2.1 Getting started

In this project you will implement a popular Recurrent Neural Network (RNN) architecture to create an English language sequence generator capable of building semi-coherent English sentences from scratch by building them up character-by-character. This will require a substantial amount amount of parameter tuning on a large training corpus (at least 100,000 characters long). In particular for this project we will be using a complete version of Sir Arthur Conan Doyle's classic book The Adventures of Sherlock Holmes.

How can we train a machine learning model to generate text automatically, character-by-character? *By showing the model many training examples so it can learn a pattern between input and output.* With this type of text generation each input is a string of valid characters like this one

dogs are grea

whlie the corresponding output is the next character in the sentence - which here is 't' (since the complete sentence is 'dogs are great'). We need to show a model many such examples in order for it to make reasonable predictions.

Fun note: For those interested in how text generation is being used check out some of the following fun resources:

- Generate wacky sentences with this academic RNN text generator
- Various twitter bots that tweet automatically generated text likethis one.
- the NanoGenMo annual contest to automatically produce a 50,000+ novel automatically
- Robot Shakespeare a text generator that automatically produces Shakespear-esk sentences

2.2 Preprocessing a text dataset

Our first task is to get a large text corpus for use in training, and on it we perform a several light pre-processing tasks. The default corpus we will use is the classic book Sherlock Holmes, but you can use a variety of others as well - so long as they are fairly large (around 100,000 characters or more).

In [14]:

```
# read in the text, transforming everything to lower case
text = open('datasets/holmes.txt').read().lower()
print('our original text has ' + str(len(text)) + ' characters')
```

```
our original text has 581864 characters
```

Next, lets examine a bit of the raw text. Because we are interested in creating sentences of English words automatically by building up each word character-by-character, we only want to train on valid English words. In other words - we need to remove all of the other characters that are not part of English words.

In [15]:

```
### print out the first 1000 characters of the raw text to get a sense of what we need to throw out
text[:2000]
```

Out[15]:

```
"\ufeffproject gutenberg's the adventures of sherlock holmes, by arthur conan doyle\n\nthis ebook is for the use of anyone anywhere at no cost and with\nalmost no restrictions whatsoever.  you may copy it, give it away or\nre-use it under the terms of the project gutenberg license included\nwith this ebook or online at www.gutenberg.net\n\n\ntitle: the adventures of sherlock holmes\n\nauthor: arthur conan doyle\n\nposting date: april 18, 2011 [ebook #1661]\nfirst posted: november 29, 2002\n\nlanguage: english\n\n\n*** start of this project gutenberg ebook the adventures of sherlock holmes ***\n\n\n\n\nproduced by an anonymous project gutenberg volunteer and jose menendez\n\n\n\n\n\n\n\n\n\nthe adventures of sherlock holmes\n\nby\n\nsir
```

```
arthur conan doyle\n\n\n\n   i. a scandal in bohemia\n  ii. the red-headed
league\n iii. a case of identity\n  iv. the boscombe valley mystery\n   v. the
five orange pips\n  vi. the man with the twisted lip\n vii. the adventure of the
blue carbuncle\nviii. the adventure of the speckled band\n  ix. the adventure of
the engineer's thumb\n   x. the adventure of the noble bachelor\n  xi. the
adventure of the beryl coronet\n xii. the adventure of the copper
beeches\n\n\n\n\nadventure i. a scandal in bohemia\n\ni.\n\nto sherlock holmes
she is always the woman. i have seldom heard\nhim mention her under any other
name. in his eyes she eclipses\nand predominates the whole of her sex. it was
not that he felt\nany emotion akin to love for irene adler. all emotions, and
that\none particularly, were abhorrent to his cold, precise but\nadmirably
balanced mind. he was, i take it, the most perfect\nreasoning and observing
machine that the world has seen, but as a\nlover he would have placed himself in
a false position. he never\nspoke of the softer passions, save with a gibe and a
sneer. they\nwere admirable things for the observer--excellent for drawing
the\nveil from men's motives and actions. but for the trained reasoner\nto admit
such intrusions into his own delicate and finely\nadjusted temperament was to
introduce a dist"
```

Wow - there's a lot of junk here (i.e., weird uncommon character combinations - as this first character chunk contains the title and author page, as well as table of contents)! To keep things simple, we want to train our RNN on a large chunk of more typical English sentences - we don't want it to start thinking non-english words or strange characters are valid! - so lets clean up the data a bit.

First, since the dataset is so large and the first few hundred characters contain a lot of junk, lets cut it out. Lets also find-and-replace those newline tags with empty spaces.

In [16]:

```
### find and replace '\n' and '\r' symbols - replacing them
text = text[1302:]
text = text.replace('\n',' ')    # replacing '\n' with '' simply removes the
sequence
text = text.replace('\r',' ')
```

Lets see how the first 1000 characters of our text looks now!

In [17]:

```
### print out the first 1000 characters of the raw text to get a sense of what
we need to throw out
text[:1000]
```

Out[17]:

```
"is eyes she eclipses and predominates the whole of her sex. it was not that he
felt any emotion akin to love for irene adler. all emotions, and that one
particularly, were abhorrent to his cold, precise but admirably balanced mind.
he was, i take it, the most perfect reasoning and observing machine that the
world has seen, but as a lover he would have placed himself in a false position.
he never spoke of the softer passions, save with a gibe and a sneer. they were
admirable things for the observer--excellent for drawing the veil from men's
motives and actions. but for the trained reasoner to admit such intrusions into
his own delicate and finely adjusted temperament was to introduce a distracting
factor which might throw a doubt upon all his mental results. grit in a
sensitive instrument, or a crack in one of his own high-power lenses, would not
```

```
be more disturbing than a strong emotion in a nature such as his. and yet there
was but one woman to him, and that woman was the late irene ad"
```

TODO: finish cleaning the text

Lets make sure we haven't left any other atypical characters (commas, periods, etc., are ok) lurking around in the depths of the text. You can do this by ennumerating all the text's unique characters, examining them, and then replacing any unwanted characters with empty spaces! Once we find all of the text's unique characters, we can remove all of the atypical ones in the next cell. Note: don't remove the punctuation marks given in my_answers.py.

In [18]:

```
### TODO: implement cleaned_text in my_answers.py
from my_answers import cleaned_text

text = cleaned_text(text)

# shorten any extra dead space created above
text = text.replace('  ',' ')
```

With your chosen characters removed print out the first few hundred lines again just to double check that everything looks good.

In [19]:

```
### print out the first 2000 characters of the raw text to get a sense of what
we need to throw out
text[:2000]
```

Out[19]:

```
'is eyes she eclipses and predominates the whole of her sex. it was not that he
felt any emotion akin to love for irene adler. all emotions, and that one
particularly, were abhorrent to his cold, precise but admirably balanced mind.
he was, i take it, the most perfect reasoning and observing machine that the
world has seen, but as a lover he would have placed himself in a false position.
he never spoke of the softer passions, save with a gibe and a sneer. they were
admirable things for the observer excellent for drawing the veil from men s
motives and actions. but for the trained reasoner to admit such intrusions into
his own delicate and finely adjusted temperament was to introduce a distracting
factor which might throw a doubt upon all his mental results. grit in a
sensitive instrument, or a crack in one of his own high power lenses, would not
be more disturbing than a strong emotion in a nature such as his. and yet there
was but one woman to him, and that woman was the late irene adler, of dubious
and questionable memory. i had seen little of holmes lately. my marriage had
drifted us away from each other. my own complete happiness, and the home centred
interests which rise up around the man who first finds himself master of his own
establishment, were sufficient to absorb all my attention, while holmes, who
loathed every form of society with his whole bohemian soul, remained in our
lodgings in baker street, buried among his old books, and alternating from week
to week between cocaine and ambition, the drowsiness of the drug, and the fierce
energy of his own keen nature. he was still, as ever, deeply attracted by the
study of crime, and occupied his immense faculties and extraordinary powers of
observation in following out those clues, and clearing up those mysteries which
had been abandoned as hopeless by the official police. from time to time i heard
```

```
some vague account of his doings: of his summons to odessa in the case of the
trepoff murder, of his clearing up o'
```

Now that we have thrown out a good number of non-English characters/character sequences lets print out some statistics about the dataset - including number of total characters and number of unique characters.

In [20]:

```
# count the number of unique characters in the text
chars = sorted(list(set(text)))

# print some of the text, as well as statistics
print ("this corpus has " +  str(len(text)) + " total number of characters")
print ("this corpus has " +  str(len(chars)) + " unique characters")
```

```
this corpus has 573681 total number of characters
this corpus has 33 unique characters
```

2.3 Cutting data into input/output pairs

Now that we have our text all cleaned up, how can we use it to train a model to generate sentences automatically? First we need to train a machine learning model - and in order to do that we need a set of input/output pairs for a model to train on. How can we create a set of input/output pairs from our text to train on?
Remember in part 1 of this notebook how we used a sliding window to extract input/output pairs from a time series? We do the same thing here! We slide a window of length T along our giant text corpus - everything in the window becomes one input while the character following becomes its corresponding output. This process of extracting input/output pairs is illustrated in the gif below on a small example text using a window size of T = 5.

Notice one aspect of the sliding window in this gif that does not mirror the analaogous gif for time series shown in part 1 of the notebook - we do not need to slide the window along one character at a time but can move by a fixed step size M greater than 1 (in the gif indeed $M=1$). This is done with large input texts (like ours which has over 500,000 characters!) when sliding the window along one character at a time we would create far too many input/output pairs to be able to reasonably compute with.

More formally lets denote our text corpus - which is one long string of characters - as follows

$$s0,s1,s2,...,sPs0,s1,s2,...,sP$$

where P is the length of the text (again for our text $P \approx 500,000$!). Sliding a window of size T = 5 with a step length of M = 1 (these are the parameters shown in the gif above) over this sequence produces the following list of input/output pairs

Input	Output
$\langle s_1, s_2, s_3, s_4, s_5 \rangle$	s_6
$\langle s_2, s_3, s_4, s_5, s_6 \rangle$	s_7
$\vdots$	$\vdots$
$\langle s_{P-5}, s_{P-4}, s_{P-3}, s_{P-2}, s_{P-1} \rangle$	s_P

Notice here that each input is a sequence (or vector) of 4 characters (and in general has length equal to the window size T) while each corresponding output is a single character. We created around P total number of input/output pairs (for general step size M we create around ceil(P/M) pairs).

Now its time for you to window the input time series as described above!

TODO: Create a function that runs a sliding window along the input text and creates associated input/output pairs. A skeleton function has been provided for you. Note that this function should input a) the text b) the window size and c) the step size, and return the input/output sequences. Note: the return items should be *lists* - not numpy arrays.

Sequence generator

(remember to copy your completed function into the script *my_answers.py* function titled *window_transform_text* before submitting your project)

In [21]:

```
### TODO: implement window_transform_series in my_answers.py
from my_answers import window_transform_series
```

With our function complete we can now use it to produce input/output pairs! We employ the function in the next cell, where the window_size = 100 and step_size = 5.

In [22]:

```
# run your text window-ing function
window_size = 100
step_size = 5
inputs, outputs = window_transform_text(text,window_size,step_size)
```

Lets print out a few input/output pairs to verify that we have made the right sort of stuff!

In [23]:

```
# print out a few of the input/output pairs to verify that we've made the right kind of stuff to learn from
print('input = ' + inputs[2])
print('output = ' + outputs[2])
print('--------------')
print('input = ' + inputs[100])
print('output = ' + outputs[100])
```

```
input = e eclipses and predominates the whole of her sex. it was not that he felt any emotion akin to love f
output = o
--------------
input = er excellent for drawing the veil from men s motives and actions. but for the trained reasoner to ad
output = m
```

Looks good!

2.4 Wait, what kind of problem is text generation again?

In part 1 of this notebook we used the same pre-processing technique - the sliding window - to produce a set of training input/output pairs to tackle the problem of time series prediction *by treating the problem as one of regression*. So what sort of problem do we have here now, with text generation? Well, the time series prediction was a regression problem because the output (one value of the time series) was a continuous value. Here - for character-by-character text generation - each output is a *single character*. This isn't a continuous value - but a distinct class - therefore **character-by-character text generation is a classification problem**.

How many classes are there in the data? Well, the number of classes is equal to the number of unique characters we have to predict! How many of those were there in our dataset again? Lets print out the value again.

In [24]:

```
# print out the number of unique characters in the dataset
chars = sorted(list(set(text)))
print("this corpus has " +  str(len(chars)) + " unique characters")
print('and these characters are ')
print(chars)
```

```
this corpus has 33 unique characters
```

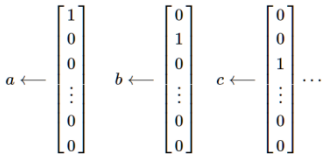
```
and these characters are
[' ', '!', ',', '.', ':', ';', '?', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y',
'z']
```

Rockin' - so we have a multi-class classification problem on our hands!

2.5 One-hot encoding characters

There's just one last issue we have to deal with before tackle: machine learning algorithm deal with numerical data and all of our input/output pairs are characters. So we just need to transform our characters into equivalent numerical values. The most common way of doing this is via a 'one-hot encoding' scheme. Here's how it works.

We transform each character in our inputs/outputs into a vector with length equal to the number of unique characters in our text. This vector is all zeros except one location where we place a 1 - and this location is unique to each character type. e.g., we transform 'a', 'b', and 'c' as follows



where each vector has 32 entries (or in general: number of entries = number of unique characters in text). The first practical step towards doing this one-hot encoding is to form a dictionary mapping each unique character to a unique integer, and one dictionary to do the reverse mapping. We can then use these dictionaries to quickly make our one-hot encodings, as well as re-translate (from integers to characters) the results of our trained RNN classification model.

In [25]:

```
# this dictionary is a function mapping each unique character to a unique int
chars_to_indices = dict((c, i) for i, c in enumerate(chars))  # map each unique
character to unique integer

# this dictionary is a function mapping each unique integer back to a unique
character
indices_to_chars = dict((i, c) for i, c in enumerate(chars))  # map each unique
integer back to unique character
```

Now we can transform our input/output pairs - consisting of characters - to equivalent input/output pairs made up of one-hot encoded vectors. In the next cell we provide a function for doing just this: it takes in the raw character input/outputs and returns their numerical versions. In particular the numerical input is given as XX, and numerical output is given as the yy

In [26]:

```
# transform character-based input/output into equivalent numerical versions
def encode_io_pairs(text, window_size, step_size):
    # number of unique chars
    chars = sorted(list(set(text)))
    num_chars = len(chars)

    # cut up text into character input/output pairs
    inputs, outputs = window_transform_text(text, window_size, step_size)


    # create empty vessels for one-hot encoded input/output
```

Sequence generator

```
    X = np.zeros((len(inputs), window_size, num_chars), dtype=np.bool)
    y = np.zeros((len(inputs), num_chars), dtype=np.bool)

    # loop over inputs/outputs and tranform and store in X/y
    for i, sentence in enumerate(inputs):
        for t, char in enumerate(sentence):
            X[i, t, chars_to_indices[char]] = 1
        y[i, chars_to_indices[outputs[i]]] = 1
    return X, y
```

Now run the one-hot encoding function by activating the cell below and transform our input/output pairs!

In [27]:

```
# use your function
window_size = 100
step_size = 5
X, y = encode_io_pairs(text, window_size, step_size)
```

2.6 Setting up our RNN

With our dataset loaded and the input/output pairs extracted / transformed we can now begin setting up our RNN for training. Again we will use Keras to quickly build a single hidden layer RNN - where our hidden layer consists of LTSM modules.

Time to get to work: build a 3 layer RNN model of the following specification

- layer 1 should be an LSTM module with 200 hidden units --> note this should have input_shape = (window_size,len(chars)) where len(chars) = number of unique characters in your cleaned text
- layer 2 should be a linear module, fully connected, with len(chars) hidden units --> where len(chars) = number of unique characters in your cleaned text
- layer 3 should be a softmax activation (since we are solving a *multiclass classification*)
- Use the **categorical_crossentropy** loss

This network can be constructed using just a few lines - as with the RNN network you made in part 1 of this notebook. See e.g., the general Keras documentation and the LTSM documentation in particular for examples of how to quickly use Keras to build neural network models.

In [28]:

```
### necessary functions from the keras library
from keras.models import Sequential
from keras.layers import Dense, Activation, LSTM
from keras.optimizers import RMSprop
from keras.utils.data_utils import get_file
import keras
import random

# TODO implement build_part2_RNN in my_answers.py
from my_answers import build_part2_RNN

model = build_part2_RNN(window_size, len(chars))

# initialize optimizer
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08,
decay=0.0)
```

```
# compile model --> make sure initialized optimizer and callbacks - as defined
above - are used
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_2 (LSTM)                (None, 200)               187200
_________________________________________________________________
dense_2 (Dense)              (None, 33)                6633
_________________________________________________________________
activation_1 (Activation)    (None, 33)                0
=================================================================
Total params: 193,833
Trainable params: 193,833
Non-trainable params: 0
_________________________________________________________________
```

2.7 Training our RNN model for text generation

With our RNN setup we can now train it! Lets begin by trying it out on a small subset of the larger version. In the next cell we take the first 10,000 input/output pairs from our training database to learn on.

In [29]:

```
# a small subset of our input/output pairs
Xsmall = X[:10000,:,:]
ysmall = y[:10000,:]
```

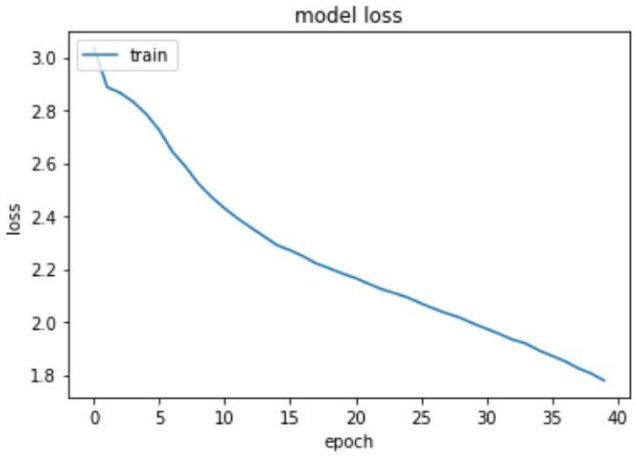
Now lets fit our model!

In [30]:

```
# train the model
epochtimer = EpochTimer()
epochs = 40
hist = model.fit(Xsmall, ysmall, batch_size=500, epochs=epochs, verbose=1,
callbacks=[epochtimer])
show_history_graph(hist)

# save weights
model.save_weights('model_weights/best_RNN_small_textdata_weights.hdf5')
```

```
Epoch 1/40
 9500/10000 [===========================>..] - ETA: 0s - loss: 3.0412Epoch 0
took 4.626322099999015 seconds
10000/10000 [==============================] - 4s - loss: 3.0343
Epoch 2/40
 9500/10000 [===========================>..] - ETA: 0s - loss: 2.8896Epoch 1
took 4.309310615999493 seconds
.....

Epoch 40/40
 9500/10000 [===========================>..] - ETA: 0s - loss: 1.7752Epoch 39
took 4.3305657820001215 seconds
10000/10000 [==============================] - 4s - loss: 1.7787
```

```
Training took 173.52824341199994 seconds
```



How do we make a given number of predictions (characters) based on this fitted model?

First we predict the next character after following any chunk of characters in the text of length equal to our chosen window size. Then we remove the first character in our input sequence and tack our prediction onto the end. This gives us a slightly changed sequence of inputs that still has length equal to the size of our window. We then feed in this updated input sequence into the model to predict the another character. Together then we have two predicted characters following our original input sequence. Repeating this process N times gives us N predicted characters.

In the next Python cell we provide you with a completed function that does just this - it makes predictions when given a) a trained RNN model, b) a subset of (window_size) characters from the text, and c) a number of characters to predict (to follow our input subset).

In [31]:

```
# function that uses trained model to predict a desired number of future
characters
def predict_next_chars(model, input_chars, num_to_predict):
    # create output
    predicted_chars = ''
    for i in range(num_to_predict):
        # convert this round's predicted characters to numerical input
        x_test = np.zeros((1, window_size, len(chars)))
        for t, char in enumerate(input_chars):
            x_test[0, t, chars_to_indices[char]] = 1.

        # make this round's prediction
        test_predict = model.predict(x_test, verbose=0)[0]

        # translate numerical prediction back to characters
        r = np.argmax(test_predict) # predict class of each test input
        d = indices_to_chars[r]

        # update predicted_chars and input
        predicted_chars += d
        input_chars += d
        input_chars = input_chars[1:]
    return predicted_chars
```

With your trained model try a few subsets of the complete text as input - note the length of each must be exactly equal to the window size. For each subset use the function above to predict the next 100 characters that follow each input.

Sequence generator

In [32]:

```
# TODO: choose an input sequence and use the prediction function in the
previous
# Python cell to predict 100 characters following it
# get an appropriately sized chunk of characters from the text
start_inds = [1000, 2000]

# load in weights
model.load_weights('model_weights/best_RNN_small_textdata_weights.hdf5')
for s in start_inds:
    start_index = s
    input_chars = text[start_index: start_index + window_size]

    # use the prediction function
    predict_input = predict_next_chars(model, input_chars, num_to_predict=100)
    # print out input characters
    print('------------------')
    input_line = 'input chars = ' + '\n' +  input_chars + '"' + '\n'
    print(input_line)

    # print out predicted characters
    line = 'predicted chars = ' + '\n' +  predict_input + '"' + '\n'
    print(line)
```

```
------------------
input chars =
er, of dubious and questionable memory. i had seen little of holmes lately. my
marriage had drifted "

predicted chars =
the has stonte the have the mas ing and coured ou has in the sert of the coupt
of the couch has sour"

------------------
input chars =
f the singular tragedy of the atkinson brothers at trincomalee, and finally of
the mission which he "

predicted chars =
hoom shere sour ho has the four has sour and coure south a sert of the hous have
the coust of the co"
```

This looks ok, but not great. Now lets try the same experiment with a larger chunk of the data - with the first 100,000 input/output pairs.

Tuning RNNs for a typical character dataset like the one we will use here is a computationally intensive endeavour and thus timely on a typical CPU. Using a reasonably sized cloud-based GPU can speed up training by a factor of 10. Also because of the long training time it is highly recommended that you carefully write the output of each step of your process to file. This is so that all of your results are saved even if you close the web browser you're working out of, as the processes will continue processing in the background but variables/output in the notebook system will not update when you open it again.

Sequence generator

In the next cell we show you how to create a text file in Python and record data to it. This sort of setup can be used to record your final predictions.

In [33]:

```
### A simple way to write output to file
f = open('my_test_output.txt', 'w')              # create an output file to
write too
f.write('this is only a test ' + '\n')           # print some output text
x = 2
f.write('the value of x is ' + str(x) + '\n')    # record a variable value
f.close()

# print out the contents of my_test_output.txt
f = open('my_test_output.txt', 'r')              # create an output file to
write too
f.read()
```

Out[33]:

```
'this is only a test \nthe value of x is 2\n'
```

With this recording devices we can now more safely perform experiments on larger portions of the text. In the next cell we will use the first 100,000 input/output pairs to train our RNN model.
First we fit our model to the dataset, then generate text using the trained model in precisely the same generation method applied before on the small dataset.

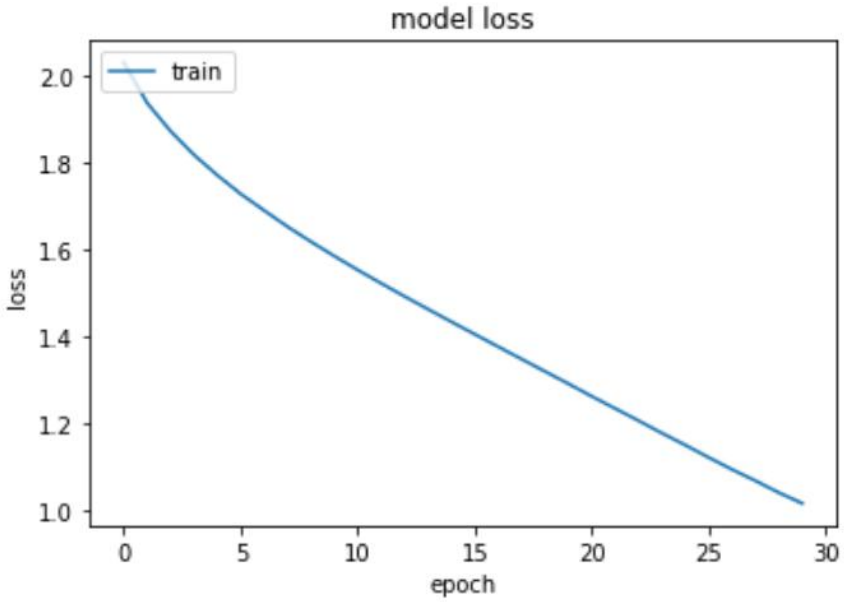
Note: your generated words should be - by and large - more realistic than with the small dataset, but you won't be able to generate perfect English sentences even with this amount of data. A rule of thumb: your model is working well if you generate sentences that largely contain real English words.

In [34]:

```
# a small subset of our input/output pairs
Xlarge = X[:100000,:,:]
ylarge = y[:100000,:]

# TODO: fit to our larger dataset
epochtimer = EpochTimer()
epochs = 30

hist = model.fit(Xlarge, ylarge, batch_size=500, epochs=epochs, verbose=1,
callbacks=[epochtimer])
show_history_graph(hist)

# save weights
model.save_weights('model_weights/best_RNN_large_textdata_weights.hdf5')
```

```
Epoch 1/30
 99500/100000 [============================>.] - ETA: 0s - loss: 2.0280Epoch 0
took 43.5252470409996 seconds
100000/100000 [==============================] - 43s - loss: 2.0279
Epoch 2/30
 99500/100000 [============================>.] - ETA: 0s - loss: 1.9358Epoch 1
took 43.47008210199965 seconds
……
Epoch 30/30
 99500/100000 [============================>.] - ETA: 0s - loss: 1.0141Epoch 29
took 43.316759814999386 seconds
100000/100000 [==============================] - 43s - loss: 1.0143
Training took 1300.820223281 seconds
```

Sequence generator



In [43]:

```
# TODO: choose an input sequence and use the prediction function in the
previous Python cell to predict 100 characters following it
# get an appropriately sized chunk of characters from the text
start_inds = [1000, 2000, 3000, 4000, 5000]
window_size = 100
# save output
f = open('text_gen_output/RNN_large_textdata_output.txt', 'w')  # create an
output file to write too

# load weights
model.load_weights('model_weights/best_RNN_large_textdata_weights.hdf5')
for s in start_inds:
    start_index = s
    input_chars = text[start_index: start_index + window_size]

    # use the prediction function
    predict_input = predict_next_chars(model, input_chars, num_to_predict =
100)
    # print out input characters
    line = '-------------------' + '\n'
    print(line)
    f.write(line)

    input_line = 'input chars = ' + '\n' +  input_chars + '"' + '\n'
    print(input_line)
    f.write(input_line)

    # print out predicted characters
    predict_line = 'predicted chars = ' + '\n' +  predict_input + '"' + '\n'
    print(predict_line)
    f.write(predict_line)
f.close()
```

Sequence generator

input chars =

```
er, of dubious and questionable memory. i had seen little of holmes lately. my
marriage had drifted "
```

predicted chars =

```
up the papers which lee had no doutt to he hin do so an ourse of the string of
the waster could holm"
```

---

input chars =

```
f the singular tragedy of the atkinson brothers at trincomalee, and finally of
the mission which he "
```

predicted chars =

```
had been seent of the corner of the bark in his head.  i am surdreate. then, and
i was alrouse of th"
```

---

input chars =

```
ands clasped behind him. to me, who knew his every mood and habit, his attitude
and manner told thei"
```

predicted chars =

```
rr about it.  well, street of you confursed said of the charrracted it was a
bearal coult rand his h"
```

---

input chars =

```
uce it. how do i know that you have been getting yourself very wet lately, and
that you have a most "
```

predicted chars =

```
fiend that i would gat that i had been about a tom the mat in she lave his and
in the man who had ha"
```

---

input chars =

```
ou had a particularly malignant boot slitting specimen of the london slavey. as
to your practice, if"
```

predicted chars =

```
 i was alrised that it was a some and a stare from and a woundssed had been
didet.  the first can a "
```

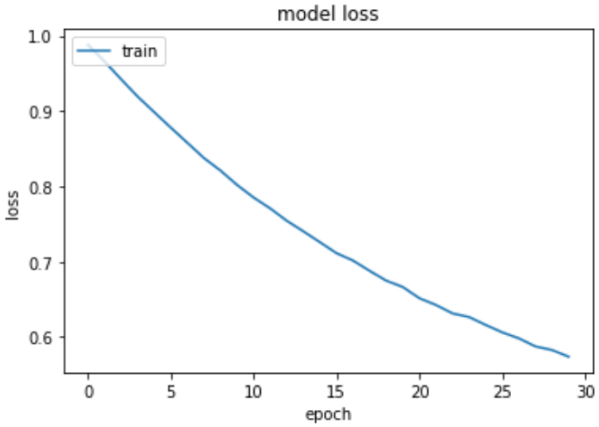
In [44]:

```
# TODO: fit to our larger dataset
epochtimer = EpochTimer()
epochs = 30

hist = model.fit(Xlarge, ylarge, batch_size=500, epochs=epochs, verbose=1,
callbacks=[epochtimer])
show_history_graph(hist)

# save weights
model.save_weights('model_weights/best_RNN_large_textdata_weights_again.hdf5')
```

```
Epoch 1/30
 99500/100000 [============================>.] - ETA: 0s - loss: 0.9875Epoch 0
took 43.530050407998715 seconds
100000/100000 [==============================] - 43s - loss: 0.9880
Epoch 2/30
```

Sequence generator

```
 99500/100000 [============================>.] - ETA: 0s - loss: 0.9653Epoch 1
took 43.43311494500085 seconds
……..

Epoch 30/30
 99500/100000 [============================>.] - ETA: 0s - loss: 0.5733Epoch 29
took 43.47800101999928 seconds
100000/100000 [==============================] - 43s - loss: 0.5739
Training took 1302.8925141510008 seconds
```



In [47]:

```
# TODO: choose an input sequence and use the prediction function in the
previous Python cell to predict 100 characters following it
# get an appropriately sized chunk of characters from the text
start_inds = [1000, 2000, 3000, 4000, 5000]
window_size = 100
# save output
f = open('text_gen_output/RNN_large_textdata_output_again.txt', 'w')  # create
an output file to write too

# load weights
model.load_weights('model_weights/best_RNN_large_textdata_weights_again.hdf5')
for s in start_inds:
    start_index = s
    input_chars = text[start_index: start_index + window_size]

    # use the prediction function
    predict_input = predict_next_chars(model, input_chars, num_to_predict =
100)

    # print out input characters
    line = '-------------------' + '\n'
    print(line)
    f.write(line)

    input_line = 'input chars = ' + '\n' +  input_chars + '"' + '\n'
    print(input_line)
    f.write(input_line)


    # print out predicted characters
```

Sequence generator

```
    predict_line = 'predicted chars = ' + '\n' +  predict_input + '"' + '\n'
    print(predict_line)
    f.write(predict_line)
f.close()
```

```
input chars =
er, of dubious and questionable memory. i had seen little of holmes lately. my marriage had drifted "
predicted chars =
up the sair. to the corountry was and a scoppections of the excuppession of his freesous doss of the"
------------------
input chars =
f the singular tragedy of the atkinson brothers at trincomalee, and finally of the mission which he "
predicted chars =
headd. still, holmes have been herenes, and well, brown emon the wasked and the corners for ter itso"
------------------
input chars =
ands clasped behind him. to me, who knew his every mood and habit, his attitude and manner told thei"
predicted chars =
ll, and he reilad dores, for it betweed the tomethe to ct. she light an oller three mest per of at t"
------------------
input chars =
uce it. how do i know that you have been getting yourself very wet lately, and that you have a most "
predicted chars =
thisherest that i will be to stall i believely becomerel,  and the lotter was which thrugh were serp"
------------------
input chars =
ou had a particularly malignant boot slitting specimen of the london slavey. as to your practice, if"
predicted chars =
 never what the came were remorded the loor. a gray was shoul for ack about here and save might her "
```

In [50]:

```
Xlarge_more = X[:200000,:,:]
ylarge_more = y[:200000,:]
print(len(Xlarge))
print(len(ylarge))
print(len(Xlarge_more))
print(len(ylarge_more))
```

```
100000
100000
114717
114717
```

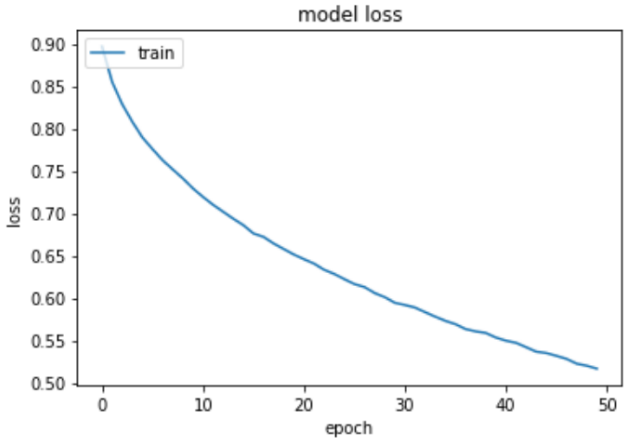
Sequence generator

In [51]:

```
# TODO: fit to our larger dataset
epochtimer = EpochTimer()
epochs = 50

hist = model.fit(Xlarge_more, ylarge_more, batch_size=500, epochs=epochs,
verbose=1, callbacks=[epochtimer])
show_history_graph(hist)

# save weights
model.save_weights('model_weights/best_RNN_large_textdata_weights_more.hdf5')
```

```
Epoch 1/50
114500/114717 [============================>.] - ETA: 0s - loss: 0.8963Epoch 0
took 50.00066477800101 seconds
114717/114717 [==============================] - 50s - loss: 0.8967
Epoch 2/50
114500/114717 [============================>.] - ETA: 0s - loss: 0.8549Epoch 1
took 49.85689944600017 seconds
```

```
Epoch 50/50
114500/114717 [============================>.] - ETA: 0s - loss: 0.5165Epoch 49
took 49.88088778800011 seconds
114717/114717 [==============================] - 49s - loss: 0.5166
Training took 2494.2848371189993 seconds
```



In [52]:

```
# TODO: choose an input sequence and use the prediction function in the
previous Python cell to predict 100 characters following it
# get an appropriately sized chunk of characters from the text
start_inds = [1000, 2000, 3000, 4000, 5000]
window_size = 100
# save output
f = open('text_gen_output/RNN_large_textdata_output_more.txt', 'w')  # create
an output file to write too
```

```
# load weights
model.load_weights('model_weights/best_RNN_large_textdata_weights_more.hdf5')
```

Sequence generator

```
for s in start_inds:
    start_index = s
    input_chars = text[start_index: start_index + window_size]

    # use the prediction function
    predict_input = predict_next_chars(model, input_chars, num_to_predict =
100)

    # print out input characters
    line = '-------------------' + '\n'
    print(line)
    f.write(line)

    input_line = 'input chars = ' + '\n' +  input_chars + '"' + '\n'
    print(input_line)
    f.write(input_line)

    # print out predicted characters
    predict_line = 'predicted chars = ' + '\n' +  predict_input + '"' + '\n'
    print(predict_line)
    f.write(predict_line)
f.close()
```

```
-------------------
input chars =
er, of dubious and questionable memory. i had seen little of holmes lately. my
marriage had drifted "
```