

# Artificial Intelligence Nanodegree

## Recurrent Neural Network Projects

Welcome to the Recurrent Neural Network Project in the Artificial Intelligence Nanodegree! In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with '**Implementation**' in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a '**Question X**' header. Carefully read each question and provide thorough answers in the following text boxes that begin with '**Answer:**'. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

**Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. In addition, Markdown cells can be edited by typically double-clicking the cell to enter edit mode.

## Implementation TODOs in this notebook

This notebook contains two problems, cut into a variety of TODOs. Make sure to complete each section containing a TODO marker throughout the notebook. For convenience we provide links to each of these sections below.

[TODO #1: Implement a function to window time series](#)

[TODO #2: Create a simple RNN model using keras to perform regression](#)

[TODO #3: Finish cleaning a large text corpus](#)

[TODO #4: Implement a function to window a large text corpus](#)

[TODO #5: Create a simple RNN model using keras to perform multiclass classification](#)

[TODO #6: Generate text using a fully trained RNN model and a variety of input sequences](#)

## Problem 1: Perform time series prediction

In this project you will perform time series prediction using a Recurrent Neural Network regressor. In particular you will re-create the figure shown in the notes - where the stock price of Apple was forecasted (or predicted) 7 days in advance. In completing this exercise you will learn how to construct RNNs using Keras, which will also aid in completing the second project in this notebook.

The particular network architecture we will employ for our RNN is known as Long Term Short Memory (LSTM) ([https://en.wikipedia.org/wiki/Long\\_short-term\\_memory](https://en.wikipedia.org/wiki/Long_short-term_memory)), which helps significantly avoid technical problems with optimization of RNNs.

### 1.1 Getting started

First we must load in our time series - a history of around 140 days of Apple's stock price. Then we need to perform a number of pre-processing steps to prepare it for use with an RNN model. First off, it is good practice to normalize time series - by normalizing its range. This helps us avoid serious numerical issues associated how common activation functions (like tanh) transform very large (positive or negative) numbers, as well as helping us to avoid related issues when computing derivatives.

Here we normalize the series to lie in the range [0,1] using this scikit function (<http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html>), but it is also commonplace to normalize by a series standard deviation.

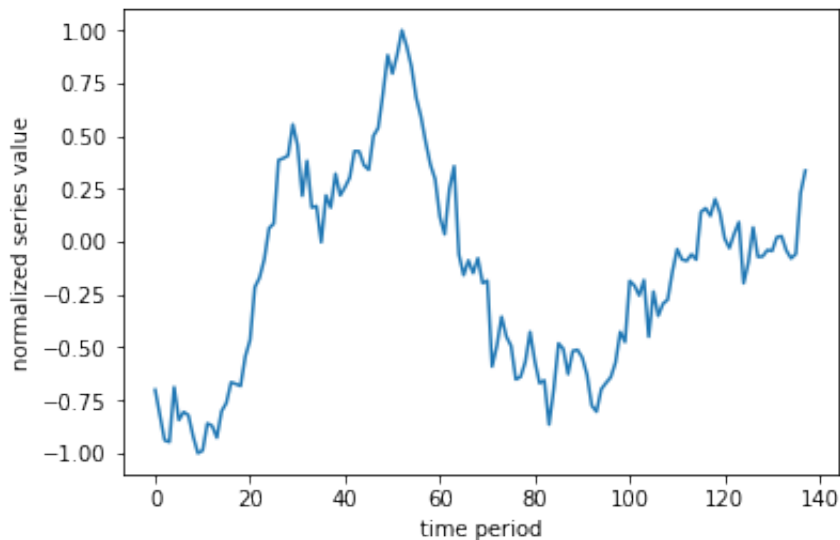
```
In [1]: ### Load in necessary libraries for data input and normalization
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

### load in and normalize the dataset
dataset = np.loadtxt('datasets/normalized_apple_prices.csv')
```

Lets take a quick look at the (normalized) time series we'll be performing predictions on.

```
In [2]: # lets take a look at our time series
plt.plot(dataset)
plt.xlabel('time period')
plt.ylabel('normalized series value')
```

Out[2]: <matplotlib.text.Text at 0x7fb32a3ce198>

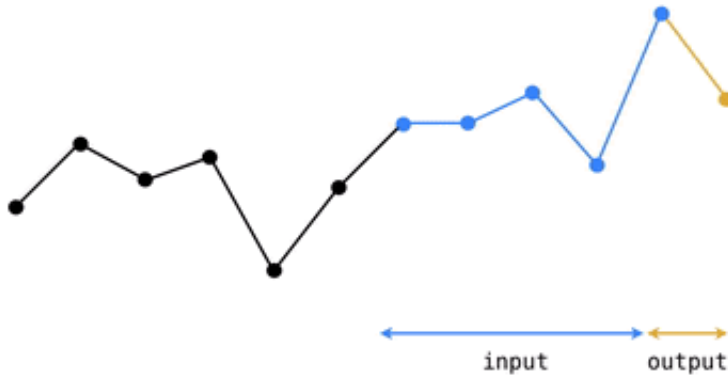


## 1.2 Cutting our time series into sequences

Remember, our time series is a sequence of numbers that we can represent in general mathematically as

$$s_0, s_1, s_2, \dots, s_P$$

where  $s_p$  is the numerical value of the time series at time period  $p$  and where  $P$  is the total length of the series. In order to apply our RNN we treat the time series prediction problem as a regression problem, and so need to use a sliding window to construct a set of associated input/output pairs to regress on. This process is animated in the gif below.



For example - using a window of size  $T = 5$  (as illustrated in the gif above) we produce a set of input/output pairs like the one shown in the table below

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 length 4 (and in general has length equal to the window size  $T$ ) while each corresponding output is a scalar value. Notice also how given a time series of length  $P$  and window size  $T = 5$  as shown above, we created  $P - 5$  input/output pairs. More generally, for a window size  $T$  we create  $P - T$  such pairs.

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

**TODO:** Fill in the function below - called **window\_transform\_series** - that runs a sliding window along the input series and creates associated input/output pairs. Note that this function should input a) the series and b) the window length, and return the input/output subsequences. Make sure to format returned input/output as generally shown in table above (where window\_size = 5), and make sure your returned input is a numpy array.

---

You can test your function on the list of odd numbers given below

```
In [3]: odd_nums = np.array([1,3,5,7,9,11,13])
```

To window this sequence with a window\_size = 2 using the **window\_transform\_series** you should get the following input/output pairs

```

In [35]: # run a window of size 2 over the odd number sequence and display the
         results
         window_size = 2
         X,y = window_transform_series(odd_nums,window_size)

         # print out input/output pairs --> here input = X, corresponding output = y
         print ('--- the input X will look like ----')
         print (X)

         print ('--- the associated output y will look like ----')
         print (y)

         print ('the shape of X is ' + str(np.shape(X)))
         print ('the shape of y is ' + str(np.shape(y)))
         print('the type of X is ' + str(type(X)))
         print('the type of y is ' + str(type(y)))

--- the input X will look like ----
[[ 1  3]
 [ 3  5]
 [ 5  7]
 [ 7  9]
 [ 9 11]]
--- the associated output y will look like ----
[[ 5]
 [ 7]
 [ 9]
 [11]
 [13]]
the shape of X is (5, 2)
the shape of y is (5, 1)
the type of X is <class 'numpy.ndarray'>
the type of y is <class 'numpy.ndarray'>

```

Again - you can check that your completed **window\_transform\_series** function works correctly by trying it on the odd\_nums sequence - you should get the above output.

(remember to copy your completed function into the script *my\_answers.py* function titled *window\_transform\_series* before submitting your project)

```
In [5]: ### TODO: fill out the function below that transforms the input series
and window-size into a set of input/output pairs for use with our RNN
model
def window_transform_series(series,window_size):
    # containers for input/output pairs
    X = [series[i:i+window_size] for i in range(len(series) - window_s
ize)]
    y = series[window_size:]

    # reshape each
    X = np.asarray(X)
    X.shape = (np.shape(X)[0:2])
    y = np.asarray(y)
    y.shape = (len(y),1)

    return X,y
```

With this function in place apply it to the series in the Python cell below. We use a window\_size = 7 for these experiments.

```
In [6]: # window the data using your windowing function
window_size = 7
X,y = window_transform_series(series = dataset,window_size = window_si
ze)
```

## 1.3 Splitting into training and testing sets

In order to perform proper testing on our dataset we will lop off the last 1/3 of it for validation (or testing). This is that once we train our model we have something to test it on (like any regression problem!). This splitting into training/testing sets is done in the cell below.

Note how here we are **not** splitting the dataset *randomly* as one typically would do when validating a regression model. This is because our input/output pairs *are related temporally*. We don't want to validate our model by training on a random subset of the series and then testing on another random subset, as this simulates the scenario that we receive new points *within the timeframe of our training set*.

We want to train on one solid chunk of the series (in our case, the first full 2/3 of it), and validate on a later chunk (the last 1/3) as this simulates how we would predict *future* values of a time series.

```
In [7]: # split our dataset into training / testing sets
train_test_split = int(np.ceil(2*len(y)/float(3))) # set the split point

# partition the training set
X_train = X[:train_test_split,:]
y_train = y[:train_test_split]

# keep the last chunk for testing
X_test = X[train_test_split:,:]
y_test = y[train_test_split:]

# NOTE: to use keras's RNN LSTM module our input must be reshaped to [
samples, window size, stepsize]
X_train = np.asarray(np.reshape(X_train, (X_train.shape[0], window_size,
1)))
X_test = np.asarray(np.reshape(X_test, (X_test.shape[0], window_size,
1)))
```

## 1.4 Build and run an RNN regression model

Having created input/output pairs out of our time series and cut this into training/testing sets, we can now begin setting up our RNN. We use Keras to quickly build a two hidden layer RNN of the following specifications

- layer 1 uses an LSTM module with 5 hidden units (note here the input\_shape = (window\_size,1))
- layer 2 uses a fully connected module with one unit
- the 'mean\_squared\_error' loss should be used (remember: we are performing regression here)

This can be constructed using just a few lines - see e.g., the [general Keras documentation \(https://keras.io/getting-started/sequential-model-guide/\)](https://keras.io/getting-started/sequential-model-guide/) and the [LSTM documentation in particular \(https://keras.io/layers/recurrent/\)](https://keras.io/layers/recurrent/) for examples of how to quickly use Keras to build neural network models. Make sure you are initializing your optimizer given the [keras-recommended approach for RNNs \(https://keras.io/optimizers/\)](https://keras.io/optimizers/)

(given in the cell below). (remember to copy your completed function into the script *my\_answers.py* function titled *build\_part1\_RNN* before submitting your project)



```
In [8]: ### TODO: create required RNN model
# import keras network libraries
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM, Embedding, Dropout
import keras

# given - fix random seed - so we can all reproduce the same results o
n our default time series
np.random.seed(0)

# TODO: build an RNN to perform regression on our time series input/ou
tput data
model = Sequential()
model.add(Dense(128, activation='softmax', input_shape = (7,1)))
model.add(LSTM(128))
model.add(Dropout(0.5))
model.add(Dense(1, activation='softmax'))
model.summary()

# build model using keras documentation recommended optimizer initiali
zation
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08,
decay=0.0)

# compile the model
model.compile(loss='mean_squared_error', optimizer=optimizer)
```

Using TensorFlow backend.

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 7, 128)	256
lstm_1 (LSTM)	(None, 128)	131584
dropout_1 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 1)	129
Total params: 131,969		
Trainable params: 131,969		
Non-trainable params: 0		

With your model built you can now fit the model by activating the cell below! Note: the number of epochs (np\_epochs) and batch\_size are preset (so we can all produce the same results). You can choose to toggle the verbose parameter - which gives you regular updates on the progress of the algorithm - on and off by setting it to 1 or 0 respectively.

```
In [9]: # run your model!  
model.fit(X_train, y_train, epochs=1000, batch_size=50, verbose=0)
```

```
Out[9]: <keras.callbacks.History at 0x7fb32a3dee80>
```

## 1.5 Checking model performance

With your model fit we can now make predictions on both our training and testing sets.

```
In [10]: # generate predictions for training  
train_predict = model.predict(X_train)  
test_predict = model.predict(X_test)
```

In the next cell we compute training and testing errors using our trained model - you should be able to achieve at least

*training\_error* < 0.02

and

*testing\_error* < 0.02

with your fully trained model.

If either or both of your accuracies are larger than 0.02 re-train your model - increasing the number of epochs you take (a maximum of around 1,000 should do the job) and/or adjusting your batch\_size.

```
In [11]: # print out training and testing errors  
training_error = model.evaluate(X_train, y_train, verbose=0)  
print('training error = ' + str(training_error))  
  
testing_error = model.evaluate(X_test, y_test, verbose=0)  
print('testing error = ' + str(testing_error))  
  
training error = 1.53932051225  
testing error = 1.28897686615
```

Activating the next cell plots the original data, as well as both predictions on the training and testing sets.

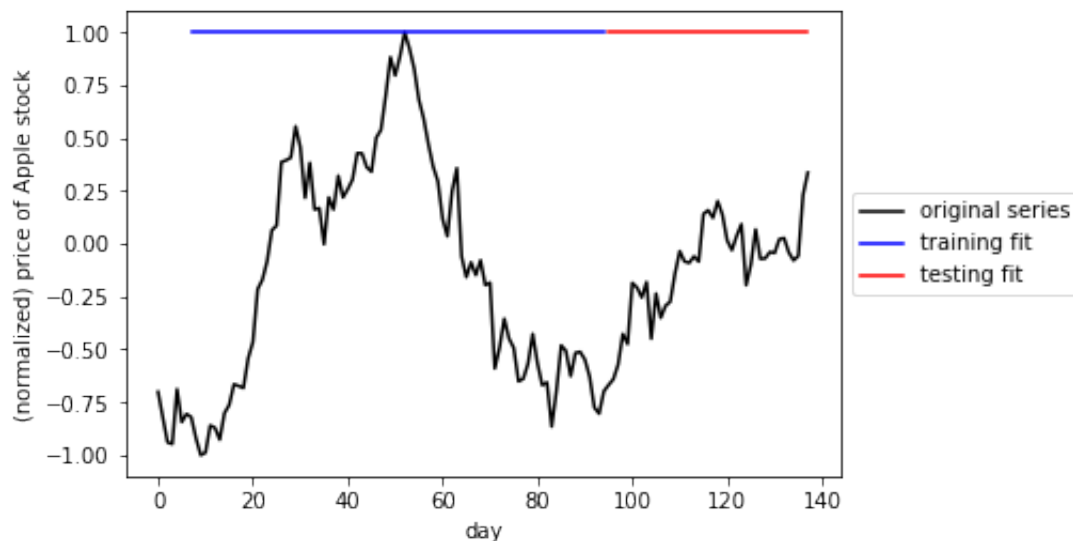
```
In [12]: ### Plot everything - the original series as well as predictions on training and testing sets
import matplotlib.pyplot as plt
%matplotlib inline

# plot original series
plt.plot(dataset,color = 'k')

# plot training set prediction
split_pt = train_test_split + window_size
plt.plot(np.arange(window_size,split_pt,1),train_predict,color = 'b')

# plot testing set prediction
plt.plot(np.arange(split_pt,split_pt + len(test_predict),1),test_predict,color = 'r')

# pretty up graph
plt.xlabel('day')
plt.ylabel('(normalized) price of Apple stock')
plt.legend(['original series','training fit','testing fit'],loc='center left', bbox_to_anchor=(1, 0.5))
plt.show()
```



**Note:** you can try out any time series for this exercise! If you would like to try another see e.g., [this site](https://datamarket.com/data/list/?q=provider%3Atsd) containing thousands of time series (<https://datamarket.com/data/list/?q=provider%3Atsd>) and pick another one!

# Problem 2: Create a sequence generator

## 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 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*

while 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 (<http://www.cs.toronto.edu/~ilya/rnn.html>) with this academic RNN text generator
- Various twitter bots that tweet automatically generated text like this one (<http://tweet-generator-alex.herokuapp.com/>).
- the NanoGenMo (<https://github.com/NaNoGenMo/2016>) annual contest to automatically produce a 50,000+ novel automatically
- Robot Shakespeare (<https://github.com/genekogan/RobotShakespeare>) 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 [13]: # 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, let's 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 junk characters that aren't words!

```
In [14]: ### print out the first 1000 characters of the raw text to get a sense
of what we need to throw out
text[:2000]
```

```
Out[14]: "\uffffproject gutenbergs the adventures of sherlock holmes, by art
hur conan doyle\n\nthis ebook is for the use of anyone anywhere at n
o cost and with\nalmost no restrictions whatsoever. you may copy it
, give it away or\nre-use it under the terms of the project gutenber
g 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: n
ovember 29, 2002\n\nlanguage: english\n\n\n*** start of this project
gutenberg ebook the adventures of sherlock holmes ***\n\n\n\n\nprodu
ced by an anonymous project gutenberg volunteer and jose menendez\n
\n\n\n\n\n\n\nthe adventures of sherlock holmes\n\nby\n\nsir arth
ur conan doyle\n\n\n\n\n i. a scandal in bohemia\n ii. the red-head
ed league\n iii. a case of identity\n iv. the boscombe valley myste
ry\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. th
e 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\n\nadventur
e 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. i
t was not that he felt\nany emotion akin to love for irene adler. al
l emotions, and that\nnone particularly, were abhorrent to his cold,
precise but\nadmirably balanced mind. he was, i take it, the most pe
rfect\nreasoning and observing machine that the world has seen, but
as a\nlover he would have placed himself in a false position. he nev
er\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\n
to 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)! e.g., all the carriage return and newline sequences '\n' and '\r' sequences. We want to train our RNN on a large chunk of real 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 [15]: ### 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 [16]: ### print out the first 1000 characters of the raw text to get a sense of what we need to throw out
text[:1000]
```

```
Out[16]: "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. 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"
```

**TODO: finish cleaning the text**

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

(remember to copy your completed function into the script *my\_answers.py* function titled *clean\_text* before submitting your project)

```
In [17]: ### TODO: list all unique characters in the text and remove any non-english ones
# find all unique characters in the text

import string

# remove as many non-english characters and character sequences as you can

for punc in string.punctuation:
    if punc == ',' or punc == '.':
        continue
    text = text.replace(punc, ' ')

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

```
Out[18]: '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 emoti  
ons, and that one particularly, were abhorrent to his cold, precise  
but admirably balanced mind. he was, i take it, the most perfect rea  
soning 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 admirab  
le things for the observer excellent for drawing the veil from men s  
motives and actions. but for the trained reasoner to admit such intr  
usions into his own delicate and finely adjusted temperament was to  
introduce a distracting factor which might throw a doubt upon all hi  
s 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 st  
rong emotion in a nature such as his. and yet there was but one woma  
n to him, and that woman was the late irene adler, of dubious and qu  
estionable memory. i had seen little of holmes lately. my marriage h  
ad drifted us away from each other. my own complete happiness, and t  
he home centred interests which rise up around the man who first fin  
ds himself master of his own establishment, were sufficient to absor  
b all my attention, while holmes, who loathed every form of society  
with his whole bohemian soul, remained in our lodgings in baker stre  
et, buried among his old books, and alternating from week to week be  
tween cocaine and ambition, the drowsiness of the drug, and the fier  
ce energy of his own keen nature. he was still, as ever, deeply attr  
acted by the study of crime, and occupied his immense faculties and  
extraordinary powers of observation in following out those clues, an  
d clearing up those mysteries which had been abandoned as hopeless b  
y 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 mu  
rder, of his clearing up of'
```

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 [19]: # 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 573376 total number of characters
this corpus has 43 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  $T = 5$ .

d o g s   a r e   g r e a t

Notice one aspect of the sliding window in this gif that does not mirror the analogous 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 let's denote our text corpus - which is one long string of characters - as follows

$$s_0, s_1, s_2, \dots, s_P$$

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  $\text{ceil}(P/M)$  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.

(remember to copy your completed function into the script *my\_answers.py* function titled *window\_transform\_text* before submitting your project)

```
In [20]: ### TODO: fill out the function below that transforms the input text and window-size into a set of input/output pairs for use with our RNN model
def window_transform_text(text, window_size, step_size):
    # containers for input/output pairs
    inputs = [text[i:i+window_size] for i in range(len(text) - window_size)][:step_size]
    outputs = text[window_size::step_size]

    return inputs, outputs
```

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 = 50 and step\_size = 5.

```
In [21]: # 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 [22]: # 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 act
ions. 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 [23]: # 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 43 unique characters
and these characters are
[' ', '.', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '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', 'à', 'â',
 'è', 'é']
```

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

$$a \leftarrow \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \quad b \leftarrow \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \quad c \leftarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \dots$$

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 [24]: # this dictionary is a function mapping each unique character to a unique integer
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 **X**, and numerical output is given as the **y**

```
In [25]: # 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
    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 transform 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 [26]: # 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 LSTM 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 \(https://keras.io/getting-started/sequential-model-guide/\)](https://keras.io/getting-started/sequential-model-guide/) and the [LSTM documentation in particular \(https://keras.io/layers/recurrent/\)](https://keras.io/layers/recurrent/) for examples of how to quickly use Keras to build neural network models.

```
In [27]: ### 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 build the required RNN model: a single LSTM hidden layer with s
oftmax activation, categorical_crossentropy loss

model = Sequential()
model.add(LSTM(200, input_shape = (window_size, len(chars))) )
model.add(Dense(len(chars), activation = 'softmax' ))

# 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)
```

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

Now lets fit our model!

```
In [29]: # train the model
model.fit(Xsmall, ysmall, batch_size=500, epochs=40, verbose = 1)

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

```
Epoch 1/40
10000/10000 [=====] - 4s - loss: 3.0617
Epoch 2/40
10000/10000 [=====] - 4s - loss: 2.8763
Epoch 3/40
10000/10000 [=====] - 4s - loss: 2.8563
Epoch 4/40
10000/10000 [=====] - 4s - loss: 2.8216
Epoch 5/40
10000/10000 [=====] - 4s - loss: 2.7773
Epoch 6/40
10000/10000 [=====] - 4s - loss: 2.7184
Epoch 7/40
10000/10000 [=====] - 4s - loss: 2.6580
Epoch 8/40
10000/10000 [=====] - 4s - loss: 2.5936
Epoch 9/40
10000/10000 [=====] - 4s - loss: 2.5338
Epoch 10/40
10000/10000 [=====] - 4s - loss: 2.4811
Epoch 11/40
10000/10000 [=====] - 4s - loss: 2.4411
Epoch 12/40
10000/10000 [=====] - 4s - loss: 2.4062
Epoch 13/40
10000/10000 [=====] - 4s - loss: 2.3706
Epoch 14/40
```

```
10000/10000 [=====] - 4s - loss: 2.3421
Epoch 15/40
10000/10000 [=====] - 4s - loss: 2.3150
Epoch 16/40
10000/10000 [=====] - 4s - loss: 2.2858
Epoch 17/40
10000/10000 [=====] - 4s - loss: 2.2686
Epoch 18/40
10000/10000 [=====] - 4s - loss: 2.2462
Epoch 19/40
10000/10000 [=====] - 4s - loss: 2.2293
Epoch 20/40
10000/10000 [=====] - 4s - loss: 2.2037
Epoch 21/40
10000/10000 [=====] - 4s - loss: 2.1947
Epoch 22/40
10000/10000 [=====] - 4s - loss: 2.1749
Epoch 23/40
10000/10000 [=====] - 4s - loss: 2.1591
Epoch 24/40
10000/10000 [=====] - 4s - loss: 2.1432
Epoch 25/40
10000/10000 [=====] - 4s - loss: 2.1223
Epoch 26/40
10000/10000 [=====] - 4s - loss: 2.1095
Epoch 27/40
10000/10000 [=====] - 4s - loss: 2.0948
Epoch 28/40
10000/10000 [=====] - 4s - loss: 2.0837
Epoch 29/40
10000/10000 [=====] - 4s - loss: 2.0611
Epoch 30/40
10000/10000 [=====] - 4s - loss: 2.0457
Epoch 31/40
10000/10000 [=====] - 4s - loss: 2.0325
Epoch 32/40
10000/10000 [=====] - 4s - loss: 2.0192
Epoch 33/40
10000/10000 [=====] - 4s - loss: 2.0009
Epoch 34/40
10000/10000 [=====] - 4s - loss: 1.9898
Epoch 35/40
10000/10000 [=====] - 4s - loss: 1.9680
Epoch 36/40
10000/10000 [=====] - 4s - loss: 1.9585
Epoch 37/40
10000/10000 [=====] - 4s - loss: 1.9332
Epoch 38/40
10000/10000 [=====] - 4s - loss: 1.9229
Epoch 39/40
```



```

10000/10000 [=====] - 4s - loss: 1.9035
Epoch 40/40
10000/10000 [=====] - 4s - loss: 1.8858

```

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 [30]: # 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
        t 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 us the function above to predict the next 100 characters that follow each input.

```
In [31]: # 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]

# 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 there ard and and and and and and and and and and and and an
d and and and and and and and and an"
```

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.

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 [32]: ### A simple way to write output to file
         f = open('my_test_output.txt', 'w')           # create an output fi
         le to write too
         f.write('this is only a test ' + '\n')        # print some output t
         ext
         x = 2
         f.write('the value of x is ' + str(x) + '\n')  # record a variable v
         alue
         f.close()

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

```
Out[32]: '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 [33]: # a small subset of our input/output pairs
Xlarge = X[:100000,:,:]
ylarge = y[:100000,:]

# TODO: fit to our larger dataset
model.fit(Xlarge, ylarge, batch_size=500, nb_epoch=30, verbose = 1)

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

Epoch 1/30

/home/aind2/anaconda3/envs/aind2/lib/python3.6/site-packages/keras/models.py:837: UserWarning: The `nb\_epoch` argument in `fit` has been renamed `epochs`.

warnings.warn('The `nb\_epoch` argument in `fit` '

100000/100000 [=====] - 43s - loss: 2.0472

Epoch 2/30

100000/100000 [=====] - 43s - loss: 1.9652

Epoch 3/30

100000/100000 [=====] - 43s - loss: 1.9027

Epoch 4/30

100000/100000 [=====] - 43s - loss: 1.8499

Epoch 5/30

100000/100000 [=====] - 43s - loss: 1.8014

Epoch 6/30

100000/100000 [=====] - 43s - loss: 1.7584

Epoch 7/30

100000/100000 [=====] - 43s - loss: 1.7179

Epoch 8/30

100000/100000 [=====] - 43s - loss: 1.6822

Epoch 9/30

100000/100000 [=====] - 43s - loss: 1.6473

Epoch 10/30

100000/100000 [=====] - 43s - loss: 1.6147

Epoch 11/30

100000/100000 [=====] - 43s - loss: 1.5834

Epoch 12/30

100000/100000 [=====] - 43s - loss: 1.5530

Epoch 13/30

100000/100000 [=====] - 43s - loss: 1.5241

Epoch 14/30

100000/100000 [=====] - 43s - loss: 1.4955

Epoch 15/30

100000/100000 [=====] - 43s - loss: 1.4680

Epoch 16/30

100000/100000 [=====] - 43s - loss: 1.4399

Epoch 17/30

```
100000/100000 [=====] - 43s - loss: 1.4122
Epoch 18/30
100000/100000 [=====] - 43s - loss: 1.3859
Epoch 19/30
100000/100000 [=====] - 43s - loss: 1.3581
Epoch 20/30
100000/100000 [=====] - 43s - loss: 1.3318
Epoch 21/30
100000/100000 [=====] - 43s - loss: 1.3057
Epoch 22/30
100000/100000 [=====] - 43s - loss: 1.2794
Epoch 23/30
100000/100000 [=====] - 43s - loss: 1.2535
Epoch 24/30
100000/100000 [=====] - 43s - loss: 1.2270
Epoch 25/30
100000/100000 [=====] - 43s - loss: 1.2007
Epoch 26/30
100000/100000 [=====] - 43s - loss: 1.1734
Epoch 27/30
100000/100000 [=====] - 43s - loss: 1.1487
Epoch 28/30
100000/100000 [=====] - 43s - loss: 1.1230
Epoch 29/30
100000/100000 [=====] - 43s - loss: 1.0983
Epoch 30/30
100000/100000 [=====] - 43s - loss: 1.0725
```

```
In [34]: # 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 = [10000]

         # 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()
```

-----

input chars =  
 he passage, paused immediately outside the door. then there was a loud and authoritative tap. come "

predicted chars =  
 in the some orride of the sone of his side. i have a fook for me and in the seemon of the strange fr"

In [ ]:

In [ ]: