**STOCK PRICE PREDICTOR**

**Introduction**

We always hear news regarding stock prices dropping, markets crashing and not know why this is happening. Stock prices and its fluctuations can jolt our financial systems. Lot of people have lost their earnings by not understanding the market trend and investing in companies whose stock prices went down. Stock Price Predictor (SPP) is developed with an objective to help such people. SPP can predict the future prices of a stock by learning the price trends in the past. SPP can automatically download the historical data of the company (TICKER SYMBOL) he wants to invest, and he can get the price trend in the future. The historical data of the companies are publicly made available, so anyone can access them.

The stock price data is time series. Time Series is a series of data points indexed in time order. For stock prices, it’s the everyday opening and closing prices along with the highest and lowest prices recorded for each day. By closely monitoring the price trends, we can predict what may happen in future. Instead of spending too much time monitoring the ups and downs of stock market, the user can just feed the data to SPP and SPP will do all the work. This application can be deployed as a mobile application so that the user can easily monitor the price details. In this assignment along with developing the model, experiments have been carried out to find out the optimum CPU and memory usage scenarios so that SPP can be tuned for optimum performance.

The following sections in the document will showcase the technical details of the application development and the experiments conducted.

**Data Collection**

The ground work for developing SPP started by searching for datasets. There were lot of sites available that provides historical stock prices. But downloading data from a remote source in program was one of the requirements of the assignment, more focus was put on sites that provides API’s from where I can directly download the data. Python is my preferred language for doing Machine Learning projects. So, I mainly focused on sites that provide python compatible APIs. Some of the APIs I found at first was deprecated, for eg Yahoo Finance and Google Finance API. Then I stumbled across quandl which had a python API. I registered in quandl and got an API key through which I can download dataset from their site directly using their python API. I tried data sets of many companies. For eg GOOGLE had historical data only from 2014 and EQUINX had from 2011. I selected historical stock price from Apple ticker to train the model. The date range collected was from 2010 – 2018 (present).

Coding for this assignment is done in python3.6. Below mentioned steps are performed for installing quandll API and downloading data

* pip install quandl (installation command)
* API Key - 5ybZhPgMUSgmzT7BNYmb (Registration in Quandl site is mandatory for getting the API key)
* quandl.get\_table () – command used to import data from quandl website to the program

**Attributes of the Dataset**

Open: Opening price of the stock in a day.

Close: Closing price of the stock in a day

High: Highest price of the stock in a day

Low: Lowest price of the stock in a day

Volume: Volume of stocks traded in a day

**import quandl**

**import pandas as pd**

**# add quandl API key for unrestricted**

**quandl.ApiConfig.api\_key = '5ybZhPgMUSgmzT7BNYmb'**

**# get the table for daily stock prices and,**

**# filter the table for selected tickers, columns within a time range**

**# set paginate to True because Quandl limits tables API to 10,000 rows per call**

**df = quandl.get\_table('WIKI/PRICES', ticker = ['AAPL'],**

**qopts = { 'columns': ['ticker', 'date', 'open', 'close','low', 'high', 'volume'] },**

**date = { 'gte': '2000-01-01', 'lte': '2018-10-14' },**

**paginate=True)**

**Data Collection Steps**

**Data Analysis and Pre-processing**

Data is king in Machine Learning and data preprocessing is one the most important steps in an ML project.

Data Pre-processing and Analysis Steps are detailed below.

**Step 1: See the format of data**.

describe () and head () functions in pandas library is used for this purpose.

Head () will give show the first 5 rows of the dataset. Through that we can see various attributes of dataset, their data-type etc.

**Step 2: Visualize the data**

Visualization of data is saved in .eps, .pdf and .svg formats in different folders

Python provides options to save images in the above-mentioned formats: The sample code for saving a matplotlib graph in .eps, .pdf, .svg and .png formats is provided below. Dots per Inch (DPI) is set to 1200.

**plt.savefig('./eps/open.eps', format='eps', dpi=1200)**

**plt.savefig('./svg/open.svg', format='svg', dpi=1200)**

**plt.savefig('./pdf/open.pdf', format='pdf', dpi=1200)**

In the program itself, code has been written to create separate folders in current working directory to save images of each format in different folders.

**#current directory**

**directory = os.getcwd()**

**#creating eps directory to save .eps images**

**if not os.path.exists(directory+"\\eps"):**

**os.mkdir("eps")**

**#creating pdf directory to save .pdf images**

**if not os.path.exists(directory+"\\pdf"):**

**#Creating svg directory to save .svg files**

**if not os.path.exists(directory + "\\svg"):**

**os.mkdir("svg")**

**os.makedirs("pdf")**

**To View Encapsulated Post Script (.eps) files**

In Windows 10, EPS viewer was installed to view the images in .eps files.

<https://epsviewer.org/download.aspx>

In Ubuntu GIMP can be used to view the .eps file.

**To view Supported Vector Graphics (.svg) files**

Files in .svg formats can be opened using web browsers. I used Internet Explorer and Google chrome to view the files.

**To view Portable Document Format (.pdf) files**

Adobe Acrobat Reader is used to open files with .pdf extension. In ubuntu if Adobe is not installed LibreOffice Writer can be used to open the same.

Tensorboard visualization was obtained in .svg format. SVG Crowbar Application has been used to get the tensorflow graphas in .svg format.

[**https://nytimes.github.io/svg-crowbar/**](https://nytimes.github.io/svg-crowbar/)

**Visualization of various attributes**

The distribution of each attribute in the dataset has been plotted

Below are the histograms for open, close, high and close values of stocks for apple from 2000- 2018.

We can infer from below plots that – open, close, high and low prices have a similar distribution while volume is having different. Since volume is having an entirely different distribution and it is the number of stocks sold, we will be currently dropping this attribute for training the model.

Later we can see that how the attribute influences the prediction accuracy by training the model with and without the attribute volume.

**“open” field in dataset**

**plt.plot(df['open'])**

**plt.title('Distribution of Open values')**

**plt.xlabel('year')**

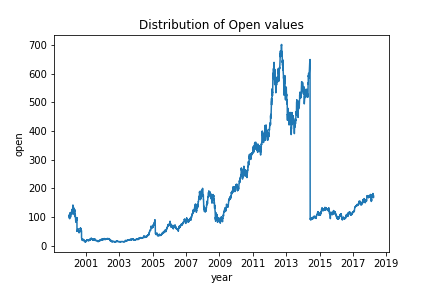
**plt.ylabel('open')**

**plt.savefig('./eps/open.eps', format='eps', dpi=1200)**

**plt.savefig('./svg/open.svg', format='svg', dpi=1200)**

**plt.savefig('./pdf/open.pdf', format='pdf', dpi=1200)**

**plt.savefig('./png/open.png', format='png', dpi=1200)**



**“Close” attribute in data set**

**plt.plot(df['close'])**

**plt.title('Distribution of Close values')**

**plt.xlabel('year')**

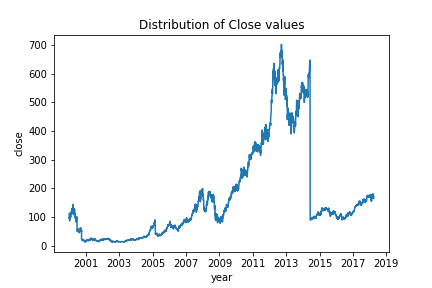
**plt.ylabel('close')**

**plt.savefig('./eps/close.eps', format='eps', dpi=1200)**

**plt.savefig('./svg/close.svg', format='svg', dpi=1200)**

**plt.savefig('./pdf/close.pdf', format='pdf', dpi=1200)**

**plt.savefig('./png/close.png', format='png', dpi=1200)**



**“low” attribute in dataset**

**plt.plot(df['low'])**

**plt.title('Distribution of Low values')**

**plt.xlabel('year')**

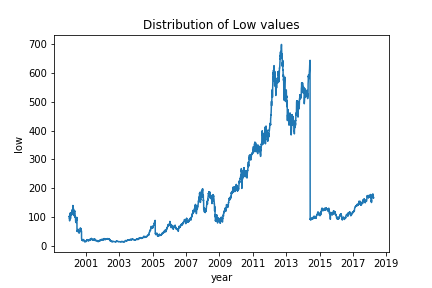
**plt.ylabel('low')**

**plt.savefig('./eps/low.eps', format='eps', dpi=1200)**

**plt.savefig('./svg/low.svg', format='svg', dpi=1200)**

**plt.savefig('./pdf/low.pdf', format='pdf', dpi=1200)**

**plt.savefig('./png/low.png', format='png', dpi=1200)**



**“high” attribute in dataset**

**plt.plot(df['high'])**

**plt.xlabel('year')**

**plt.ylabel('high')**

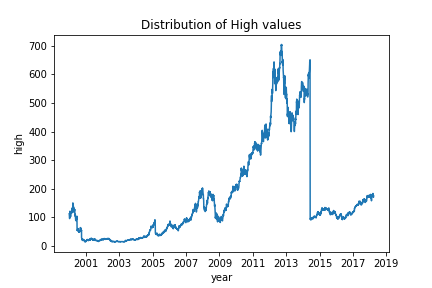
**plt.title('Distribution of High values')**

**plt.savefig('./eps/high.eps', format='eps', dpi=1200)**

**plt.savefig('./svg/high.svg', format='svg', dpi=1200)**

**plt.savefig('./pdf/high.pdf', format='pdf', dpi=1200)**

**plt.savefig('./png/high.png', format='png', dpi=1200)**



**“Volume” attribute in dataset**

**plt.plot(df['volume'])**

**plt.xlabel('year')**

**plt.ylabel('volume')**

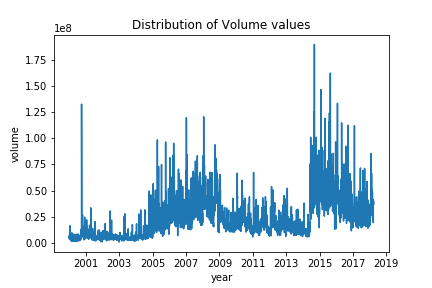
**plt.title('Distribution of Volume values')**

**plt.savefig('./eps/volume.eps', format='eps', dpi=1200)**

**plt.savefig('./svg/volume.svg', format='svg', dpi=1200)**

**plt.savefig('./pdf/volume.pdf', format='pdf', dpi=1200)**

**plt.savefig('./png/volume.png', format='png', dpi=1200)**



**Step 3: Normalizing the data**

In this step all the attributes for training the model has been put in the range of zero and one.

Please find below the graphs that shows the distribution of stock prices before and after normalization. We can see that before normalization price range was from 0-700. After normalization it’s from 0 to 1. Normalization is performed to keep the data values in a predefined range, so that it can help in training the model.

The lines for open and close are hidden behind high/low.

**“Before Normalization”**

**plt.figure(figsize=(15, 5));**

**plt.plot(df[df.ticker == 'AAPL'].open.values, color='black', label='open', linewidth = .2)**

**plt.plot(df[df.ticker == 'AAPL'].close.values, color='green', label='close')**

**plt.plot(df[df.ticker == 'AAPL'].low.values, color='blue', label='low')**

**plt.plot(df[df.ticker == 'AAPL'].high.values, color='magenta', label='high')**

**plt.title('Stock Price Distribution before Normalization')**

**plt.xlabel('time [days]')**

**plt.ylabel('price')**

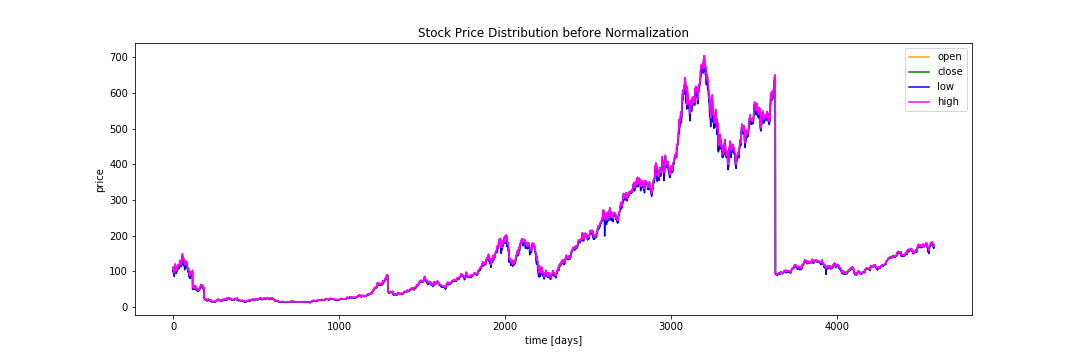
**plt.legend(loc='best')**

**plt.savefig('./eps/before\_normalization.eps', format='eps', dpi=1200)**

**plt.savefig('./svg/before\_normalization.svg', format='svg', dpi=1200)**

**plt.savefig('./pdf/before\_normalization.pdf', format='pdf', dpi=1200)**

**plt.savefig('./png/before\_normalization.png', format='png', dpi=1200)**

Before Normalization

**After Normalization**

**plt.figure(figsize=(15, 5));**

**plt.plot(df\_stock\_norm.open.values, color='violet', label='open')**

**plt.plot(df\_stock\_norm.close.values, color='green', label='close')**

**plt.plot(df\_stock\_norm.low.values, color='blue', label='low')**

**plt.plot(df\_stock\_norm.high.values, color='red', label='high')**

**#plt.plot(df\_stock\_norm.volume.values, color='gray', label='volume')**

**plt.title('Stock Price after Normalization')**

**plt.xlabel('time [days]')**

**plt.ylabel('Normalized Price')**

**plt.legend(loc='best')**

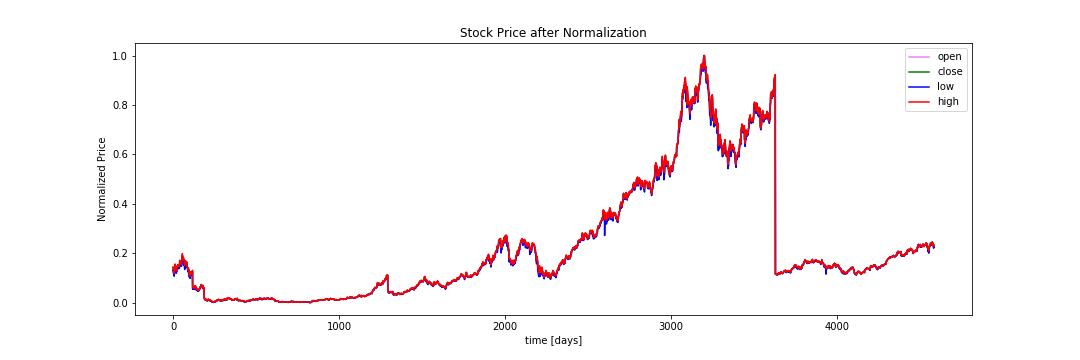
**plt.savefig('./eps/after\_normalization.eps', format='eps', dpi=1200)**

**plt.savefig('./svg/after\_normalization.svg', format='svg', dpi=1200)**

**plt.savefig('./pdf/after\_normalization.pdf', format='pdf', dpi=1200)**

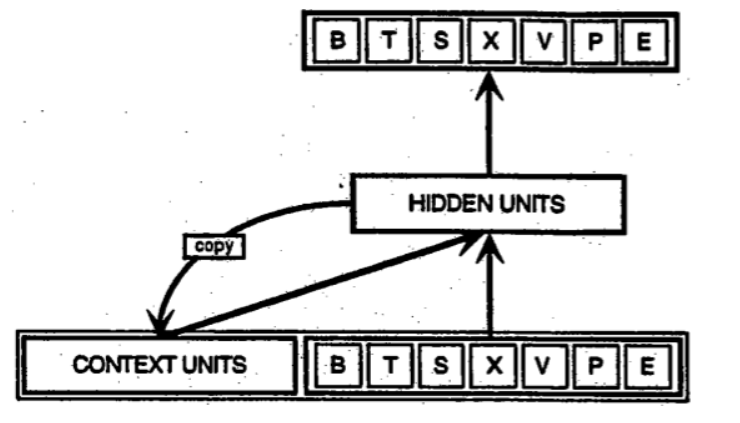
**plt.savefig('./png/after\_normalization.png', format='png', dpi=1200)**

**plt.show()**



**Training the Model**

Time Series predictions involve high dependency on the past readings and patterns. So, an algorithm which can keep track of past values is best suitable for time series prediction model. Recurrent Neural Networks is one such deep learning model which considers previous patterns also in consideration. As you can see in the figure below, the current output of the RNN depends on the current input as well as the previous outputs refereed by context units. The hidden units basically contain neurons which processes the input data and provides an output. RNN is a combination of feed forward and backward networks. Feed backward networks propagate the error in the current state back through the inputs to adjust the weight of the attributes in the text iteration depending on error. RNNs can sometimes be faced with exploding gradient problems. Here the value of weights of attributes become very high because of which they can control the overall behavior of the model. This can be prevented using truncated Backward Propagation Through Time.



TensorFlow was used for training and testing the model in RNN and it was installed using the following command.

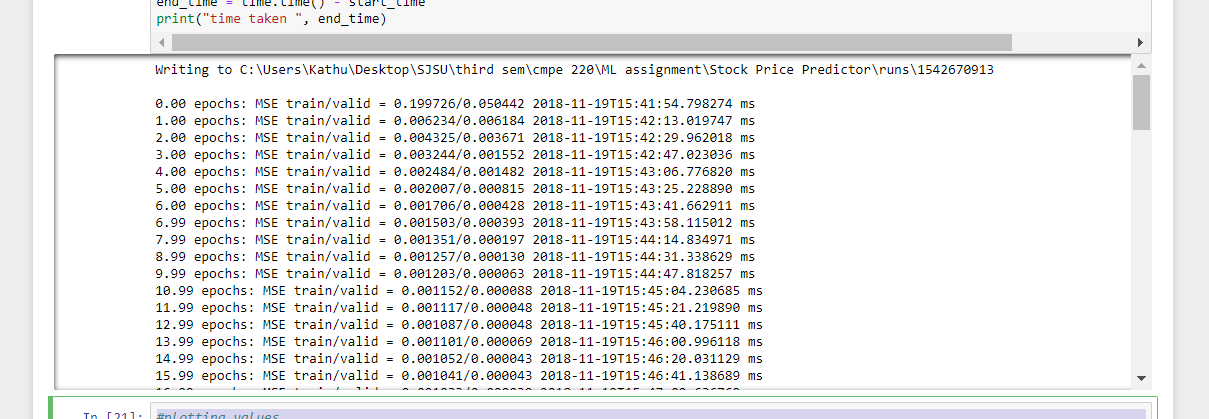
**pip install TensorFlow**.

I have installed the CPU version of TensorFlow. Tensorboard and Perfmon was used for analysis and measuring the process performance.

**Learning Algorithm Implementation**

Recurrent Neural Networks are mainly used to recognize sequences of data, speech and numeric time series data like stock prices, sensor outputs etc. 80 percent of the available data is used for training the model, 10% is used for evaluation and 10% is used for testing the model. Mean Square Error is measured during every 5 epochs and the model with the least MSE is used for future predictions.

Provided below are the MSE readings obtained during the training of the model



**0.00 epochs: MSE train/valid = 0.119297/0.022737**

**5.00 epochs: MSE train/valid = 0.004071/0.002290**

**9.99 epochs: MSE train/valid = 0.002222/0.000734**

**14.99 epochs: MSE train/valid = 0.001595/0.000122**

**19.98 epochs: MSE train/valid = 0.001346/0.000072**

**24.98 epochs: MSE train/valid = 0.001235/0.000060**

**29.98 epochs: MSE train/valid = 0.001165/0.000068**

**34.97 epochs: MSE train/valid = 0.001116/0.000055**

**39.97 epochs: MSE train/valid = 0.001078/0.000058**

**44.96 epochs: MSE train/valid = 0.001051/0.000042**

**49.96 epochs: MSE train/valid = 0.001014/0.000045**

**54.95 epochs: MSE train/valid = 0.000985/0.000049**

**59.95 epochs: MSE train/valid = 0.000961/0.000053**

**64.95 epochs: MSE train/valid = 0.000938/0.000049**

**69.94 epochs: MSE train/valid = 0.000946/0.000094**

**74.94 epochs: MSE train/valid = 0.000903/0.000053**

**79.93 epochs: MSE train/valid = 0.000880/0.000032**

**84.93 epochs: MSE train/valid = 0.000878/0.000065**

**89.93 epochs: MSE train/valid = 0.000849/0.000031**

**94.92 epochs: MSE train/valid = 0.000832/0.000031**

**99.92 epochs: MSE train/valid = 0.000871/0.000101**

**time taken 982.8535385131836**

**Parameters**

**seq\_len = 20**

**n\_steps = seq\_len-1**

**n\_inputs = 4**

**n\_neurons = 200**

**n\_outputs = 4**

**n\_layers = 4**

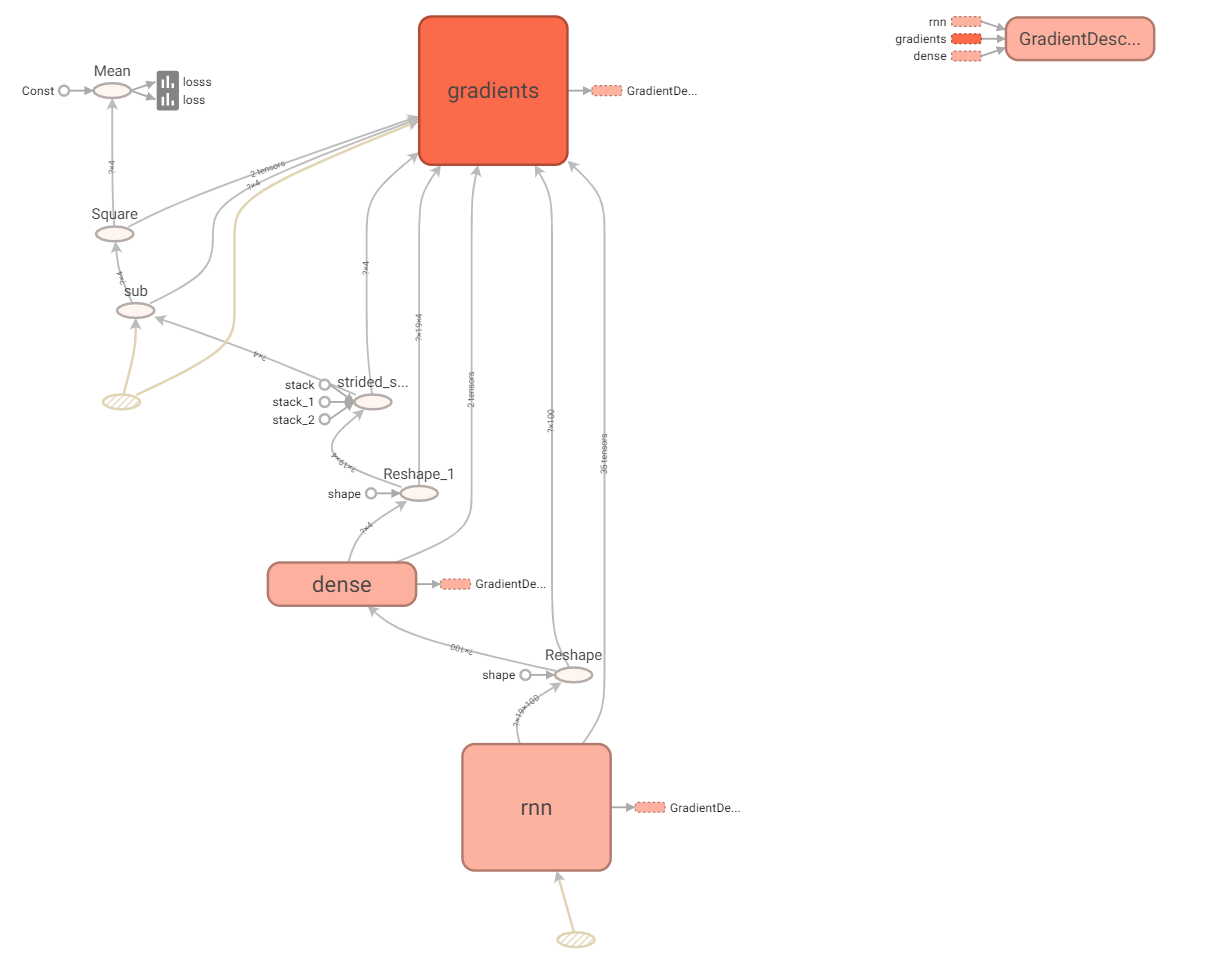
**learning\_rate = 0.001**

**batch\_size = 50**

**n\_epochs = 50**

**Architectural Diagram**

Architectural diagram was obtained from tensorboard. Conversion to “.svg” format has been done with the help of SVG crowbar application.



**Overlay of the training and testing values.**

In the graph we can see the predicted result is almost same as expected value. Slight variations in number predictions can be seen. Training with larger dataset can reduce these fluctuations.

ft = 0 # 0 = open, 1 = close, 2 = highest, 3 = lowest

**## show predictions**

**plt.figure(figsize=(15, 5));**

**#plt.subplot(1,2,1);**

**plt.plot(np.arange(y\_train.shape[0]), y\_train[:,ft], color='blue', label='train target')**

**plt.plot(np.arange(y\_train.shape[0], y\_train.shape[0]+y\_valid.shape[0]), y\_valid[:,ft],**

**color='gray', label='valid target')**

**plt.plot(np.arange(y\_train.shape[0]+y\_valid.shape[0],**

**y\_train.shape[0]+y\_test.shape[0]+y\_test.shape[0]),**

**y\_test[:,ft], color='black', label='test target')**

**plt.plot(np.arange(y\_train\_pred.shape[0]),y\_train\_pred[:,ft], color='red',**

**label='train prediction')**

**plt.plot(np.arange(y\_train\_pred.shape[0], y\_train\_pred.shape[0]+y\_valid\_pred.shape[0]),**

**y\_valid\_pred[:,ft], color='orange', label='valid prediction')**

**plt.plot(np.arange(y\_train\_pred.shape[0]+y\_valid\_pred.shape[0],**

**y\_train\_pred.shape[0]+y\_valid\_pred.shape[0]+y\_test\_pred.shape[0]),**

**y\_test\_pred[:,ft], color='green', label='test prediction')**

**plt.title('past and future stock prices for AAPL')**

**plt.xlabel('time [days]')**

**plt.ylabel('normalized price')**

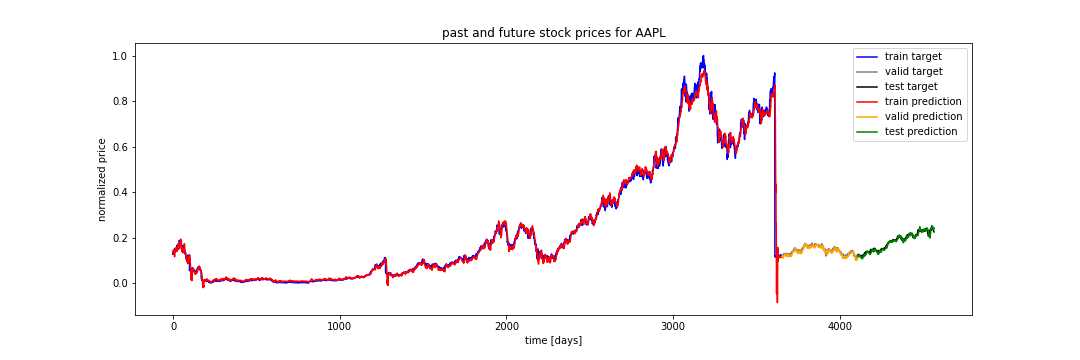
**plt.legend(loc='best');**

**plt.savefig('./eps/past\_and\_future\_prices\_neurons300.eps', format='eps', dpi=1200)**

**plt.savefig('./svg/past\_and\_future\_prices\_neurons300.svg', format='svg', dpi=1200)**

**plt.savefig('./pdf/past\_and\_future\_prices\_neurons300.pdf', format='pdf', dpi=1200)**

**plt.savefig('./png/past\_and\_future\_prices\_neurons300.png', format='png', dpi=1200)**



**Target Vs Predicted Values**

**plt.plot(np.arange(y\_train.shape[0], y\_train.shape[0]+y\_test.shape[0]),**

**y\_test[:,ft], color='black', label='test target')**

**plt.plot(np.arange(y\_train\_pred.shape[0], y\_train\_pred.shape[0]+y\_test\_pred.shape[0]),**

**y\_test\_pred[:,ft], color='green', label='test prediction')**

**plt.title('future stock prices for AAPL')**

**plt.xlabel('time [days]')**

**plt.ylabel('normalized price')**

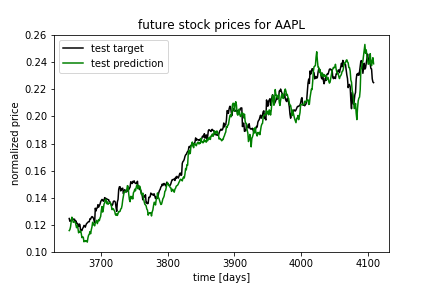
**plt.legend(loc='best')**

**plt.savefig('./eps/targetVsPredicted\_neuron300.eps', format='eps', dpi=1200)**

**plt.savefig('./svg/targetVsPredicted\_neuron300.svg', format='svg', dpi=1200)**

**plt.savefig('./pdf/targetVsPredicted\_neuron300.pdf', format='pdf', dpi=1200)**

**plt.savefig('./png/targetVsPredicted\_neuron300.png', format='png', dpi=1200)**



Code for saving MSE and test prediction values

**with open("MSE", "w") as file:**

**writer = csv.writer(file, delimiter=',', dialect = 'excel')**

**for item in mse\_values:**

**writer.writerow([item])**

**y\_train\_pred = sess.run(outputs, feed\_dict={X: x\_train})**

**y\_valid\_pred = sess.run(outputs, feed\_dict={X: x\_valid})**

**y\_test\_pred = sess.run(outputs, feed\_dict={X: x\_test})**

**with open("test\_predictions.csv",'w') as resultFile:**

**wr = csv.writer(resultFile, dialect='excel')**

**wr.writerows(y\_test\_pred)**

**end\_time = time.time() - start\_time**

**Running Tensorboard**

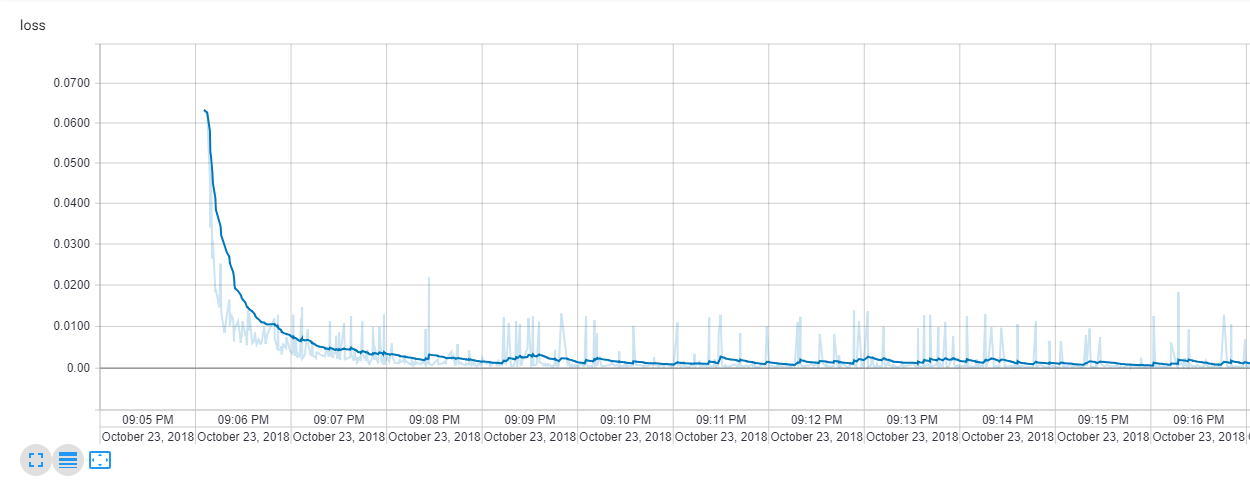
Tensorboard is a very powerful tool provided by tensorflow to visualize the model developed. It provides options to store the model so that we can import the model and run any test data. For tensor board to be running, first we must create a folder named “runs” which stores all the summary and the model data. Inside runs, tensorflow will create “checkpoints” folder which stores the model at regular intervals are defined by the user. We can train any set of data using the best model. I mainly focused on writing summaries after evry 5 epochs to get the graph showing the variation of loss with respect to the number of epochs. Add run metadata helps us to visualize the memory usage and CPU consumption of each component in the architecture.

train\_summary\_writer.add\_run\_metadata (run\_metadata, 'step%d' % iteration)

TensorFlow manual very clearly explains all these options which can help us visualize and understand more about the model.

Tensorboard can be run following the below mentioned steps.

* Open Anaconda Command Prompt
* Navigate to the parent folder which contains the “runs” folder
* Type the command **tensorboard - - logdir runs**
* This will initiate tensorboard running at port 6006 at the localhost.
* Open the browser and give the url http://localhost:6006

**Loss Graph from tensorboard**

**Code for loss**

**loss = tf.reduce\_mean(tf.square(outputs - y)) # loss function = mean squared error**

**optimizer = tf.train.GradientDescentOptimizer(learning\_rate=learning\_rate)**

**training\_op = optimizer.minimize(loss)**

**loss\_summary = tf.summary.scalar("loss", loss)**

**Loss graph obtained from Tensorboard. SVG crowbar has been used to get the svg format of the graph.**

**RESULTS**

The results of the testing phase are stored in the local machine in Excel Format and MSE values for the training phase is written in Notepad and stored in the local machine.

**MSE Comparisons for epochs 200/100/50/10/4**

Mean Square Error (MSE) is the most commonly used regression loss function. MSE is the sum of squared distances between our target variable and predicted values.

***MSE = € (expected value – predicted vale)2/Total number of elements***

The lower the MSE the more accurate the predicted value.

**#plotting values**

**plt.figure(figsize=(15, 5))**

**plt.plot(epoch\_values, mse\_values, color='red', linewidth = 4)**

**plt.title('mse vs epoch')**

**plt.xlabel('epoch')**

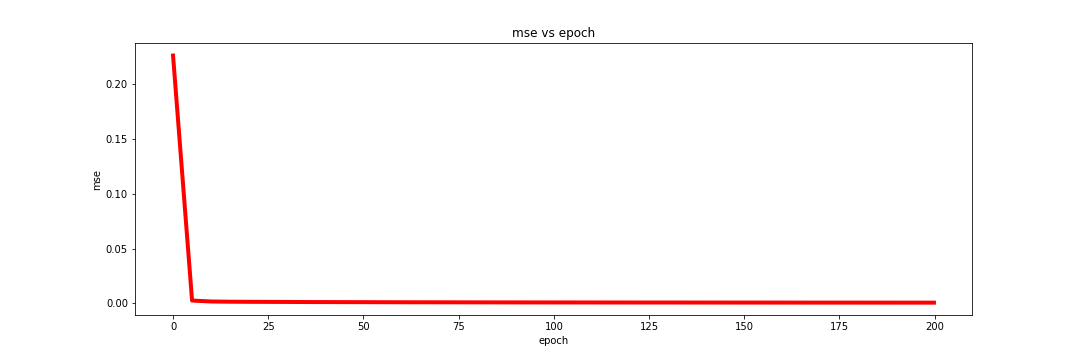
**plt.ylabel('mse')**

**plt.savefig('./eps/mse\_epochs100.eps', format='eps', dpi=1200)**

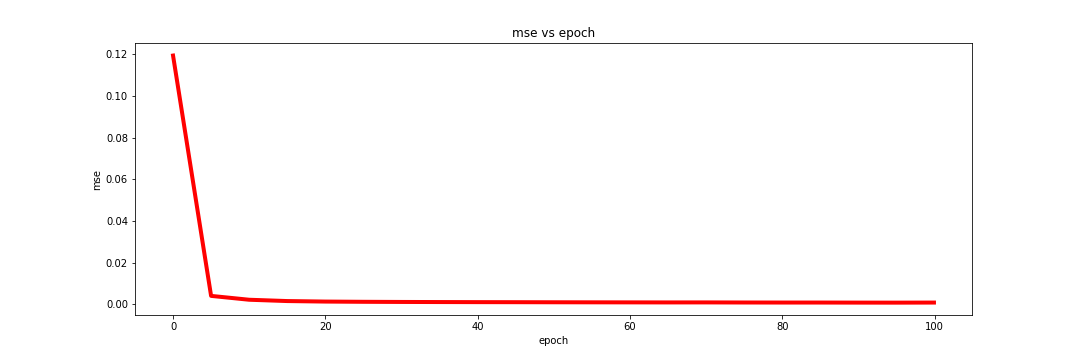
**plt.savefig('./svg/mse\_epochs100.svg', format='svg', dpi=1200)**

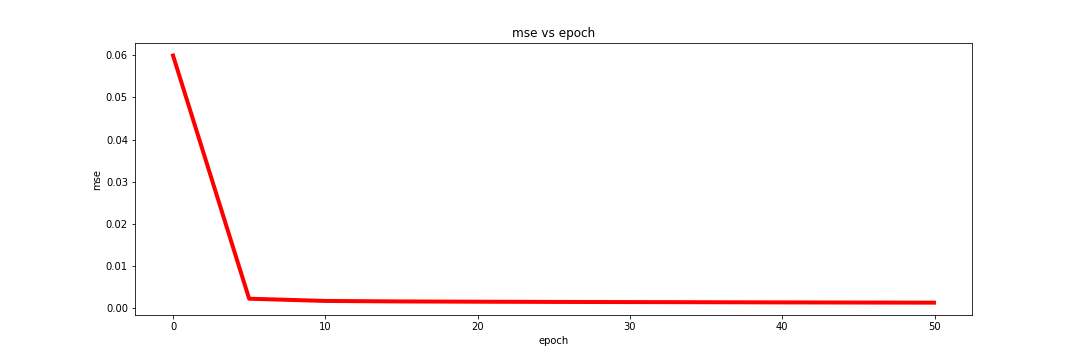
**plt.savefig('./pdf/mse\_epochs100.pdf', format='pdf', dpi=1200)**

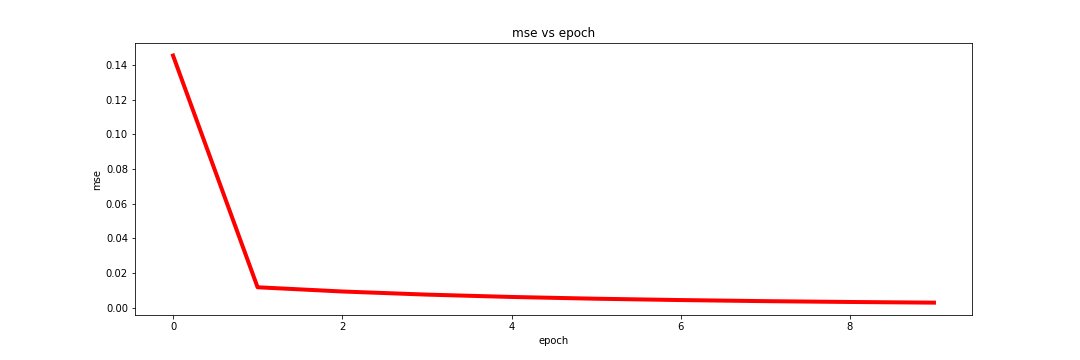
**plt.savefig('./png/mse\_epochs100.png', format='png', dpi=1200)**



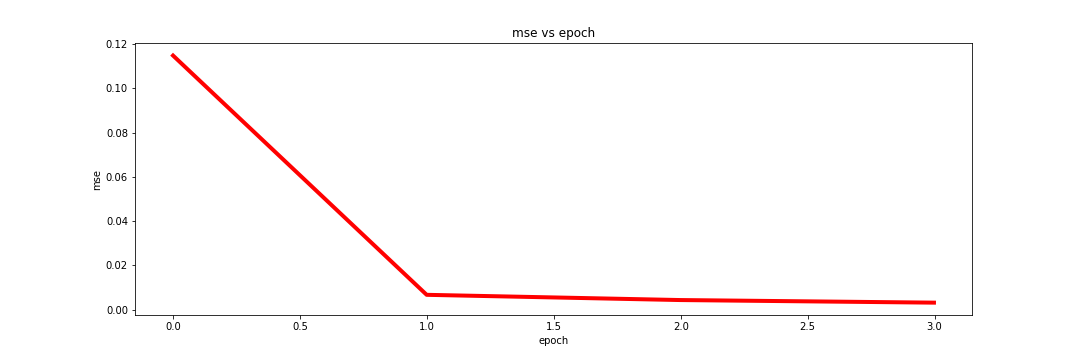
**MSE vs Epoch (200): Training time = 36 min, Final MSE = 0.000700**

**MSE vs Epoch (100) Training time = 16.4 min, Final MSE = 0.000871**

**MSE vs Epoch (50), Training time =9.3 min, Final MSE = 0.001164**



**MSE vs Epoch (10), Training time = 1.9min, Final MSE = 0.002941**

**MSE vs Epoch (4), Training time = 48.9 sec, Final MSE = 0.003210**

As we can see from above training MSE values, as the number of epochs increases, MSE reduces considerably. The percentage decrease of MSE slowly decrease with increase in the number of epochs.

|  |  |
| --- | --- |
| *Epoch* | *Percentage decrease in MSE* |
| *200* | *20% less than epoch 100* |
| *100* | *25% less than epoch 50* |
| *50* | *60% less than epoch 10* |

**Performance Evaluation**

I have used Perfmon tool in windows 10 to capture the CPU and memory performance.

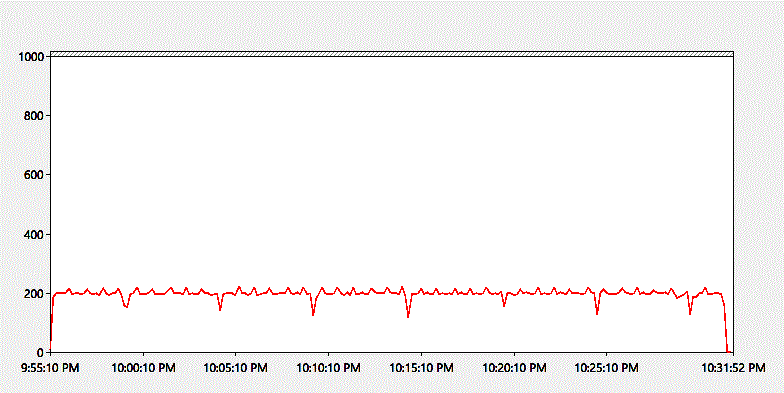
Path of perfmon Run🡪Perfmon

Below are the visualizations of CPU utilizations for different epochs

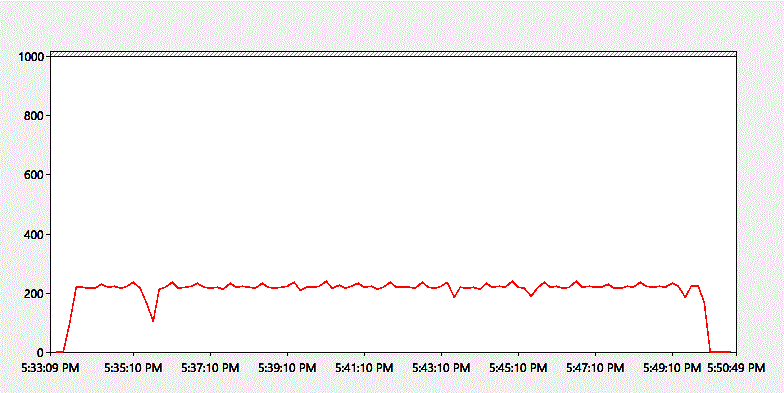
The program was run on an Intel Core i5 – 7200U. The CPU is a 2 core, 4 thread CPU with Simultaneous Multithreading (SMT). In the performance graphs below, you can see that as the number of epochs increases, CPU time also increases. For a 2 core, 4 threaded CPU, the CPU consumption is expected to be greater than 200% and less than 400%. SMT is seen in superscalar CPUs which allows multiple processes to run on a single CPU core. SMT utilizes the concurrency between processes. In my machine 2 cores can run 4 threads, but the total CPU utilization is always less than 400% because there is always some sort of data dependency or serial dependency because of which full utilization is not obtained. Furthermore, full 400% is unrealistic for the workload because of hardware limitations for SMT threads.

Performance Monitor graph has been provided in the visualization folder.

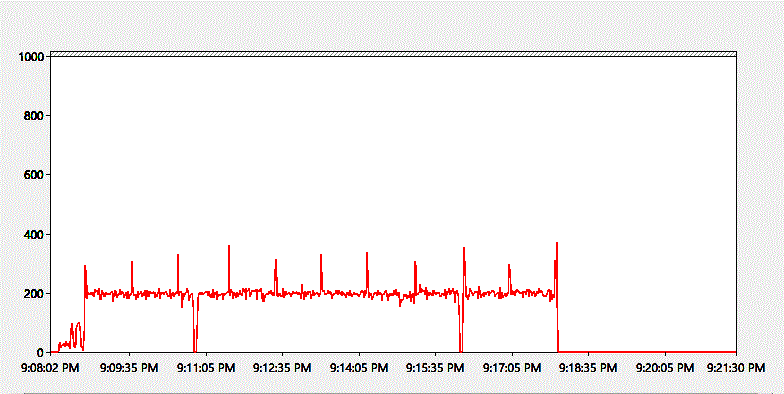
**Epoch 200**



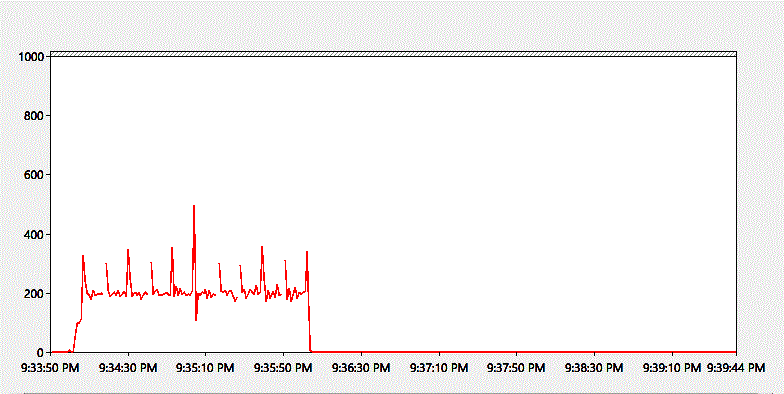
**Epoch 100**



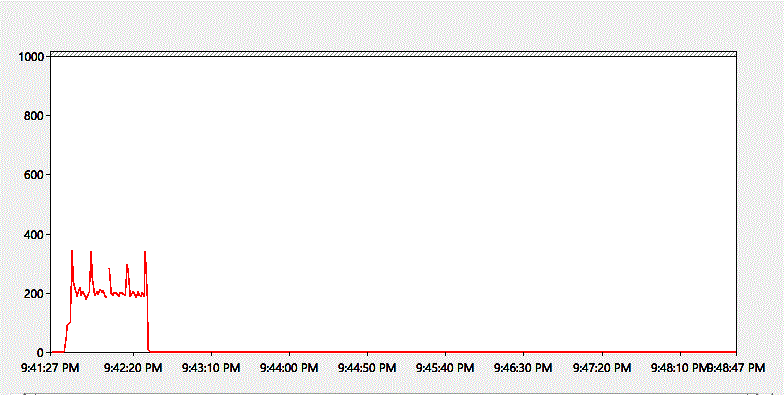
**Epoch 50**



**Epoch 10**



**Epoch 4**



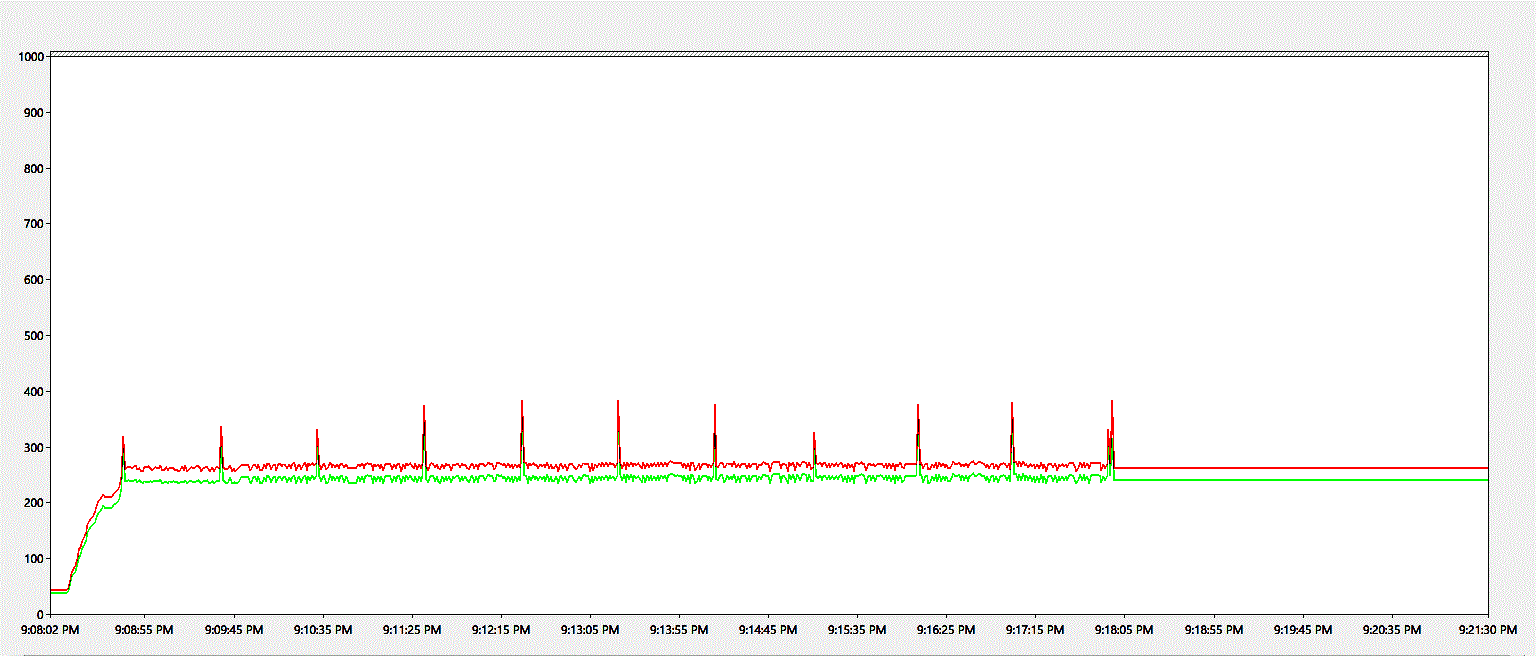
In all the readings above, we can see that CPU utilizations was varying between 200%-400%. With change is number of epochs, increase/ decrease CPU utilization is not observed. This is expected as the number of epochs only has a serial effect.

**Memory Consumption**

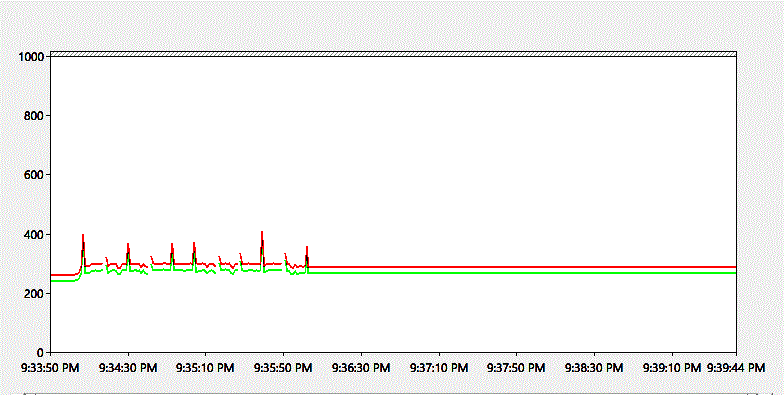
Memory Consumption was measured based on Private Bytes and Working Set – Private

**Private Bytes** is defined as the memory allocated to a process that cannot be shared with other processes. Indicated in red color in following graphs.

**Working Set Private** is the number of bytes of memory that is used by the process currently resident in physical memory (RAM). Working set is a subset of private bytes. Indicated in green in following graphs.

**Epochs 50**

**Epochs 10**



As you can see in graphs, the amount of memory allocated to the program falls in the range of 200-400 MB. Apart from occasional spikes in activity, the memory consumption does not have an impact with number of epochs.

**MSE Comparison for Learning rate 0.0001 vs Learning Rate 0.001**

Learning rate is one of the hyper-parameters for Recurrent Neural Networks. Efficient tuning of hyperparameters can result in low errors and higher accuracies.

Please find below the hyperparameters for the model and their default values

n\_neurons = 200

learning\_rate = 0.001

batch\_size = 50

n\_epochs = 200

Keeping n\_neurons, batch\_size, n\_epochs as constant, the model was run with learning rate 0.001 and 0.0001. Below findings were observed

|  |  |
| --- | --- |
| **Learning Rate** | **MSE** |
| 0.0001 | 0.0013234224 |
| 0.001 | 0.00069977145 |

Learning rate 0.001 provided a lower MSE than 0.0001. There by I came to the assumption that for the above values of neurons, learning rate, batch size, learning rate 0.001 provided better accuracy.

|  |  |
| --- | --- |
| **Learning Rate** | **Training time** |
| 0.0001 | 2173.69 sec |
| 0.001 | 2219.02 sec |

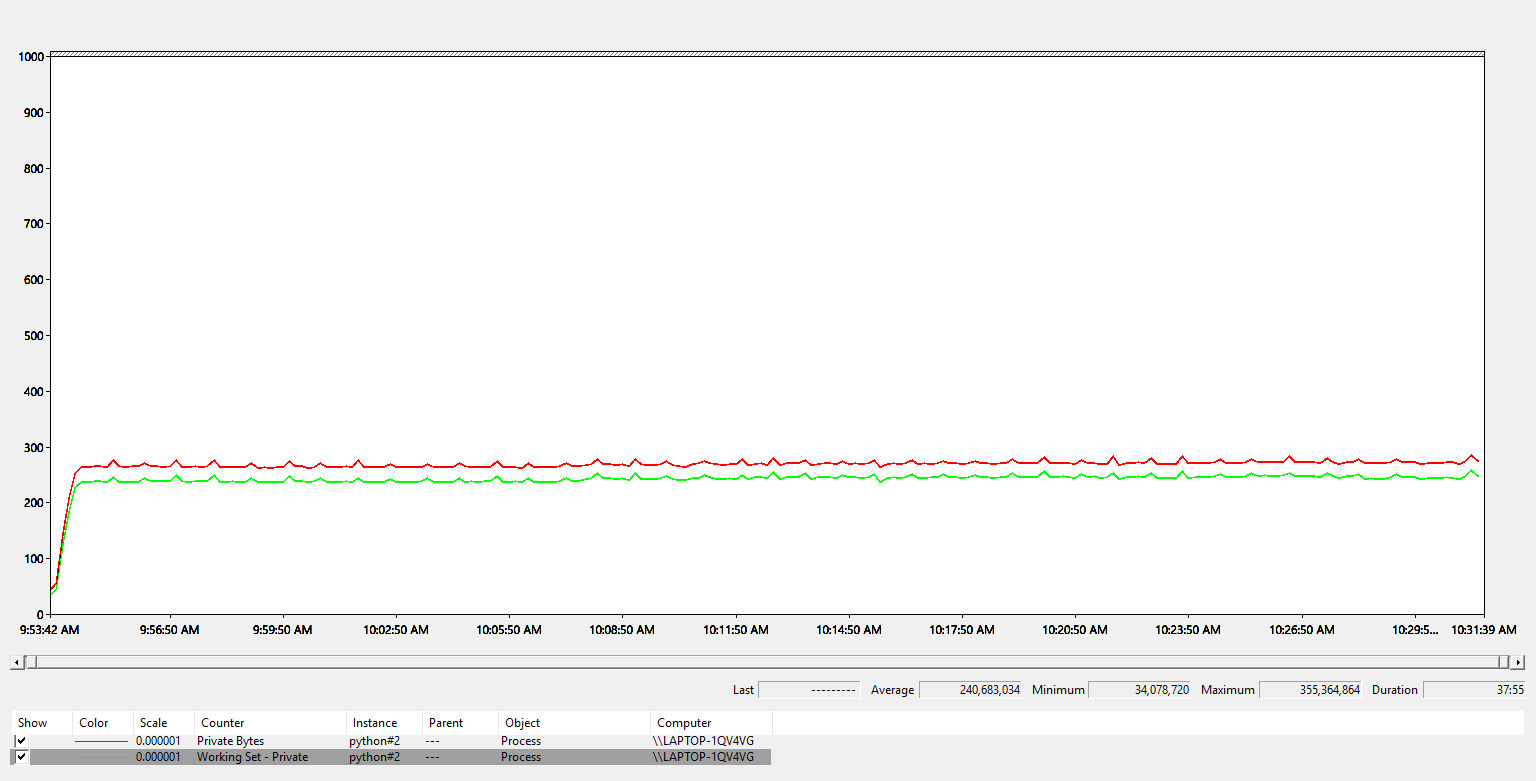
With number of epochs kept constant, the time taken to train the model with learning rate 0.0001 is greater than the time taken to train the model with learning rate 0.001. From this evaluation it can be concluded that lower the learning rate, the higher the time taken by the model to converge to a low error rate.

The CPU and memory performance remain the same for both learning rates.

CPU performance at around 200% and memory consumption between 200-300 MB.

Since memory and CPU consumptions are almost the same for both learning rates I am avoiding the comparison graphs in report.

**Memory Consumption**



**CPU Performance**

**Evaluating MSE, Time Consumption, CPU usage and memory consumption by varying the neurons**

Learning rate = 0.001

Batch size = 50

Epochs = 100

* **MSE comparison**

As shown below, as the number of neurons increases the MSE decreases. This is due to the increased previous information storage and better back propagation with increase in neurons

|  |  |
| --- | --- |
| **Number of Neurons** | **MSE** |
| 100 | 0.001182 |
| 200 | 0.000871 |
| 300 | 0.00069977145 |

* **Time Consumption**

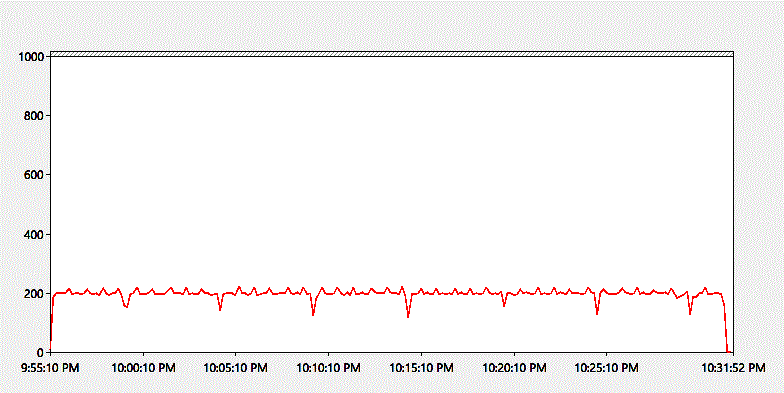
As the number of neurons increases time taken to execute same number of epochs increases. This is due to the increased computations, especially matrix multiplications and gradient calculations involved with RNN

|  |  |
| --- | --- |
| **Number of Neurons** | **Time Taken** |
| 100 | 783.35 sec |
| 200 | 982.9 sec |
| 300 | * 1. sec |

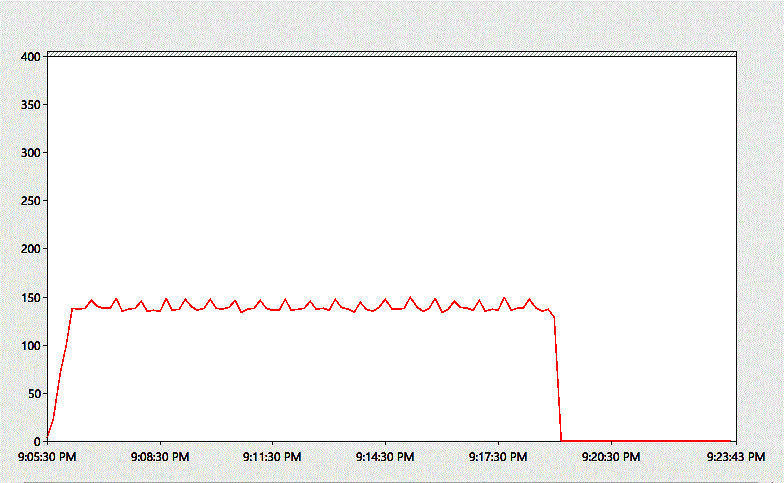
* **CPU Utilization**

As shown below 100 neurons has a CPU consumption of approximately 150% and 300 neurons have a CPU consumption of approximately 250%. As the neurons increases, computations also increase which results in higher CPU utilization. However, there will be an upper-limit for the number of neurons beyond which CPU utilization will not be increased but time taken will be increased by a bigger margin. The increase in CPU utilization is not observed to be increasing linearly with the number of neurons.

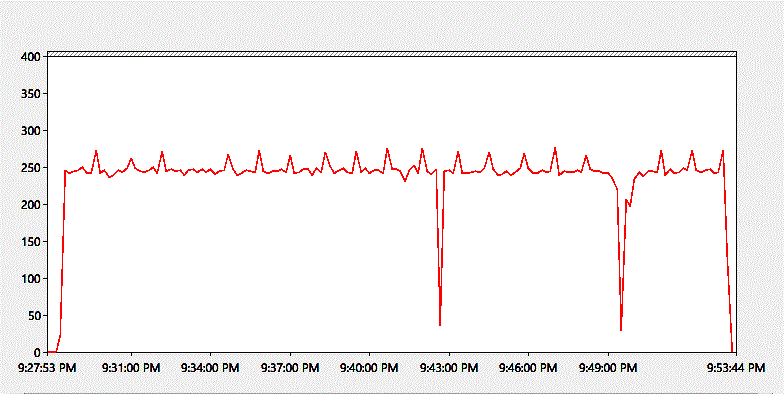
**CPU utilization for 200 neurons**



**CPU utilization for 100 neurons**



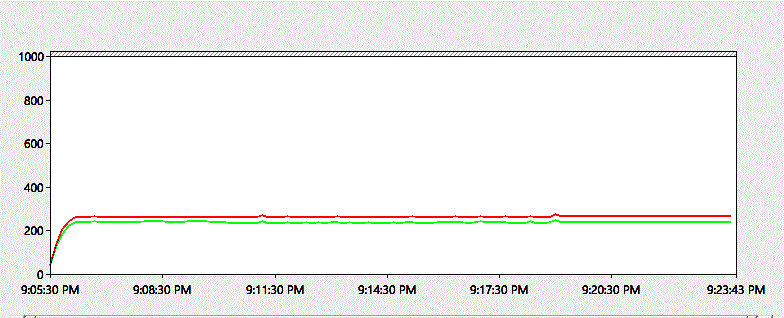
**CPU utilization for 300 neurons**



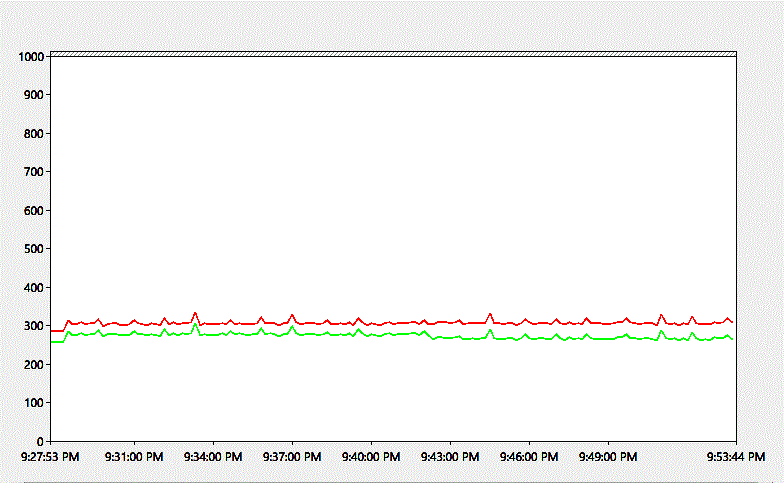
* **Memory Consumption**

Memory Consumption also increases with increase in number of neurons as shown below.

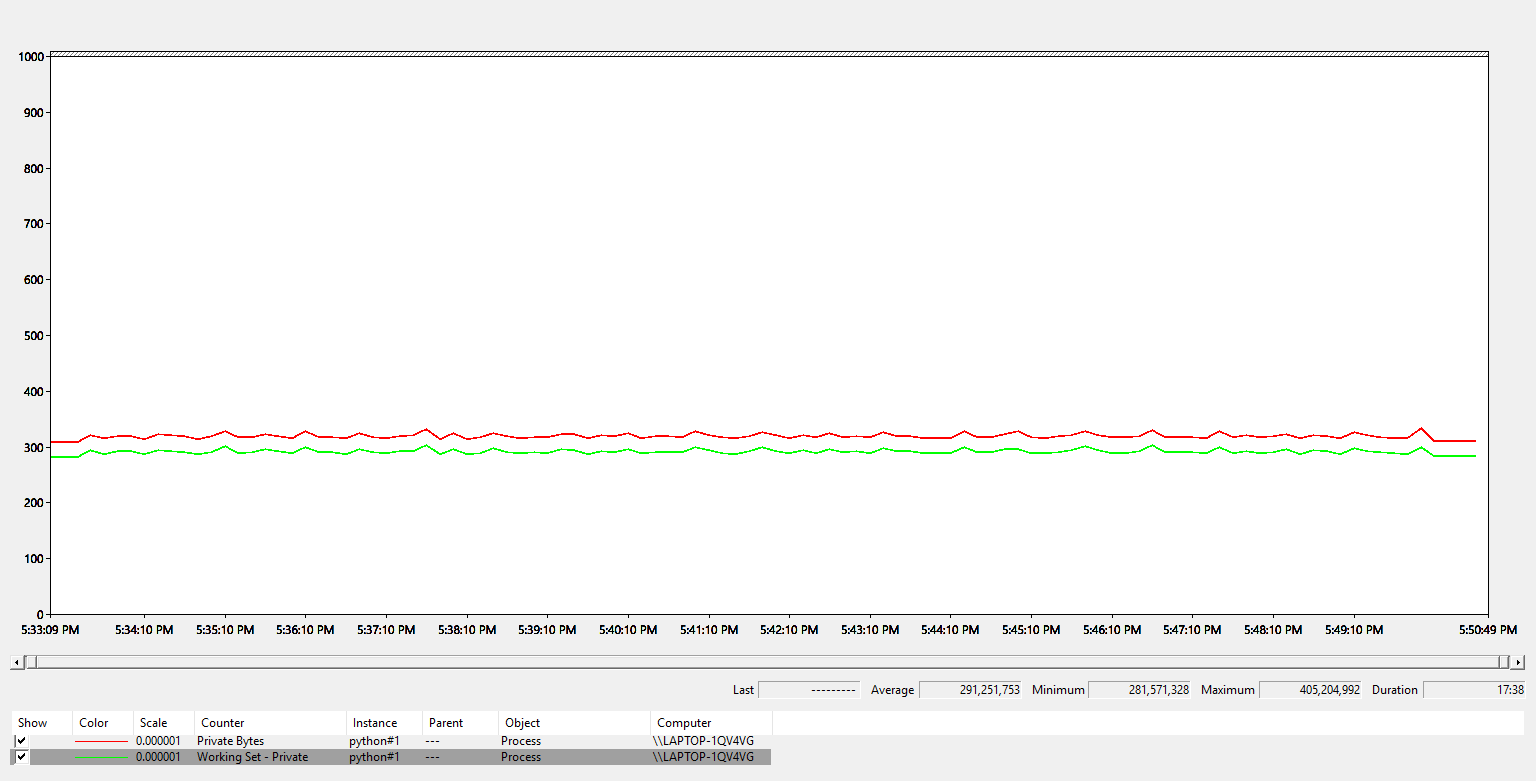
**Number of neurons: 100**



**Number of neurons: 300**



**Number of neurons: 200**



From the experiments conducted, I have found that the optimal number of neurons for model development and training is 200

**Conclusion**

From the experiments it’s been found that the optimum values of hyperparameters for training the model would be:

**Number of neurons: 200**

**Number of epochs: 100**

**Batch size: 50**

**Learning rate: 0.001**

Model run of these parameters can provide reduced MSE, optimal CPU and memory utilization. Time taken to run the model can be also kept at optimal levels.

Training the model with a larger dataset can also result in improved accuracy.

**References**

<https://medium.com/python-data/quandl-getting-end-of-day-stock-data-with-python-8652671d6661>

<https://en.wikipedia.org/wiki/Open-high-low-close_chart>

<https://www.kaggle.com/dgawlik/nyse>

<https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/cc749154(v=ws.11)>

<https://www.tensorflow.org/guide/summaries_and_tensorboard>

<http://blogs.microsoft.co.il/sasha/2016/01/05/windows-process-memory-usage-demystified/>