Installation

The easiest way to install keras is using pip:

sudo pip install --upgrade keras

or you can install it from the source code:

git clone https://github.com/fchollet/keras.git (https://github.com/fchollet/keras.git)

cd keras

sudo python setup.py install

Required packages

Important libraries you need to import for this tutorial includes numpy, matplotlib, gensim and sklearn:

In [153]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.optimizers import SGD, RMSprop
from keras.utils import np_utils
from keras.layers import LSTM, SimpleRNN
from keras.layers.normalization import BatchNormalization
from keras.preprocessing import sequence
from keras.layers.embeddings import Embedding
from gensim.models import word2vec
from sklearn.decomposition import PCA

# initialize random seed for reproducibility
np.random.seed(666)
```

Word Embeddings

We will use a pre-trained word embedding model, GoogleNews-vectors-negative300.bin released by Google: https://code.google.com/archive/p/word2vec (https://code.google.com/archive/p/word2vec) This model includes pre-trained vectors trained on part of Google News dataset (about 100 billion words) and it contains 300-dimensional vectors for 3 million words and phrases.

In [120]:
```
# load the pre-trained word embedding model (it will take a while to loa
d)
word2vec_loc = "/Users/pinar/Downloads/GoogleNews-vectors-negative300.bi
n" # replace the path with your local file
word2vec_model = word2vec.Word2Vec.load_word2vec_format(word2vec_loc, bi
nary=True)
```

One way to explore the model is to visualize the embeddings of given list of words. The intuition here is that, similar words should be close to each other in d-dimensional space. Let's look at a list of jobs (the following example is taken from: [http://euler.stat.yale.edu/~tba3/stat665/lectures/lec20/notebook20.html (http://euler.stat.yale.edu/~tba3/stat665/lectures/lec20/notebook20.html)](http://euler.stat.yale.edu/~tba3/stat665/lectures/lec20/notebook20.html))

In [154]:
```
jobs = ["engineer", "professor", "teacher", "actor", "clergy", "musicia
n", "philosopher",
        "writer", "singer", "dancers", "model", "anesthesiologist", "aud
iologist",
        "chiropractor", "optometrist", "pharmacist", "psychologist", "ph
ysician",
        "architect", "firefighter", "judges", "lawyer", "biologist", "bo
tanist",
        "ecologist", "geneticist",  "zoologist", "chemist",
"programmer", "designer"]

# dimension of the entire vector for this word
print np.shape(word2vec_model['professor'])
```

```
(300,)
```

In [155]:
```
# an example embedding
print word2vec_model['professor'][0:50]
```

```
[ 0.25195312 -0.09960938  0.2265625   0.25390625  0.18261719  0.3300781
2
  0.33007812 -0.39257812  0.22558594 -0.41601562  0.31445312 -0.1767578
1
 -0.11669922 -0.02160645 -0.5         0.234375    0.234375    0.0412597
7
  0.08496094 -0.38671875  0.08984375  0.00817871 -0.06689453 -0.296875
  0.3125     -0.27148438 -0.27734375 -0.16210938 -0.51171875  0.0537109
4
  0.02978516  0.09667969 -0.08544922  0.01855469  0.40039062  0.0825195
3
  0.06787109  0.43164062 -0.06347656  0.15039062 -0.0625     -0.0703125
 -0.03417969  0.18164062 -0.11621094  0.00549316 -0.25       -0.0849609
4
  0.20019531  0.0078125 ]
```

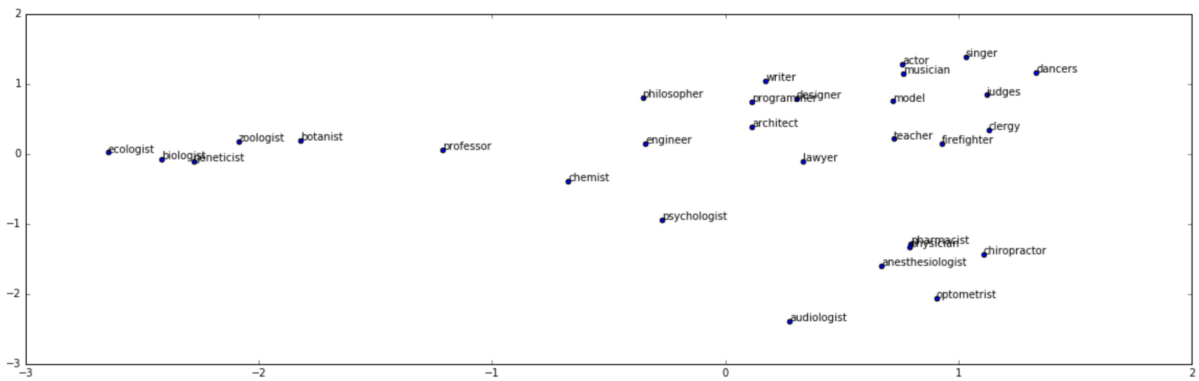
In [122]:
```
# get the d-dimensional embedding for each job in the list
embedding = np.array([word2vec_model[j] for j in jobs])
print embedding.shape
```

```
(30, 300)
```

In [123]:
```
# reduce the dimensions to 2 for visualization purposes
# read more about PCA: https://en.wikipedia.org/wiki/Principal_component
_analysis
pca = PCA(n_components=2)
pca.fit(embedding)
embedding_pca = np.transpose(pca.transform(embedding))
print embedding_pca.shape
```

```
(2, 30)
```

In [124]:
```
plt.figure(figsize=(20, 6))
plt.scatter(embedding_pca[0], embedding_pca[1])
for index,(x,y) in enumerate(np.transpose(embedding_pca)):
        plt.text(x,y,jobs[index], fontsize=10)
```



Let's explore some more examples. Can we find the top-N most similar words for a given word? This following method computes cosine similarity between a the weight vector of the given word and the vectors for each word in the model:

In [125]:
```
for widx, word in enumerate(word2vec_model.most_similar('Xbox',
topn=10)):
    print("%s) %s (%s)"%(widx, word[0], word[1]))
```

```
0) PS3 (0.840408086777)
1) XBox (0.834058463573)
2) XBOX (0.78752720356)
3) PlayStation (0.784932434559)
4) Xbox### (0.770972728729)
5) Playstation (0.737725734711)
6) PS2 (0.723367214203)
7) PS3_Xbox (0.717997670174)
8) Kinect (0.715201616287)
9) Wii (0.709048390388)
```

How about word analogies? In this case, we can feed the function a list of positive words which contribute positively towards the similarity and a list of negative words.

Famous example: if "man" is to "woman", then "king" is to _? In other words, can we extract the relationship between "man" and "woman" (i.e. gender) and apply this relationship to "king" to discover the answer? (hint: we are essentially doing an arithmetic operation between word vectors, e.g. vec(queen) ≈ vec(woman) - vec(man) + vec(king).

In [156]:
```
word2vec_model.most_similar(positive=['woman', 'king'], negative=
['man'])
```

Out[156]:
```
[(u'queen', 0.7118192315101624),
 (u'monarch', 0.6189674139022827),
 (u'princess', 0.5902431011199951),
 (u'crown_prince', 0.5499460697174072),
 (u'prince', 0.5377321839332581),
 (u'kings', 0.5236844420433044),
 (u'Queen_Consort', 0.5235946178436279),
 (u'queens', 0.5181134343147278),
 (u'sultan', 0.5098593235015869),
 (u'monarchy', 0.5087412595748901)]
```

Exercise: can you discover the same result without using most_similar() function?

Another example: if Einstein is to scientist, then Mozart to _?

In [128]:
```
word2vec_model.most_similar(positive=['Mozart', 'scientist'], negative=
['Einstein'])
```

Out[128]:
```
[(u'composer', 0.5571082234382629),
 (u'researcher', 0.5302102565765381),
 (u'biologist', 0.527501106262207),
 (u'cellist', 0.5128622055053711),
 (u'pianist', 0.5066508054733276),
 (u'microbiologist', 0.49905431270599365),
 (u'cello_soloist', 0.4963686466217041),
 (u'violinist', 0.492169052362442),
 (u'soprano', 0.48578697443008423),
 (u'Chamber_Orchestra', 0.4776730537414551)]
```

Fun exercise: Try if Japan is to sushi, then Germany is to _?

More interesting stuff: can we find which word doesn't belong to its group? doesnt_match() function outputs which word in a set is most dissimilar from the others:

In [129]:
```
word2vec_model.doesnt_match("Elevenses Luncheon Afternoon_tea Dinner Sup
per Cereal".split())
```

Out[129]: 'Cereal'

Introduction to Deep Learning with Keras

Loading data

In this tutorial, we will use IMDB dataset that is available from Keras. This dataset consist of 25,000 movies reviews from IMDB, labeled by the sentiment of each review (positive/negative).

An positive review from the dataset:

> What a surprise; two outstanding performances by the lead actresses in this film. This is the best work Busy Phillips has ever done and the best from Erika Christensen since Traffic. This film certainly should be in Oscar contention. See this movie!

A negative review from the dataset:

> Not only is it a disgustingly made low-budget bad-acted movie, but the plot itself is just STUPID!!! A mystic man that eats women? (And by the looks, not virgin ones) Ridiculous!!! If you've got nothing better to do (like sleeping) you should watch this. Yeah right.

For convenience, Keras preprocessed the reviews where each review is encoded as a sequence of word indexes. Words are indexed by overall frequency in the dataset, e.g. "3" encodes the third most frequent word in the dataset. This convention allows easy filtering operations, e.g. take top 10,000 most common words, but eliminate the top 20 most common words.

For computational purposes, let's apply some cutoffs/filtering to the dataset. Let's say we would like to consider only top 500 most commonly used words in the dataset (e.g. we are reducing the vocabulary size to 500) and we want to cap each review at 100 words (since some reviews can be too long). Finally, we would like to represent each word as a 128-dimensional vector.

In [161]:
```
vocab_size = 500
num_dimension = 128
max_length = 100
batch_size = 32

print('Loading data...')
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=vocab_siz
e)
X_train = sequence.pad_sequences(X_train, maxlen=max_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_length)

print X_train[0]
```

```
Loading data...
[  2  33   6  22  12 215  28  77  52   5  14 407  16  82   2   8   4 10
7
 117   2  15 256   4   2   7   2   5   2  36  71  43   2 476  26 400 31
7
  46   7   4   2   2  13 104  88   4 381  15 297  98  32   2  56  26 14
1
   6 194   2  18   4 226  22  21 134 476  26 480   5 144  30   2  18   5
1
  36  28 224  92  25 104   4 226  65  16  38   2  88  12  16 283   5   1
6
   2 113 103  32  15  16   2  19 178  32]
```

In [162]:
```
print X_train.shape
```

```
(25000, 100)
```

In [163]:
```
print y_train[0:10]
```

```
[1 0 0 1 0 0 1 0 1 0]
```

Python exercise: can you download the original IMDB dataset (http://ai.stanford.edu/~amaas/data/sentiment (http://ai.stanford.edu/~amaas/data/sentiment)) and pre-process the dataset yourself in order to create X_train and X_test (similarly y_train and y_test).

Defining a Model

First step is defining a neural network. Keras allows us to define a neural network as a linear stack of layers by using a class called Sequential(). This class serves as a container for these layers where we can add one layer at a time in the order that they should be connected (think of it as a pipeline that takes the input at the bottom, and outputs the predictions at the top). Lets create an instance:

In [131]:
```
model = Sequential()
```

Now lets start adding layers to our sequential model. Our first layer will be vector embeddings where we encode sequences of words into sequences of d-dimensional word vectors. In Keras, this is available via Embedding() layer (view its source code here: [https://github.com/fchollet/keras/blob/master/keras/layers/embeddings.py (https://github.com/fchollet/keras/blob/master/keras/layers/embeddings.py)](https://github.com/fchollet/keras/blob/master/keras/layers/embeddings.py)).

In [132]:
```
model.add(Embedding(vocab_size, num_dimension, input_length=max_length))
model.add(Dropout(0.25)) # to prevent overfitting
```

Lets see an example embedding of a word:

In [133]:
```
print(model.layers[0].get_weights()[0][0])
```

```
[-0.00889014 -0.01621493  0.00891493 -0.02468174 -0.02940076  0.0076559
9
 -0.03482467 -0.02339622  0.03063031  0.02401075 -0.02271061  0.0361569
 -0.0123959   0.0163225   0.03789079 -0.01465467 -0.00363456  0.0285076
9
  0.03645847  0.00569521 -0.00686419 -0.02747821 -0.03073228 -0.0131191
4
 -0.03742161  0.01105579  0.02726154 -0.02943268 -0.01203082 -0.0295983
1
 -0.00413988  0.0214023   0.03506959 -0.00910951 -0.004139    0.0078901
1
  0.02500707 -0.01255448  0.01788012 -0.04161823 -0.0259219  -0.0177736
4
  0.03232638 -0.0396347   0.04895642 -0.03504157  0.0106763   0.0455210
4
  0.04849952 -0.02137645  0.01198982  0.00262615  0.02443031  0.0212870
4
 -0.0278707   0.02527844 -0.03369141  0.02659073  0.02221744 -0.0120317
3
  0.01907264  0.00469748  0.04542368 -0.04407709  0.00734522  0.0405354
6
  0.02031228 -0.03571615  0.00524275  0.03088065  0.03671978 -0.0416312
1
  0.02784361  0.00560471  0.02118326  0.01303958 -0.01212993 -0.0191852
 -0.04345414 -0.02987951  0.04314576 -0.01583857 -0.01063441  0.0485505
  0.0433744  -0.00570309 -0.04411818  0.04663708  0.03508854 -0.0002496
4
 -0.04669866  0.02308167 -0.02465213  0.04137044 -0.03000971 -0.0175892
3
 -0.00839039  0.01278447  0.02640252 -0.01103086  0.00562639 -0.0068524
6
 -0.03703234 -0.04326813 -0.02416615 -0.01719688 -0.01875932  0.0315151
2
  0.02874782 -0.00103003 -0.01436211  0.03889801  0.00642155  0.0377351
5
  0.00939817  0.00568601  0.01786224  0.00947665 -0.02333875  0.0132854
7
 -0.00696313  0.00708728 -0.04323623  0.03621923  0.01954523 -0.0294320
1
  0.03435225  0.02278549]
```

The output of this first layer would be a matrix with the size 500x128:

In [134]:
```
print(model.layers[0].get_weights()[0].shape)
```

```
(500, 128)
```

In [135]:
```
model.add(Flatten()) #  flatten out the output of Embedding layer
```

Exercise: Try other strategies to reduce the embedding layer, e.g. averaging the weight vectors.

Dropout

Getting back to Dropouts:

> model.add(Dropout(0.25))

Dropout is a simple regularization technique to prevent overfitting. When a neural network learns, dropout technique randomly selects some neurons to be ignored during training (e.g. neurons are dropped-out randomly). In other words, some neurons temporarily become unable to contribute to the activation function (thus, other neurons will have to step in and make predictions for the missing neurons). This helps the model to learn multiple internal representations, and network becomes less sensetive to specific weights of some neurons (i.e. better generalization).

Dropout() function takes a parameter to drop out randomly selected neurons with a given probability (e.g. 25% in this case).

Dense layer

Dense() class encodes a fully-connected network structure where we can specify the number of neurons in the layer as the first argument.

In [136]:
```
model.add(Dense(256))
model.add(Dropout(0.25))
```

After that, we use an activation function on this layer. There are several activation functions we can use. In this tutorial, we will use Rectified Linear Units "relu" since it has been shown to work efficiently in deep networks (it introduces sparsity and non-linearity) and it does not face with gradient vanishing problem as with the sigmoid function.

In [137]:
```
model.add(Activation('relu')) # rectifier activation function
```

Sigmoid is often used for classification in the output layer because it provides probabilities for different classes. This activation function will ensure that our network output is between 0 and 1.

In [138]:
```
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

Exercise: explore other activation functions: https://keras.io/activations (https://keras.io/activations)

Compiling a model

After defining the model architecture, we need to compile it (this is where Theano or Tensorflow serves as a backend). When compiling the model, we need to decide some important properties that is required to train the network:

1. Loss function (evaluate a given set of weights)
2. Optimizer (search through different weights)
3. Metrics to collect and report

In [139]:
```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['ac
curacy'])
```

In this case, we used "binary_crossentropy", a logarithmic loss for binary classification problems, and we used a gradient descent algorithm "adam". Finally, because we are doing classification, we will collect and report the classification accuracy as the metric.

Exercise: try different optimizers: https://keras.io/optimizers (https://keras.io/optimizers) and different losses: https://keras.io/objectives (https://keras.io/objectives)

Fitting a model

Now we defined and compiled our model, the next step is to execute the model on our data. Two important parameters during the training process are nb_epoch and batch_size.

```
> nb_epoch: specifies a fixed number of iterations through the datasets (i.
e. epochs).

> batch_size: specifies number of instances that needs to be evaluated befor
e a weight update is performed in the network. Batch size is also important
 to ensure not too many instances are loaded into memory at a given time.
```

This step is where the network is trained using backpropogation algorithm, and optimized via the specified optimization algorithm and evaluated via specified loss function (i.e. the step where actual work happens in your CPU/GPU).

In [ ]:
```
print("Training...")
model.fit(X_train, y_train, nb_epoch=10, batch_size=10, validation_data=
(X_test, y_test), verbose=0)
print("Training is done.")
```

```
Training...
```

Evaluating the model

We will evaluate our trained neural network on the test set:

In [143]:
```
score, acc = model.evaluate(X_test, y_test, batch_size=batch_size)
print('Test accuracy:', acc)
```

```
24992/25000 [============================>.] - ETA: 0s('Test accurac
y:', 0.76471999999999996)
```

In [144]:
```
model.predict(X_test[0:10])
```

Out[144]:
```
array([[ 0.94634795],
       [ 0.03117599],
       [ 0.05857973],
       [ 0.99933451],
       [ 0.98330998],
       [ 0.25289002],
       [ 0.23697764],
       [ 0.70019507],
       [ 0.99735594],
       [ 0.99821842]], dtype=float32)
```

In [145]:
```
print(model.layers[0].get_weights()[0].shape) # Embedding
print(model.layers[3].get_weights()[0].shape) # Dense(256)
print(model.layers[6].get_weights()[0].shape) # Dense(1)
```

```
(500, 128)
(12800, 256)
(256, 1)
```

Recap the whole structure

We defined the network

> model = Sequential()
>
> model.add(Embedding(vocab_size, num_dimension, input_length=max_length))
> model.add(Dropout(0.25))
>
> model.add(Flatten())
>
> model.add(Dense(256))
>
> model.add(Dropout(0.25))
>
> model.add(Activation('relu'))
>
> model.add(Dense(1))
>
> model.add(Activation('sigmoid'))

We compiled the network

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

We fit the network

> model.fit(X_train, y_train, nb_epoch=10, batch_size=10, validation_data=(X_test, y_test))

We evaluated the network

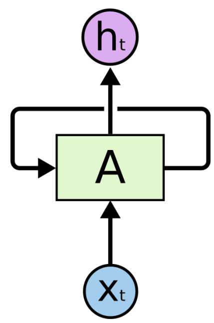
> score, acc = model.evaluate(X_test, y_test, batch_size=batch_size)

Exercise: Try different batch_size and nb_epoch parameters and observe how performance changes.

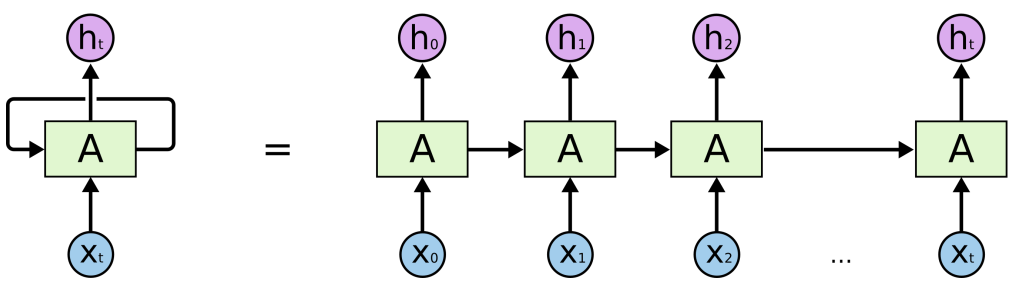
Basic RNN

From http://colah.github.io/posts/2015-08-Understanding-LSTMs (http://colah.github.io/posts/2015-08-Understanding-LSTMs)

> Humans don't start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence. Traditional neural networks can't do this, and it seems like a major shortcoming. Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.



> A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.



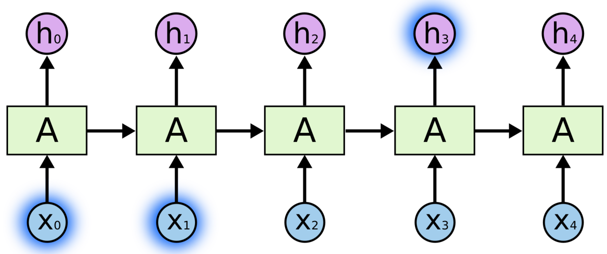
In [151]:
```
model = Sequential()
model.add(Embedding(vocab_size, 128, dropout=0.2))
model.add(Dropout(0.25))

model.add(SimpleRNN(128, dropout_W=0.2, dropout_U=0.2))

model.add(Dense(256))
model.add(Dropout(0.25))
model.add(Activation('relu'))

model.add(Dense(1))
model.add(Activation('sigmoid'))

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

print('Training...')
model.fit(X_train, y_train, batch_size=batch_size, nb_epoch=10, validati
on_data=(X_test, y_test), verbose=0)
print('Training is done.')

score, acc = model.evaluate(X_test, y_test, batch_size=batch_size)
print('Test accuracy:', acc)
```

```
Training...
Training is done.
24992/25000 [============================>.] - ETA: 0s('Test accurac
y:', 0.69179999999999997)
```

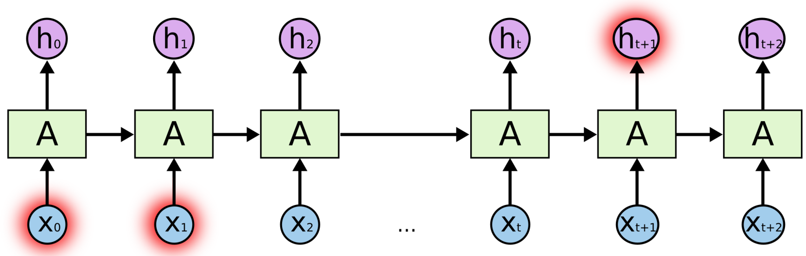
LSTM

Recurrent networks take as their input not just the current input example, but also what they perceived one step back in time. From http://colah.github.io/posts/2015-08-Understanding-LSTMs (http://colah.github.io/posts/2015-08-Understanding-LSTMs):

> Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the *sky*," we don't need any further context – it's pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.



> But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France… I speak fluent *French*." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large. Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.



LSTMs are proposed as a special kind of RNN, capable of learning long-term dependencies.

In [152]:
```
model = Sequential()
model.add(Embedding(vocab_size, 128, dropout=0.2))
model.add(Dropout(0.25))

model.add(LSTM(128, dropout_W=0.2, dropout_U=0.2))

model.add(Dense(256))
model.add(Dropout(0.25))
model.add(Activation('relu'))

model.add(Dense(1))
model.add(Activation('sigmoid'))

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

print('Train...')
model.fit(X_train, y_train, batch_size=batch_size, nb_epoch=2, validatio
n_data=(X_test, y_test), verbose=0)
print('Training is done.')

score, acc = model.evaluate(X_test, y_test, batch_size=batch_size)
print('Test accuracy:', acc)
```

```
Train...
Training is done.
25000/25000 [==============================] - 37s
('Test accuracy:', 0.78620000000000001)
```

Exercise: try with GRU (https://keras.io/layers/recurrent (https://keras.io/layers/recurrent))

Run the following example and observe how algorithm learns to generate text over time:

Generate text from Nietzsche's writings:
https://github.com/fchollet/keras/blob/master/examples/lstm_text_generation.py
(https://github.com/fchollet/keras/blob/master/examples/lstm_text_generation.py)

1st iteration:

> ybaclicaply and the doove the bloogs know, we refineraliknat of litteverliful. 2e1h in achnders itlessionishce ling ougovically sook zo affects with do edoend chunk of womfulvents, brut them andn volfelved very the selenes.s. no kniwnor hables himself intellectually timenal and ielvanist mora

10th iteration:

> god, we are on the point of successfully and of the greatest and said, the delight of his father they are the periods and such a pleasure of the subjections of which the hand, and a because a great pathious some the christian of the period to the same flated and is the senses the philosoperly of the taste of the self-contrary and long the anceitic are every morality of the present to the fact the concealed to the same contemplate the s