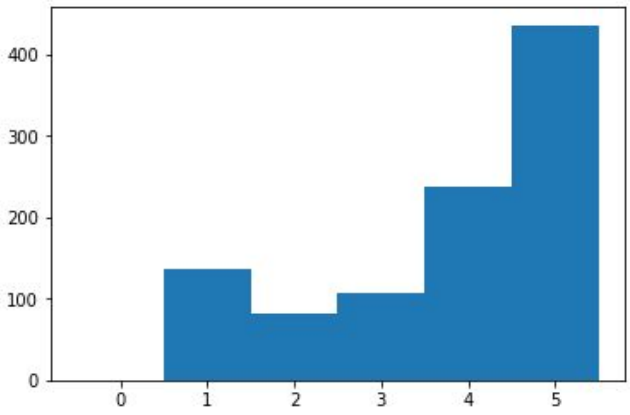
Let's extract the information needed to train our first neural network:

In [9]:
```
# the reviews
x = data[:, 4:]
# the stars, from which we will
# obtain the labels (see below)
stars = data[:,0]
# additional features we might consider:
useful = data[:,1]
cool = data[:,2]
funny = data[:,3]
```

Our goal is to predict whether the review text is positive or negative. Therefore, we need to label our examples in two categories: 0 (negative) and 1 (positive). We can use the number of stars to define these categories. For example, we could say that a review with 3 stars or more is positive.
First, let's check the distribution of stars:

In [11]:
```
plt.hist(stars[:1000], range=(-0.5, 5.5), bins=6)
```

Out[11]:
```
(array([  0., 137.,  81., 108., 238., 436.]),
 array([-0.5,  0.5,  1.5,  2.5,  3.5,  4.5,  5.5]),
 <a list of 6 Patch objects>)
```



We see that the number of stars ranges from 1 to 5, so it's not possible for a reviewer to give no star.

Then, we want to split the dataset in two categories that have roughly the same number of examples. If we were to define as positive examples with 3 stars or more, the positive category would be much larger than the negative one.I prefer to define as positive all reviews with 4 stars or more. Technically, here is how to define the targets:

In [12]:
```
# first fill an array with zeros,
# with the same shape as stars
y = np.zeros_like(stars)
# then write 1 if the number of stars is 4 or 5
y[stars>3.5] = 1
print(y, len(y))
print(stars, len(stars))
```

```
[1 0 1 ... 0 0 0] 6685900
[5 3 5 ... 1 1 3] 6685900
```

As usual, we split the dataset into a training and a test sample. At first, we will use 20000 examples for the test sample, and "only" 100,000 examples for the training sample:

In [13]:
```
n_test = 20000
n_train = 100000
x_test = x[:n_test]
y_test = y[:n_test]
x_train = x[n_test:n_train+n_test]
y_train = y[n_test:n_train+n_test]
```

4.1 Using Dense neural networks

Our first deep neural network will contain:

- An embedding layer,
- A dense layer, responsible for interpreting the results of the embedding,
- A final sigmoid neuron that will output the probability for the review to be positive.

We start by creating a new model:

```
In [11]: from keras.models import Sequential
         model = Sequential()
```

The first layer will be the embedding layer. Its role is to convert each integer representing a word into a vector in N-dimensional space. In this space, words with similar meaning will be grouped together.

```
In [15]: review_length = len(x_train[0])
         model.add(keras.layers.Embedding(len(vocab.words), 2,
                                          input_length=review_length))
```

The output of the embedding is multidimensional. Indeed, we start with a 1D array with 250 words. Since embedding gives us a two-dimensional vector for each word, the embedding layer spits out an array of shape (250, 2). This 2D array cannot be used directly as input to a dense layer, so we need to flatten it into a 1D array with 500 slots. This is done by the Flatten layer:

```
In [16]: model.add(keras.layers.Flatten())
```

Then, we add dropout regularization. In a nutshell, the dropout regularization layer drops, on a random basis, a fraction of its input values. This forces the network to learn different paths to solve the problem, and helps reduce overfitting. Here we decide to drop 40% of the values from the Flatten layer:

```
In [17]: model.add(keras.layers.Dropout(rate=0.4))
```

After that, we can add a dense layer, which will analyze the results of the embedding. Again, we start small, with only 5 neurons. We will see later if performance can be improved by increasing the number of neurons.

```
In [18]: model.add(keras.layers.Dense(5))
```

And finally, we end with a dense layer consisting of a single neuron with a sigmoid activation function . Therefore, this neuron will produce a value between 0 and 1, which is the estimated probability for the example review to be positive.

```
In [19]: model.add(keras.layers.Dense(1, activation='sigmoid'))
```

We can now compile and print the full model:

printing the full model. 1. Optimizer algo: Adam ; 2. loss function: cross-entropy (to maximize the likelihood of classifying the input data correctly) ' 3. accuracy function

```
In [17]: model.compile(optimizer='adam',
                       loss='binary_crossentropy',
                       metrics=['accuracy'])

         model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 250, 2)            40004
_________________________________________________________________
flatten_1 (Flatten)          (None, 500)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 2505
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 6
=================================================================
Total params: 42,515
Trainable params: 42,515
Non-trainable params: 0
_________________________________________________________________
```

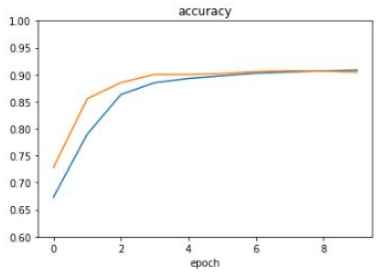
We fit the model on the training dataset:

In [18]:
```
history = model.fit(x_train,
                    y_train,
                    epochs=10,
                    batch_size=1000,
                    validation_data=(x_test, y_test),
                    verbose=1)
```

```
Train on 100000 samples, validate on 20000 samples
Epoch 1/10
100000/100000 [==============================] - 17s - loss: 0.6184 - acc: 0.6733 - val_loss: 0.5583 - val_acc: 0.7
282
Epoch 2/10
100000/100000 [==============================] - 2s - loss: 0.4580 - acc: 0.7897 - val_loss: 0.3562 - val_acc: 0.85
53
Epoch 3/10
100000/100000 [==============================] - 2s - loss: 0.3291 - acc: 0.8632 - val_loss: 0.2885 - val_acc: 0.88
54
Epoch 4/10
100000/100000 [==============================] - 2s - loss: 0.2863 - acc: 0.8852 - val_loss: 0.2646 - val_acc: 0.90
04
Epoch 5/10
100000/100000 [==============================] - 2s - loss: 0.2660 - acc: 0.8929 - val_loss: 0.2548 - val_acc: 0.90
02
Epoch 6/10
100000/100000 [==============================] - 2s - loss: 0.2555 - acc: 0.8977 - val_loss: 0.2500 - val_acc: 0.90
19
Epoch 7/10
100000/100000 [==============================] - 2s - loss: 0.2463 - acc: 0.9027 - val_loss: 0.2510 - val_acc: 0.90
56
Epoch 8/10
100000/100000 [==============================] - 2s - loss: 0.2401 - acc: 0.9049 - val_loss: 0.2463 - val_acc: 0.90
73
Epoch 9/10
100000/100000 [==============================] - 2s - loss: 0.2346 - acc: 0.9069 - val_loss: 0.2470 - val_acc: 0.90
66
Epoch 10/10
100000/100000 [==============================] - 2s - loss: 0.2301 - acc: 0.9084 - val_loss: 0.2477 - val_acc: 0.90
46
```

We see that we end up with a validation accuracy of about 90%. To have a look at the performance in more details, we will use the following function:

In [19]:
```
import matplotlib.pyplot as plt
def plot_accuracy(history, miny=None):
    '''Plot the training and validation accuracy'''
    acc = history.history['acc']
    test_acc = history.history['val_acc']
    epochs = range(len(acc))
    plt.plot(epochs, acc)
    plt.plot(epochs, test_acc)
    if miny:
        plt.ylim(miny, 1.0)
    plt.title('accuracy')
    plt.xlabel('epoch')
    plt.figure()

plot_accuracy(history, miny=0.6)
```



The training accuracy plateaus at 90%, so training further will not help much.

What we see here is that this network underfits the data, meaning that architecture is not complex enough to fit the data. By making it more complex, the training and testing accuracies can certainly be improved.

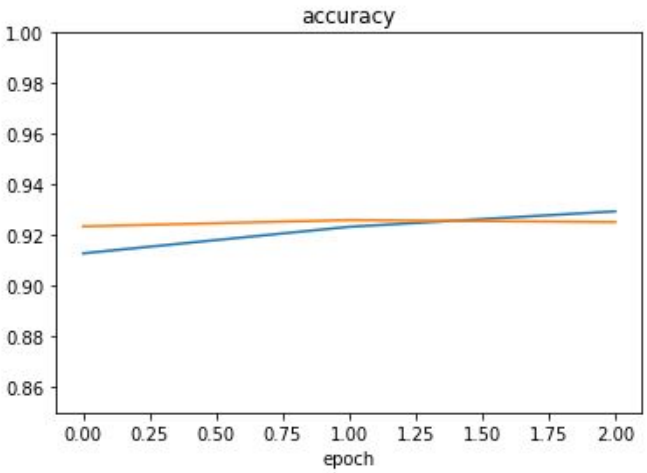
After some tuning, I converged to the following architecture. The structure of the network is complex, so I use the full dataset to avoid overfitting.

In [24]:
```
n_test = 20000
x_test = x[:n_test]
y_test = y[:n_test]
x_train = x[n_test:]
y_train = y[n_test:]

from keras.models import Sequential
model = Sequential()
model.add(keras.layers.Embedding(len(vocab.words), 128,
                                 input_length=review_length))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dropout(rate=0.4))
model.add(keras.layers.Dense(100, activation='relu'))
model.add(keras.layers.Dense(100, activation='relu'))
model.add(keras.layers.Dense(100, activation='relu'))
model.add(keras.layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.summary()
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
history = model.fit(x_train,
                    y_train,
                    epochs=3,
                    batch_size=1000,
                    validation_data=(x_test, y_test),
                    verbose=1)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 250, 128)          2560256
_________________________________________________________________
flatten_3 (Flatten)          (None, 32000)             0
_________________________________________________________________
dropout_3 (Dropout)          (None, 32000)             0
_________________________________________________________________
dense_7 (Dense)              (None, 100)               3200100
_________________________________________________________________
dense_8 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_9 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_10 (Dense)             (None, 1)                 101
=================================================================
Total params: 5,780,657
Trainable params: 5,780,657
Non-trainable params: 0
_________________________________________________________________
Train on 6665900 samples, validate on 20000 samples
Epoch 1/3
6665900/6665900 [==============================] - 9057s - loss: 0.2115 - acc: 0.9127 - val_loss: 0.1921 - val_acc:
0.9221
Epoch 2/3
6665900/6665900 [==============================] - 8627s - loss: 0.1886 - acc: 0.9230 - val_loss: 0.1835 - val_acc:
0.9253
Epoch 3/3
6665900/6665900 [==============================] - 8523s - loss: 0.1750 - acc: 0.9289 - val_loss: 0.1820 - val_acc:
0.9260
```

That's an improvement over the previous attempt, but the training is quite long, and it seems we will not be able to reach 92.6% accuracy on the test sample with this technique.

In [27]:
```
plot_accuracy(history,miny=0.85)
```



4.2 Using Convolution Neural Networks

In the first section, we will introduce a 1D convolutional layer in our network, with a kernel size of 3. At each step, the kernel moves by one word while scanning the next 3.

The convolutional layer will find it whatever its position in the sentence. Also, it will be easy for the network to understand the meaning of not good . On the contrary, in our previous attempt, not and good are not directly considered together.

Let's try. In the example below, the convolutional layer is set up with:

- a kernel size of 3,
- 64 filters. This means that 64 features (values) will be extracted from each position of the kernel,
- a ReLU activation, as usual.

In [13]:
```
n_test = 20000
n_train = 1000000
x_test = x[:n_test]
y_test = y[:n_test]
x_train = x[n_test:n_test+n_train]
y_train = y[n_test:n_test+n_train]

from keras.models import Sequential
model = Sequential()

model.add(keras.layers.Embedding(len(vocab.words), 64, input_length=250))
model.add(keras.layers.Conv1D(filters=64, kernel_size=3, activation='relu'))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dropout(rate=0.4))
model.add(keras.layers.Dense(50, activation='relu'))
model.add(keras.layers.Dense(1, activation='sigmoid'))

model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
history = model.fit(x_train,
                    y_train,
                    epochs=3,
                    batch_size=1000,
                    validation_data=(x_test, y_test),
                    verbose=1)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 250, 64)           1280128
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 248, 64)           12352
_________________________________________________________________
flatten_1 (Flatten)          (None, 15872)             0
_________________________________________________________________
dropout_1 (Dropout)          (None, 15872)             0
_________________________________________________________________
dense_1 (Dense)              (None, 50)                793650
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 51
=================================================================
Total params: 2,086,181
Trainable params: 2,086,181
Non-trainable params: 0
_________________________________________________________________
Train on 1000000 samples, validate on 20000 samples
Epoch 1/3
1000000/1000000 [==============================] - 1558s - loss: 0.2535 - acc: 0.8993 - val_loss: 0.2026 - val_acc:
0.9206
Epoch 2/3
1000000/1000000 [==============================] - 1470s - loss: 0.1947 - acc: 0.9235 - val_loss: 0.1951 - val_acc:
0.9243
Epoch 3/3
1000000/1000000 [==============================] - 1526s - loss: 0.1810 - acc: 0.9292 - val_loss: 0.1893 - val_acc:
0.9255
```

With the convolutional layer, we get almost same performance as with our best try with a simple dense network. However, please note that:

- there are only 2 million parameters in the network, instead of 10 million.
- the convolutional network is less subject to overfitting, and we could restrict the number of training examples to 1 million instead of 6.7 millions, and the training was much faster.
- there is room for optimization.

Stacked convolutional layers

In this section, we will optimize our convolutional network further by stacking convolutional layers.

We perform max pooling between each convolutional layer, and the layers extract more and more features as we progress in the network.

To avoid overfitting, we use the whole dataset for training except for 20000 events that are kept for testing.

In [31]:
```
n_test = 20000
x_test = x[:n_test]
y_test = y[:n_test]
x_train = x[n_test:]
y_train = y[n_test:]

from keras.models import Sequential
model = Sequential()

model.add(keras.layers.Embedding(len(vocab.words), 64, input_length=250))

model.add(keras.layers.Conv1D(filters=16, kernel_size=3, padding='same', activation='relu'))
model.add(keras.layers.MaxPooling1D(pool_size=2))
model.add(keras.layers.Conv1D(filters=32, kernel_size=3, padding='same', activation='relu'))
model.add(keras.layers.MaxPooling1D(pool_size=2))
model.add(keras.layers.Conv1D(filters=64, kernel_size=3, padding='same', activation='relu'))

model.add(keras.layers.Flatten())
model.add(keras.layers.Dropout(rate=0.5))

model.add(keras.layers.Dense(100, activation='relu'))

model.add(keras.layers.Dense(1, activation='sigmoid'))

model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

history = model.fit(x_train,
                    y_train,
                    epochs=4,
                    batch_size=1000,
                    validation_data=(x_test, y_test),
                    verbose=2)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_7 (Embedding)      (None, 250, 64)           1280128
_________________________________________________________________
conv1d_8 (Conv1D)            (None, 250, 16)           3088
_________________________________________________________________
max_pooling1d_5 (MaxPooling1 (None, 125, 16)           0
_________________________________________________________________
conv1d_9 (Conv1D)            (None, 125, 32)           1568
_________________________________________________________________
max_pooling1d_6 (MaxPooling1 (None, 62, 32)            0
_________________________________________________________________
conv1d_10 (Conv1D)           (None, 62, 64)            6208
_________________________________________________________________
flatten_7 (Flatten)          (None, 3968)              0
_________________________________________________________________
dropout_7 (Dropout)          (None, 3968)              0
_________________________________________________________________
dense_17 (Dense)             (None, 100)               396900
_________________________________________________________________
dense_18 (Dense)             (None, 1)                 101
=================================================================
Total params: 1,687,993
Trainable params: 1,687,993
Non-trainable params: 0
_________________________________________________________________
Train on 6665900 samples, validate on 20000 samples
Epoch 1/4
6044s - loss: 0.1935 - acc: 0.9226 - val_loss: 0.1708 - val_acc: 0.9305
Epoch 2/4
6226s - loss: 0.1651 - acc: 0.9344 - val_loss: 0.1620 - val_acc: 0.9357
Epoch 3/4
6309s - loss: 0.1565 - acc: 0.9379 - val_loss: 0.1609 - val_acc: 0.9351
Epoch 4/4
5372s - loss: 0.1501 - acc: 0.9407 - val_loss: 0.1593 - val_acc: 0.9365
```

So we have almost 93.7% accuracy, and only a tiny bit of overfitting.

5.1 Investigating the misclassified reviews

It's always interesting to look at misclassified examples to get a hint of what's going on and maybe get ideas for further improvements. That's what we're going to do now, with the first 100 examples.

Here are the predictions and the true labels for these samples:

In [32]:
```
x_sample = x_test
y_sample = y_test
preds = model.predict_classes(x_sample)
preds = np.array(preds).flatten()
print('true:')
print(y_sample)
print('predictions:')
print(preds)
```

Now, we select the misclassified examples, together with the true label and the prediction for these examples and print the first five:

In [33]:
```
idx = preds!=y_sample
miscl = x_sample[idx]
miscl_pred = preds[idx]
miscl_true = y_sample[idx]
```

In [34]:
```
for pred, true, rev in zip(miscl_pred[:5], miscl_true, miscl[:5]):
    rev = rev[rev!=0] # remove padding
    print(pred, true)
    print(' '.join(vocab.decode(rev)))
    print('\n')
```

```
1 0
on wet snowy sunday afternoon made trek out famous fairmount bagels si


1 0
went this little cozy restaurant with wife quiet intimate romantic res


1 0
happy when found out they have fish company at downtown food not as fr


0 1
this review parent or someone will visit with child without child woul


0 1
this not review as such hope try this place on next visit vegas but qu
```

It's not too easy to understand the reviews with all the missing words, especially the stop words like "the", "I", "a", etc. Still, let's try.

- 1st review: it seems that this person is comparing two bagel shops. She seems to like both and to refuse to compare them. Still, his rating is negative...
- 2nd review: the text is clearly super positive but the rating is negative...
- 3rd review: this person clearly states that this is not a review, and that she wants to ask a question about opening hours...
- 4th review: this is a mixed review. I understand that the food is very good but that the restaurant is too expensive and that there were a few issues. The person still recommends to try it once.
- 5th review: again a mixed review.

So it appears we're not doing so bad: among the 5 misclassified reviews, three are weird. The last two ones correspond to borderline cases.

We can build a pandas dataframe to look at the first 5 misclassified reviews. I know that these misclassified reviews are among the first 100 examples, so I will restrict the dataframe to this range:

In [35]:
```
import pandas as pd
# take the first 4 columns of the first 100 examples.
# give meaningful names to these columns
df = pd.DataFrame(data=data[:100,:4], columns=['stars','useful','funny',
# add a column to mark misclassified reviews:
df['misc'] = idx[:100]
# print the misclassified lines:
df[df['misc']==True]
```

Out[35]:

	stars	useful	funny	cool	misc
1	3	5	2	3	True
7	3	0	0	0	True
33	3	0	0	0	True
35	4	1	0	1	True
36	4	0	0	0	True
43	4	2	0	0	True

We don't learn much, only that these reviews are indeed borderline: they have 3 or 4 stars, and we set the boundary between our negative and positive categories between 3 and 4 stars.

5.2 Conclusion

Through this project, we learnt how to:

- perform sentiment analysis with keras and tensorflow on the large yelp dataset.
- tune a simple dense neural network for sentiment analysis.
- set up a deep convolutional network to improve performance and reduce training time.

We have seen that convolutional layers can really help in natural language processing, and are not to be restricted to image recognition. With these layers, the network is able to understand the meaning of small groups of words with a relatively small amount of data. Moreover, it is able to do so whatever the position of the group of words within the text.