# SYS 6016 - Machine Learning (Spring 2019)

**Project: Deep Learning for short-term forecasting of mid-price change in the stock market**
**Project Area: Prediction**

**Group: Team EC**
**Members: Elena Gillis (emg3sc) and Charu Rawat (cr4zy)**

**22nd April 2019**

# Deep Learning for short-term forecasting of mid-price change in the stock market

***Abstract***

*In this project we aim to predict the change in movement of the mid-range prices for Wayfair Inc. (NYSE: W) using the limit order book data from NASDAQ. As part of our process, we implement deep learning models, such as Feed Forward Neural Network (FFNN), Convolutional Neural Network (CNN), and Recurrent Neural Network (RNN). We test these models with different parameters and architectures and conclude that best accuracy is given by the FFNN model with two hidden layers, Gradient Descent Optimizer and a ReLU activation function. We compare the models built on neural networks to our baseline models built using kNN and SVM and find that the FFNN model scores higher on accuracy compared to the baseline models. Our best FFNN model achieves an accuracy of 51% over the baseline accuracy of 37% obtained using machine learning algorithms.*

## Introduction

In recent years researchers have developed a lot of interest in stock market prediction because of its dynamic and unpredictable nature. High-frequency trading or algorithmic trading has gained significant momentum over the last few years. This has continued to usher in the era of big data with advanced computational techniques compounding this phenomenon. Gaining access to real-time data and having the means to leverage complex models to gain an advantage in the market is imperative today, as margins and arbitrages are closed in a fraction of a second. With the increase in focus on trading in an order-driven market, one of the central aspects to study in such a framework in the "limit order book" (LOB), which contains information about traders' intention to buy or sell at a certain price for a particular number of shares pertaining to an asset or a stock. Accurately forecasting the dynamics of the order book is crucial in a variety of application scenarios - from brokers using it to optimize trade execution schedules [1] to asset managers contemplating portfolio rebalancing [2], the importance of forecasting stock market trends has motivated research in developing predictive algorithms, modeling the dynamics of order books as well as empirically studying the behavior of order books [3].
For this project, we have been provided with Level-I NASDAQ limit order book data or the "LOB" data which comprises of high frequency stock market data consisting of all sell/buy transactions for stocks listed on the NASDAQ exchange.

*Terminology -* A limit order is an order to buy or sell the desired number of units of an asset at the desired price. The order book at any time *t* is the collection of all outstanding limit orders at all prices at that time. The highest price among all the buy limit orders at time *t* is called the best bid price $(P_b(t))$ at time *t*. The lowest price among all the sell limit orders at time *t* is called the best ask price $(P_a(t))$ or best offer price at time *t*. The difference between the best ask and the best bid prices is called the bid-ask spread. The average of the best ask and the best bid is called the **mid-price**, denoted $P(t)$.

As described in out proposal for this project we implement a Feed Forward Neural Network (FFNN) to predict the change in the directional movement on the mid-price—average between ask and bid prices for a stock that is calculated over a window of one minute (60 second intervals) for data that spans over a period of four days. We then compare this model with baseline models, such as k-Nearest Neighbors (kNN) and Support Vector Machines (SVM), as well as Convolutional Neural Network (CNN), and Recurrent Neural Network (RNN).

**Literature Review**

Quantitative analysis of financial markets has long been of interest to many researchers. Methods such as mathematical and statistical modeling along with machine learning have been used to predict movements of stock prices, volatility of future prices, and detection of anomalous events that could affect these prices. In their paper Tsantekidis, Avraam et al. (2018). use deep learning techniques to improve the accuracy of price movement prediction and enable large sets of data to be used in such analysis [4]. In their paper *"Using Deep Learning for price prediction by exploiting stationary limit order book features"*, the authors present a method to tackle volatility of price values by constructing stationary features for Limit Order Books (LOB) and use Convolutional Neural Networks (CNN) for feature extraction and Long Short Term Memory (LSTM) recurrent neural network for classifying the movement of prices.

Tsantekidis, Avraam et al. (2018) argue that using non-stationary raw data from LOBs presents distribution shifts that harm predictive ability of the models - whereas most features can be normalized together since they follow the same distribution, normalization of price values presents an additional challenge due to unpredictable drastic fluctuations resulting in introduction of new data that does not fit the range of existing values, thus altering their distribution. The authors emphasize that distribution shifts can be avoided by normalizing the non-stationary data with mean and standard deviation of previous day's values. However, this type of normalization makes the difference between the levels of LOB very small, making it hard to extract significant features. As a solution, instead of normalizing raw LOB price values they propose setting price values as their percentage difference to the current mid-price of LOB – this removes non-stationarity and makes feature extraction easier while improving performance of the models. As a result, they derive three stationary features, normalize them separately using z-score normalization, and concatenate them into a single vector. The extracted stationary features are price level difference (difference of each price to the current mid-level price), mid-price change (change of the current mid-level price to precious mid-level price), and depth size cumsum (number of limit orders at each price level). The output vector used in this case consists of {-1, 0, 1}, each number indicating the direction of movement of the price trend, making this a classification problem. In order to prevent the model responding to miniscule changed in prices two averaging filters are applied on a sequence of prices over a predetermined interval of time. A threshold hyperparameter is then superimposed on these averages to regulate the sensitivity of the model to changes in price. Relevant features are extracted and then fed into Neural Networks (FFNN). Their research also makes use of CNN and LSTMs.

**Data**

We train our model on the subset of the data that contains transaction data for Wayfair Inc. (NYSE:W) - an e-commerce company that sells home goods. For this analysis, we combine NASDAQ data for January 3, January 4, January 6, and January 9, 2017. The combined dataset consists of ~154k observations and nine variables. For this analysis we retain six features: timestamp, order id, book even type, price, and side. The book event type consists of four categories: add, modify, cancel, and trade; side consists of ask, bid, and unknown. The 'ask' side of the data represents the supply of the asset at a given price and the 'bid' side represents the demand. If no trade occurs once the stocks are added, the ask and bid prices continue to converge until the order is wither traded or canceled. The prices in the data range from $10 to $19.99 billion and are all multiplied by 10000 to avoid using decimals; time is recorded in nanoseconds (10^-9 seconds).

The number of book events for Wayfair (NYSE:W) that we use in this analysis are approximately equally distributed across the four dates represented in this dataset with roughly 25 thousand observations for each day. The time stamp for the book events ranges from 7:00 a.m. (pre-market trading) to 4:00 p.m. (stock market close) as the end time. To observe this

distribution and validate the health of data we plotted the count of book events over time aggregated by the hour. In Figure 1 we can clearly observe spikes during the times and dates represented in our dataset as well as an underlying pattern in distribution of the data within each given timeframe.

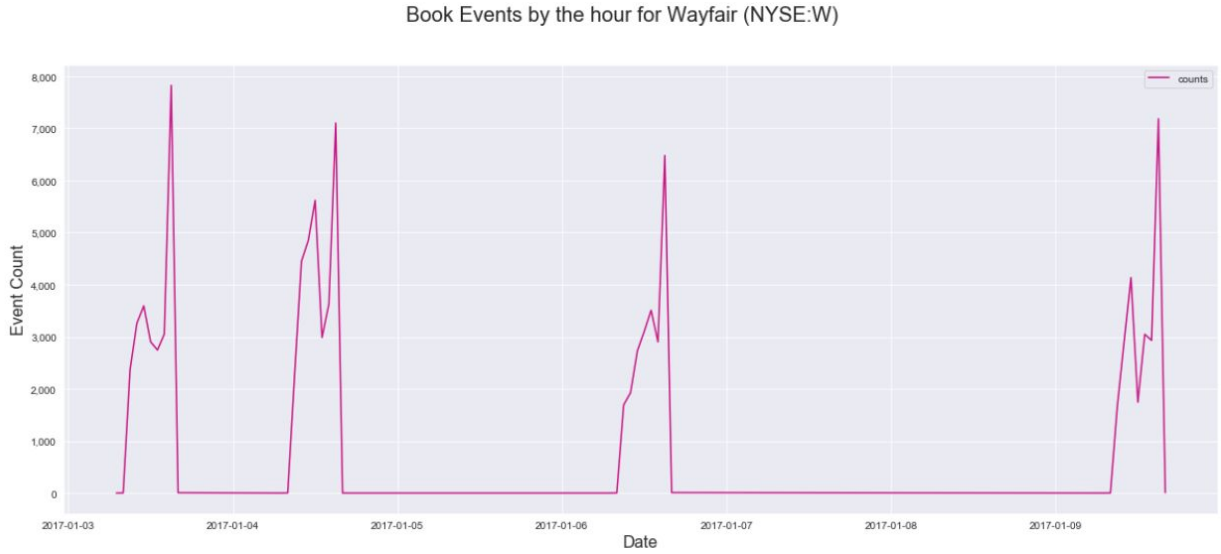



Figure 1

The spread in the bid-ask prices are indicative of the liquidity of a stock. We can see that the stock for Wayfair (NYSE:W) is highly liquid, as the demand is generally higher than the supply. (Figure 2).

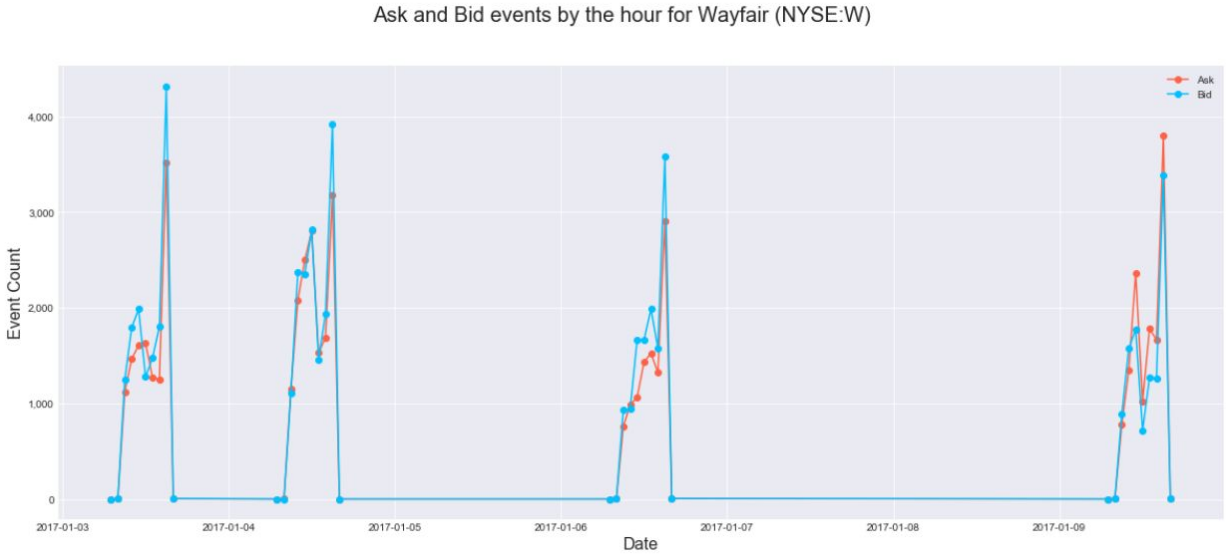



Figure 2

Post pre-processing and reshaping our data ,we divide our data into a 75-25% split for train and test. Below is a table explaining the size of our data (train/test) by label

**Table I - Number of Records in the Data**

| Set / Label | 1 | -1 | 0 |
|---|---|---|---|
| Train | 6285 | 6565 | 4637 |
| Test | 2095 | 2188 | 1546 |

## Pre-processing

For the purpose of this analysis, our data pre-processing comprises largely of two steps - data cleaning and processing for feature generation.

*Data Cleaning* - This dataset consists of many records that don't need to be used in this analysis eg - canceled orders. Canceled orders are the orders that were not executed due to inability of the ask and bid prices to converge. We remove observations pertaining to these cancellations. We also remove observation where the order side is unknown, since this data does not give us sufficient information to compute the mid-prices for prediction. Each order has a timestamp attributed to it which is granular to the nanosecond. We convert this timestamp column to a datetime format and round it to seconds. We further localize the timezone and convert it to EDT since the trading is done in the Eastern time zone.

*Processing for feature generation* - For every timestamp (rounded to the minute) there exist multiple bid and ask orders in the data. Owing to the nature of how orders are placed, it is known that only the best bid-ask orders get executed. For every timestamp *t*, we chose the best bid order (max bid price) and the best ask order (min ask price). Since all timestamps don't necessarily have a bid and ask order, we have records with empty values. To impute these empty values, we use linear interpolation. To generate features that can be used for modeling, our first step was to calculate the mid-price for every timestamp.

The mid-price is defined as the mid-point between the best bid and the best ask prices at time t by

$$p^{(1)}{}_m(t) = \frac{p^{(1)}{}_a(t) + p^{(1)}{}_b(t)}{2}$$

The difference between the price values in different order levels are almost always minuscule. Hence it is not appropriate to directly measure a change in mid-price value from one event to another. This dynamic makes it tough to work with the data as is and requires the use of sophisticated methods to engineer features that are appropriate to use as inputs into the model. To compute the change in the mid-price correctly, we engineer 2 unique types of features related to mid-price similar to the work done by Tsantekidis, Avraam et al. (2018).

- Price level difference - The difference of each price level (both bid and ask) to the current mid-price. These serve as statistic features that represent the proportional difference between ith price and the mid-price at time t. There will be 2 of these features for every timestamp.

$$p'^{(i)}(t) = \frac{p^{(i)}(t)}{p_m(t)} - 1$$

- Mid-price change - The change of the current mid-price to the mid-price of the previous time step.

$$p'_m(t) = \frac{p_m(t)}{p_m(t-1)} - 1$$

We then applied z-score normalization onto the mid-price m(t) values. The formula for z-score defined as –

$$x_{norm} = \frac{(x - \mu)}{\sigma}$$

where $x$ is a feature to be normalized, $\mu$ is the mean and $\sigma$ is the standard deviation across all samples.

This problem *ad rem* to our objective for this project is that of classification where we need to predict (or classify) the direction of the change in the mid-price. For a given stock, we define the direction of the mid-price as $l_k(t) = \{-1, 0, 1\}$ depending on whether the mid-price decreased (-1), remained stationary (0) or increased (1) after *k* LOB events occurred. Simply comparing the mid-price change to determine the upward direction of the mid-price would be noisy, since the smallest change would be registered as directional movement. This was one of the major challenges we encountered with this data. To remedy this, we use a moving or rolling average methodology wherein we compute a rolling average on the normalized mid-price over a window of k orders defined below

$$m_a(t) = \frac{1}{k} \sum_{i=1}^{k} p_m(t+i)$$

We choose k = 60 since we want to predict 1 minute into the future from a given time step.
The penultimate goal is to fit a model that on examining the current and past supply and demand of limit orders for a stock will predict the direction $l_k(t)$ of the mid-price after *k* orders (k = 60).
The label *l(t)*, that expresses the direction of price movement at time *t* is to be computed over the moving avg mid-prices defined above. Therefore, the labels are redefined as:

$$l_t = \begin{cases} 1, \ if \ \dfrac{m_a(t)}{p_m(t)} > 1 + \alpha \\ -1, \ if \ \dfrac{m_a(t)}{p_m(t)} < 1 - \alpha \\ 0, \ otherwise \end{cases}$$

where $\boldsymbol{\alpha}$ can be considered as a penalizing factor or a tuning parameter since it is the threshold that determines how significant a mid-price change must be in order to label the movement as upward or downward. The resulting labels consisting of 3 classes present the trend to be predicted.

An important point to note here is that, for a multilabel classification problem such as in this analysis, one can measure performance of models using various metrics such as accuracy, ROC, f1 score,precision,recall ,etc. In the study done by Tsantekidis, Avraam et al. (2018) , model performance is measured using f1,precision, and recall because of heavy class imbalance in the data. The parameter $\boldsymbol{\alpha}$ can be used to adjust the distribution of classes in such a scenario. In our analysis, we looked at the class distribution across various measures of alpha and decided to go with alpha = 0.001 that gives us a relatively uniform class distribution. Because of negligible class imbalance in our analysis, we measure model performance by looking at the accuracy metric which would give us a fair measure of model performance. In the future, when we apply our analysis onto other stocks, depending on the value of $\boldsymbol{\alpha}$ chosen and the class distribution, it might make more sense to look at metrics such as ROC or precision to measure model performance. We divide our data into a 75-25% split for train and test.

**Baseline Model Description**

To predict the movement of prices we use multiclass classification machine learning algorithms as our baseline models—k-Nearest Neighbors (kNN) and Support Vector Machines (SVM).
kNN is a non-parametric classification model that computes average values for each class and classifies observations by assigning them to the class, the average value of which is the shortest Euclidean distance away from the given observation. We use 75% of our dataset to train the kNN model and test it on remaining 25% of the data. The final model classifies out data into seven clusters and predicts with accuracy of 37.1%.

kNN confusion matrix

```
[[ 980 1142  490]
 [ 521  997  341]
 [ 735 1162  609]]
```

An SVM model with a linear kernel function uses n-dimensional hyperplanes to divide the input space into classes and classifies observations by finding a hyperplane that maximizes the margin between classes. We train the SVM model with a linear kernel on 75% of the data and test it on 25%. The model predicts with accuracy of 38%. SVM takes relatively longer to execute as compared to kNN.

SVM confusion matrix

```
[[2200    3   11]
 [1532    0    1]
 [2050    1   16]]
```

**Models**

We implemented 3 neural network models - Feed Forward Neural Network (FFNN), Convolutional Neural Network (CNN), and Recurrent Neural Network (RNN). Below, we describe our methodology in implementing these models and discuss the architecture of each one of them.
*FFNN* - We train a Feed Forward Neural Network (FFNN). In FFNN the connections between layers do not form a cycle and as such the data is sent forward through hidden layers that apply weight and biases to the data and one output layer that uses these parameters to calculate the estimates of the response.
As an additional step in data preprocessing we one-hot-encode the response variable to produce a matrix of the response values. We then split both the input feature matrix and the response matrix horizontally at 10 thousand rows, which results in approximately equal numbers of observations for train and test datasets. By creating an iterator we sequentially extract features from the dataframe and feed them into the hidden layer.
In construction of the model we use one hidden layer with 15 units, a weight matrix initialized with random variables from a normal distribution, and a bias vector initialized with zeros. In each neuron the data matrix is multiplied by the weight matrix and the bias vector is added to the result. The Rectified Linear Units Function (ReLU) is then used as the activation function in the hidden layer.

The Rectified Linear Units Function is defined below as

$$R(x) = max(0, x)$$

Where x is a data point in the preactivation matrix.

The output layer uses a weight vector initialized as random variables from a normal distribution and a bias vector initialized as zeros. These vectors are applied to the hidden layer output matrix flattened by the ReLU function in order to compute the final output vector. We then use the cross-entropy function to compute errors at each datapoint. Since our model works on a classification problem we use softmax function on the data points, which normalizes the values into a vector of range (0, 1) with all the resulting values adding up to one (essentially indicating probabilities of the data belonging to each class), before sending them through the cross entropy function. To calculate the overall loss of a single run we find the mean across all the cross-entropy errors. We then specify a learning rate - a minimum acceptable error - that will act as a buffer when minimizing loss.

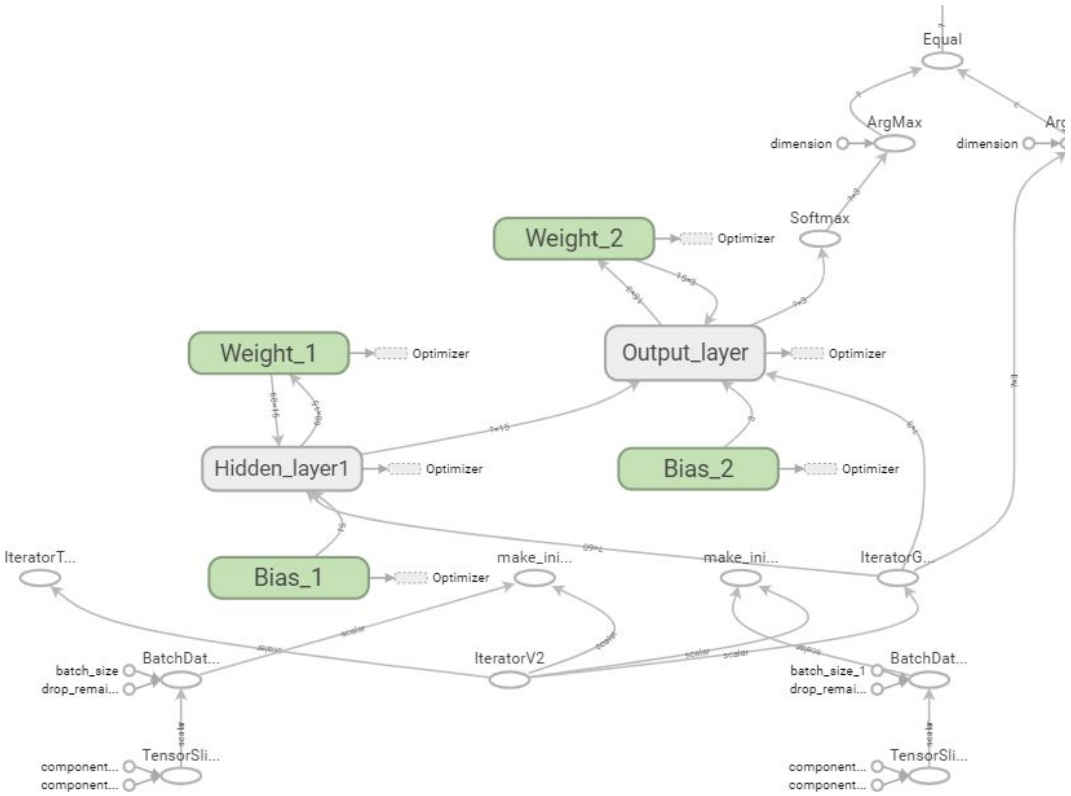
We define Cross-entropy function for classification as -

$$CE = -\sum_{i}^{C} \mathrm{t}_i \log\left(\mathrm{S}_i\right)$$

where C is the number of classes, and S is the output of the soft max for each set of observations.

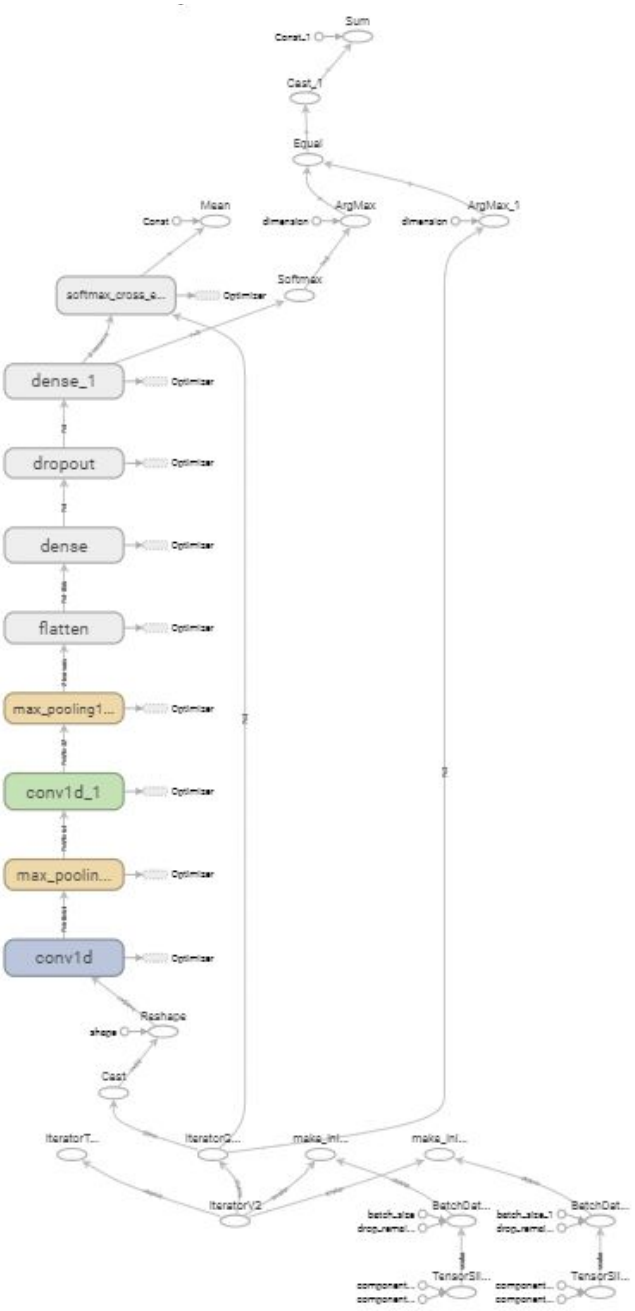
To train our model we re-evaluate the weight and biases vectors through backpropagation by iterating it over multiple epochs. In our best model we choose 1000 epochs, leaning rate of 0.01, and split the data into into batches of 100 to run simultaneously within each epoch.
Below is the architecture graph for our feed Forward Neural Network (FFNN) -

**Architecture Graph for FFNN based on our best model (Model 3)**



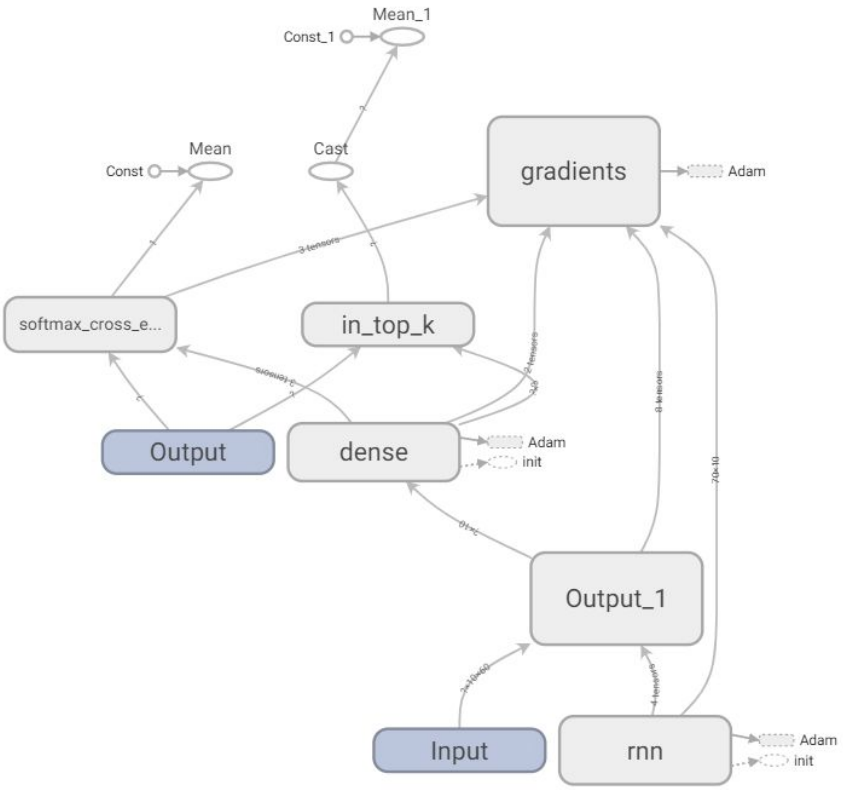
*CNN* - We trained a 1D Convolutional Neural Network Model (CNN) as well. We begin by training a CNN with two convolutional layers, max pooling layers, and outputting the results from a dense layer with a dropout rate of 0.4. We tested the model with Gradient Descent Optimizer, Adam Optimizer, and RMS Prop Optimizer. Initially, we used 128 filters in the first convolutional layer and 64 filters in the second, but ended up choosing 64 and 32 respectively to optimize efficiency of computation. We used a kernel size of 3 in both convolutional layers, same paddings, and a ReLU activation function that was defined above.

**Architecture Graph for CNN**



*RNN* - In addition to FFNN and CNN we trained and tested a simple Recurrent Neural Network (RNN) on our data. We continued to use the Rectified Linear Units Function (ReLU) defined above, as the activation function in the hidden layer. The architecture graph for that model can be seen in the figure below.

**Architecture Graph for RNN**




Below is a comparison of the results across the 3 best models obtained under each of the neural networks. As we can see from the results below, a FFNN seems to give us the best accuracy.

**Table II - Model Comparison across different Neural Network Implementations**

| Model | Epochs (FFNN,CNN) | Neurons (RNN) | Learning Rate | Batches (FFNN,CNN) | Iteration (RNN) | Test Accuracy | Time (sec) | Optimizer |
|---|---|---|---|---|---|---|---|---|
| CNN | 20 | - | 0.001 | 100 | - | 28.54% | 58.28 | Gradient Descent |
| RNN | 100 | - | 0.01 | - | 1000 | 42.66% | 14.83 | Adam |
| FFNN | - | 1000 | 0.01 | 100 | - | 51.00% | 147.42 | Gradient Descent |

**Results**

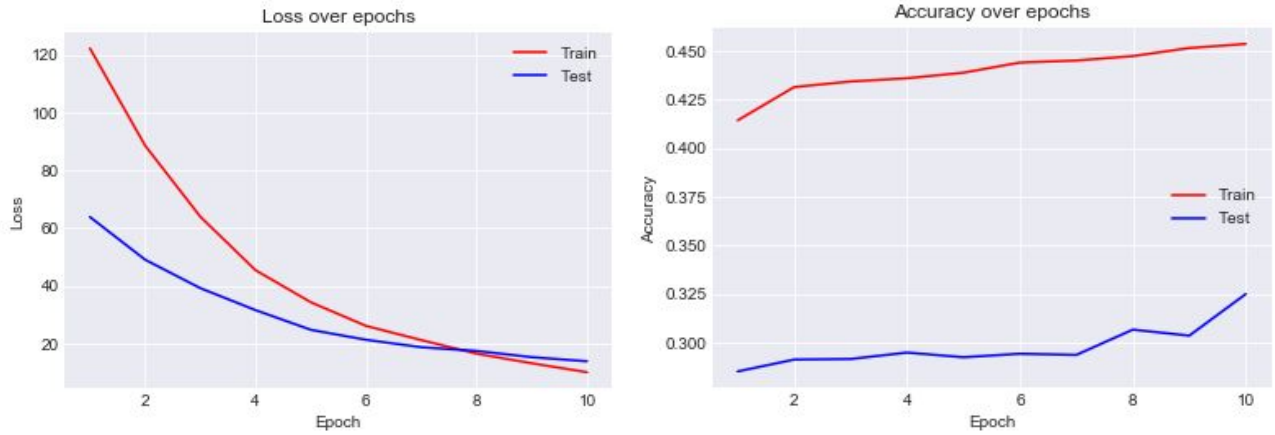
*FFNN* - We ran various FFNN models changing values for epochs,learning rate and then recorded performance metrics to compare the models. The results of our runs are documented below in the the table -

**Table III - FFNN Model Comparison for various iterations**

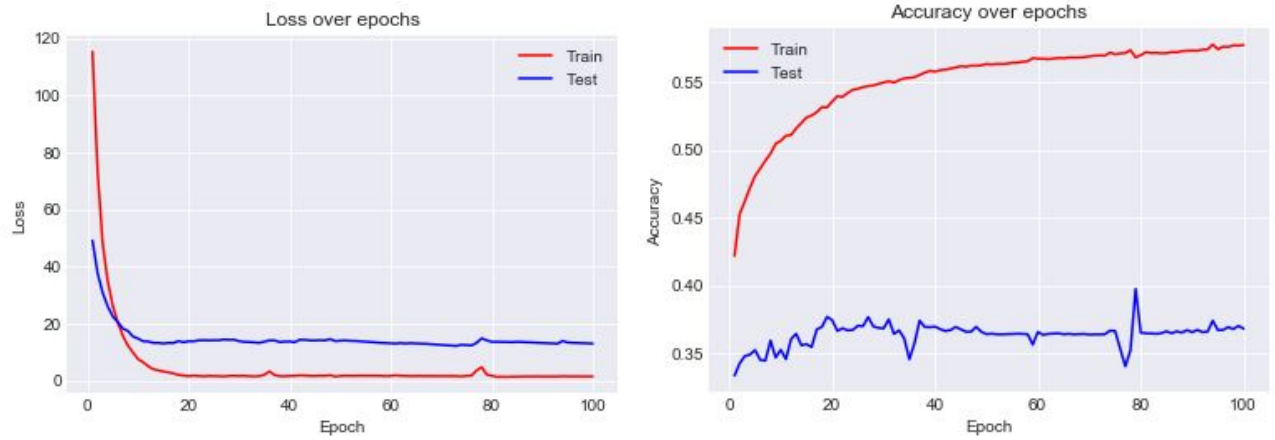
| FFNN Model | Epochs | Learning Rate | Batches | Test Accuracy | Test Loss | Time (sec) | Optimizer |
|---|---|---|---|---|---|---|---|
| Model 1 | 10 | 0.01 | 100 | 32.49% | 13.75 | 1.97 | Gradient Descent |
| Model 2 | 100 | 0.01 | 100 | 36.81% | 13.39 | 14.47 | Gradient Descent |
| Model 3 | 1000 | 0.01 | 100 | 51.00% | 18.56 | 147.42 | Gradient Descent |
| Model 4 | 10 | 0.001 | 100 | 30.55% | 13.34 | 1.89 | Gradient Descent |
| Model 5 | 100 | 0.001 | 100 | 37.35% | 10.15 | 13.74 | Gradient Descent |

Below are the plots that record the training and testing loss and accuracy over changing epochs and learning rate.

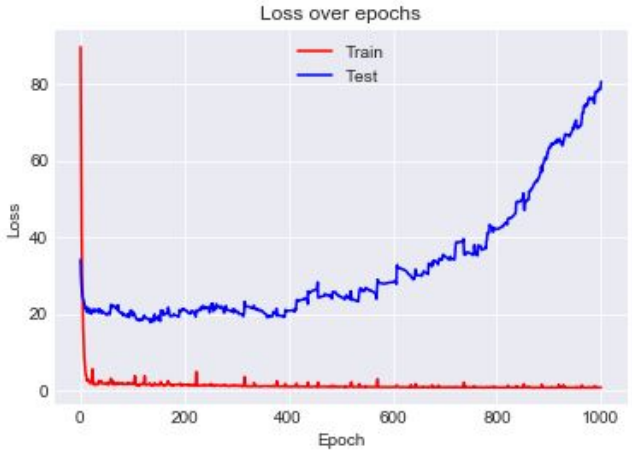
Model 1 -

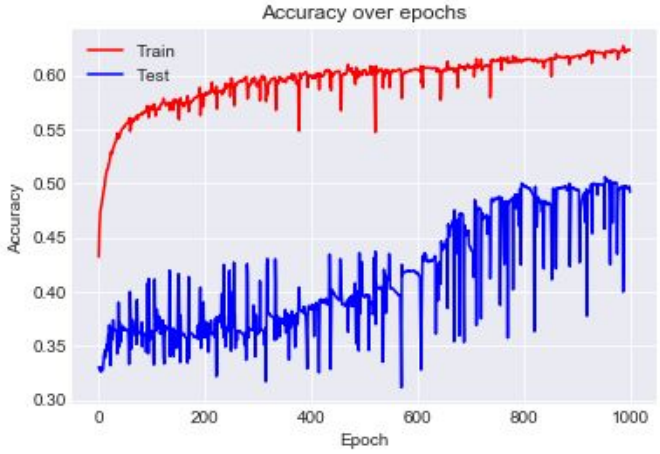



Model 2 -

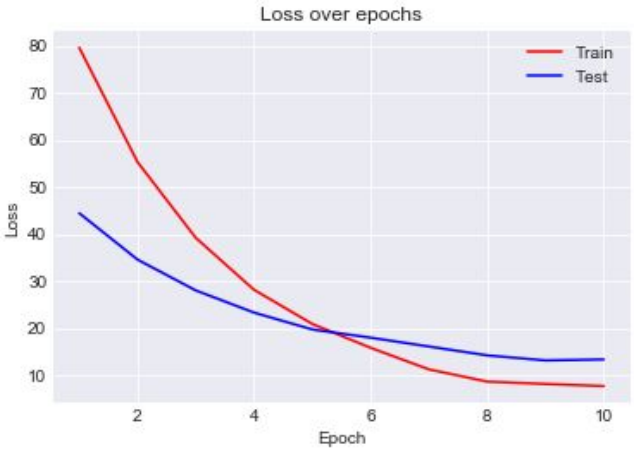


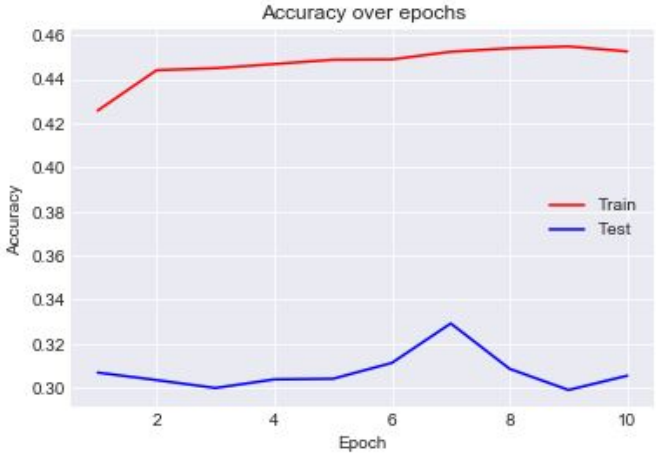
Model 3 - Our best execution had batches of 100 observations and learning rate of 0.01, which we ran over 1000 epochs. This gave us accuracy of 51%



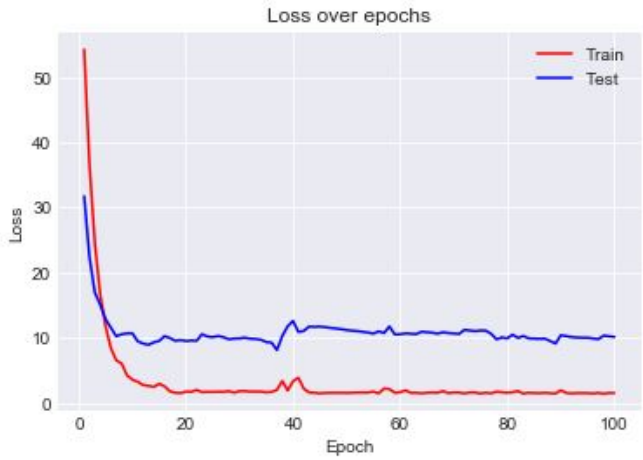



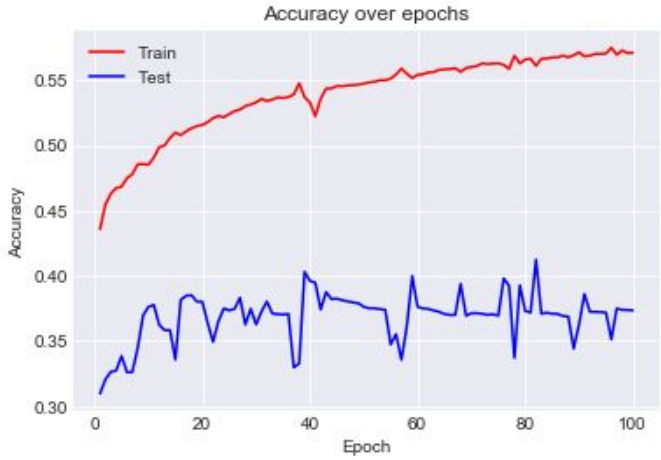

Model 4 -



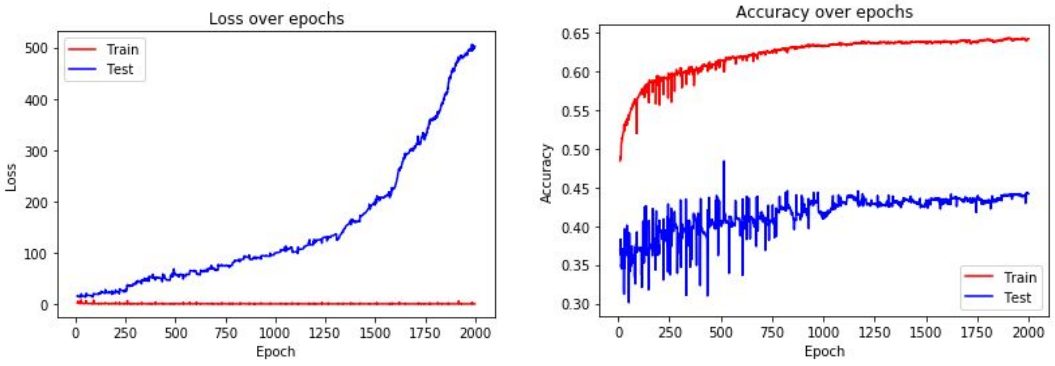




Model 5 -



As we can see from the above plots recording accuracy, as we increase the number of epochs keeping the learning rate constant at 0.01, the test accuracy increases considerably from epochs = 10 to 100 to 1000. Beyond epochs = 800, we see the test accuracy doesn't increase too much. We also increased the number of epochs to 2000 using original best model settings and observed that after about 1000 epochs the test accuracy starts to go down slightly and saturate. Since the accuracy goes down on increasing it beyond 1000, we can say that it is the point where the model starts to overfit. Keeping that dynamic in mind, we decided to choose epochs = 1000 as an appropriate cut off. Below is the plot with epochs = 2000.



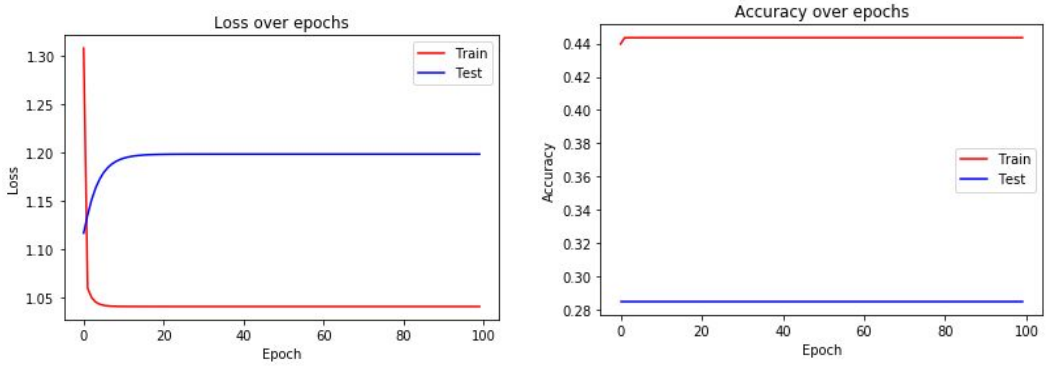

*CNN* - We ran various CNN models changing values for epochs, learning rate and then recorded performance metrics to compare the models. The results of our runs are documented below in the the table -

**Table IV - CNN Model Comparison for various iterations**

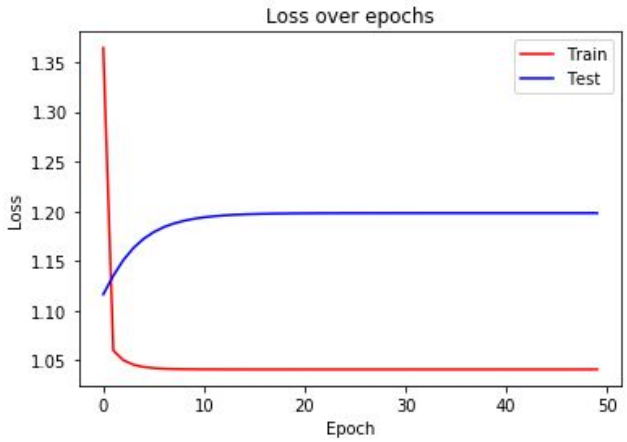
| CNN Model | Epochs | Learning Rate | Batches | Test Accuracy | Test Loss | Time (sec) | Optimizer |
|---|---|---|---|---|---|---|---|
| Model 1 | 100 | 0.01 | 100 | 28.52% | 1.20 | 103.18 | Gradient Descent |
| Model 2 | 50 | 0.01 | 100 | 28.52% | 1.20 | 103.18 | Gradient Descent |
| Model 3 | 50 | 0.01 | 100 | 28.49% | 1.19 | 215.86 | Adam |
| Model 4 | 50 | 0.01 | 100 | 28.47% | 1.20 | 202.78 | RMS Prop |
| Model 5 | 20 | 0.001 | 100 | 28.54% | 1.14 | 58.28 | Gradient Descent |
| Model 6 | 20 | 0.01 | 100 | 28.51% | 1.20 | 90.93 | Gradient Descent |
| Model 7 | 15 | 0.01 | 100 | 28.51% | 1.20 | 45.16 | Gradient Descent |

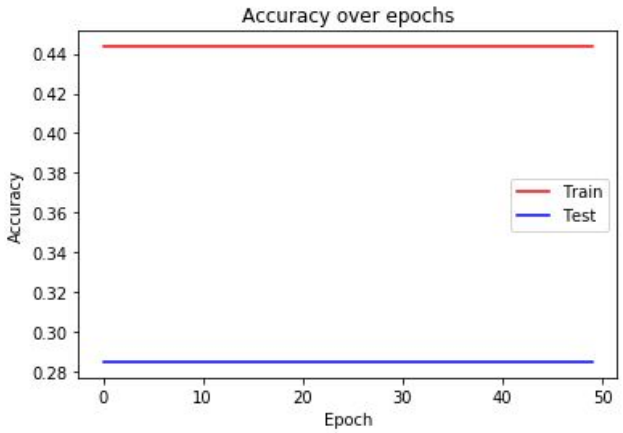
Below are the plots that record the training and testing loss and accuracy over changing epochs and learning rate

Model 1 -

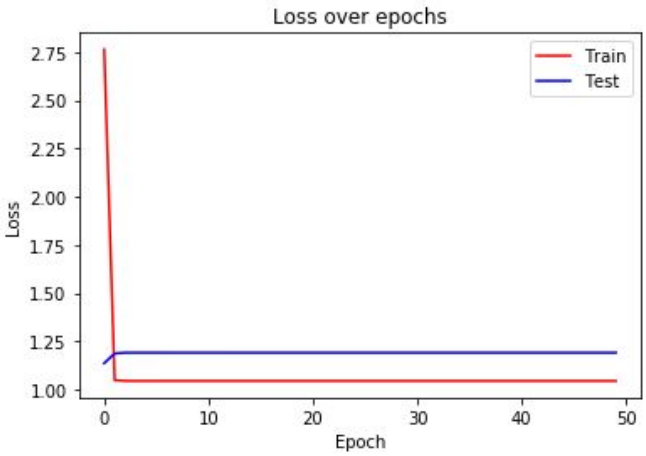


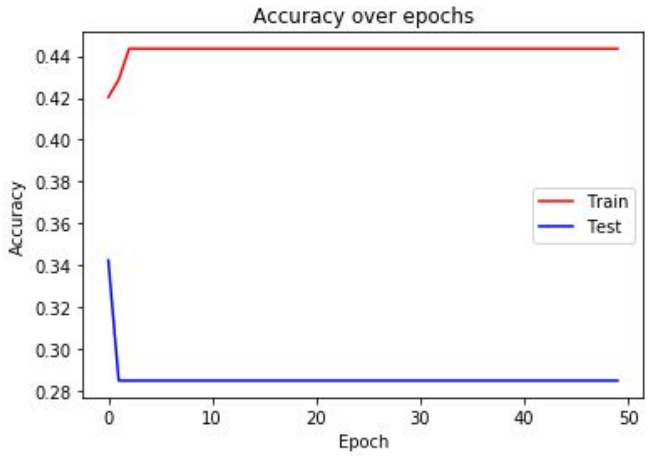
Model 2 -



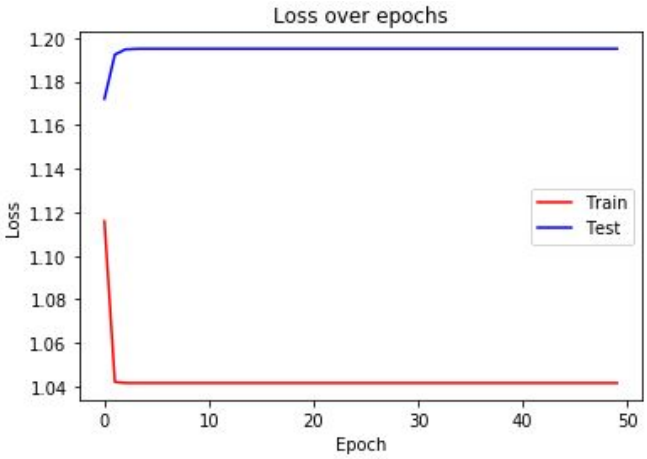



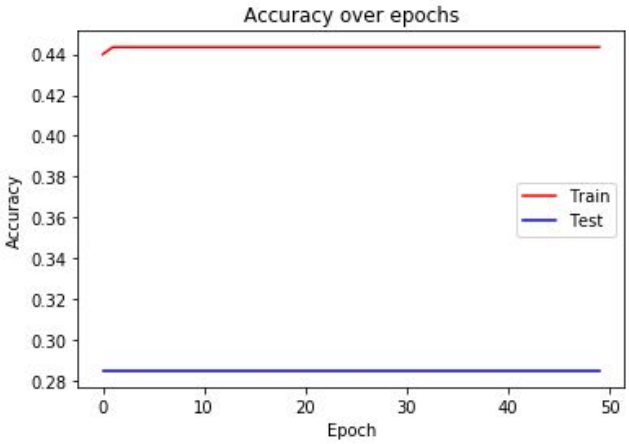

Model 3 -



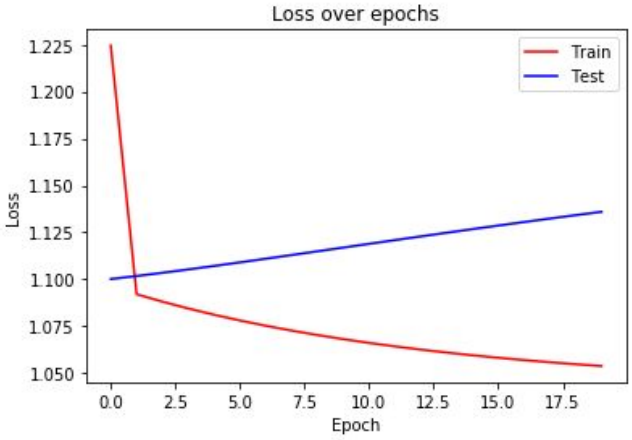

Model 4 -

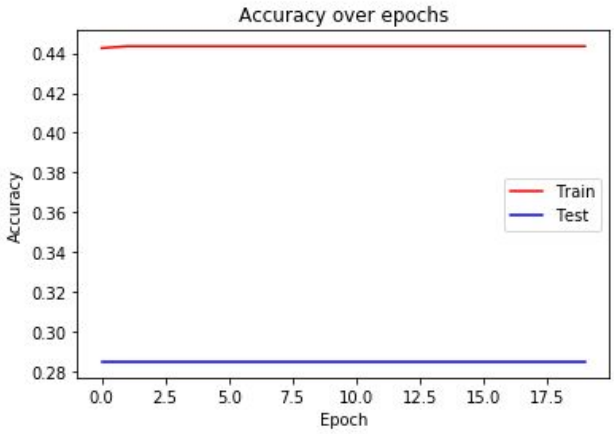


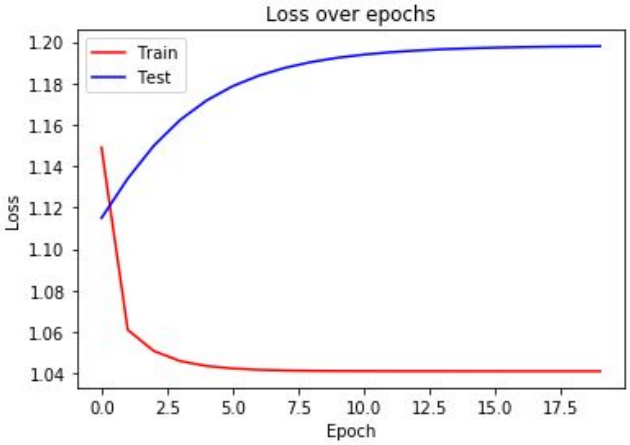
Given that Gradient Descent Optimizer gives us higher accuracy with lower computational expense, we continue testing it on lower number of epochs to improve efficiency:

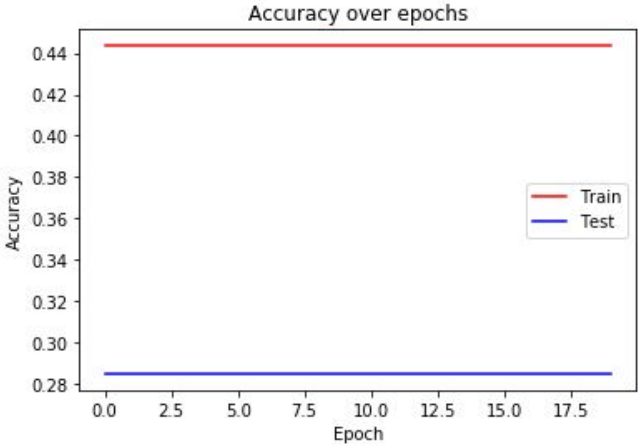
Model 5 -



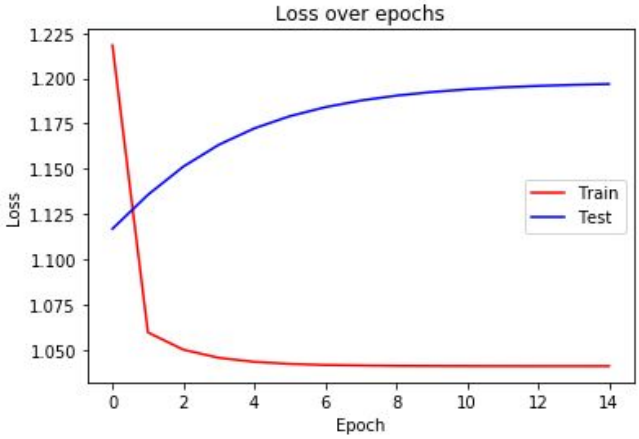



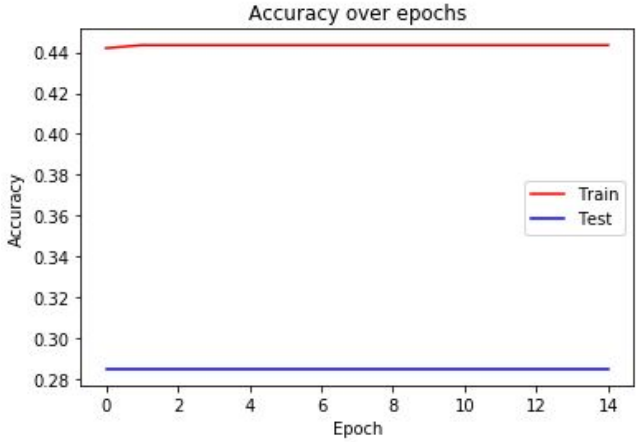

Model 6 -




Model 7 -



We end up choosing Model 7 as our best CNN model - it gives an optimal accuracy and is least computationally expensive than all the previous models. We do however notice that FFNN Model 3 still gives us a significantly higher test accuracy and consider it the best model out all of FFNN and CNN models we tested so far.

*RNN* - In addition to CNN, we tested our results with RNN modes with different number of neurons, learning rates and interactions, and then recorded performance metrics to compare the models. The results of our runs are documented below in the the table -

**Table V - RNN Model Comparison for various iterations**

| RNN Model | Neurons | Learning Rate | iterations | Test Accuracy | Test Loss | Time (sec) | Optimizer |
|---|---|---|---|---|---|---|---|
| Model 1 | 10 | 0.001 | 1000 | 42.22% | 6.56 | 15.01 | Adam |
| Model 2 | 100 | 0.001 | 1000 | 41.22% | 3.34 | 14.97 | Adam |
| Model 3 | 100 | 0.01 | 1000 | 42.66% | 2.81 | 14.83 | Adam |
| Model 4 | 200 | 0.001 | 1500 | 40.21% | 2.33 | 24.09 | Adam |
| Model 5 | 500 | 0.001 | 1000 | 41.55% | 2.61 | 28.26 | Adam |
| Model 6 | 500 | 0.001 | 1000 | 27.66% | 2.31 | 26.80 | Gradient Descent |
| Model 7 | 1000 | 0.001 | 2000 | 37.73% | 1.81 | 27.20 | Adam |

From the table above we can see that although RNN results were significantly better than those for CNN, they are still inferior to the best model results that we have received from running FFNN Model 3. As a result, we choose FFNN Model 3 as our final model.

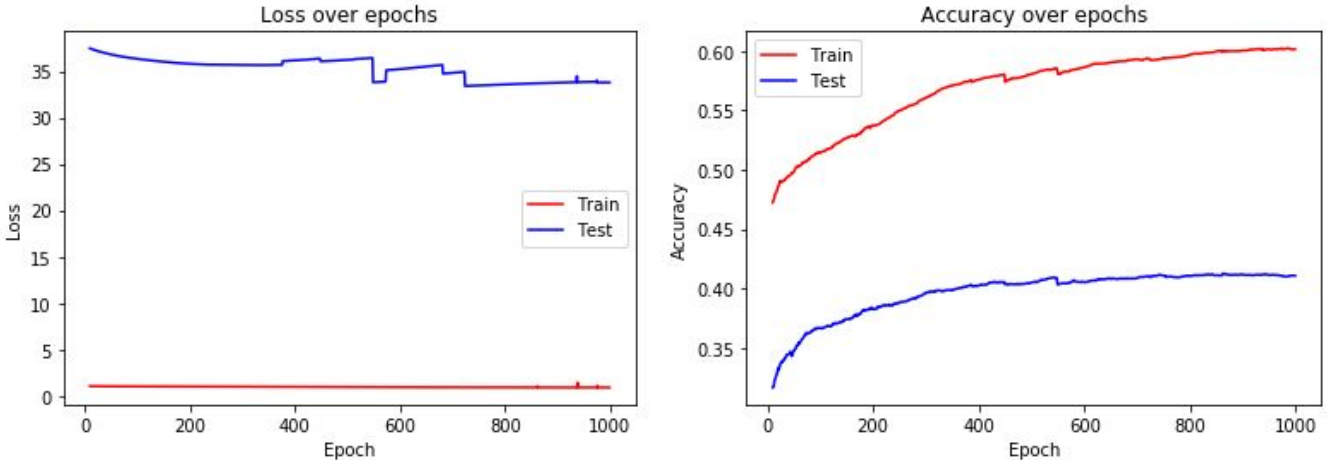
**Improvement**

The models can be tuned in a variety of ways to enhance performance. The list is exhaustive and is something that we will keep experimenting with during the course of this analysis. For now, the improvements we've incorporated are adding a hidden layer, increasing number of epochs along with it and changing the optimizer.We ran our best model (Model 3) with epochs = 1000, learning rate = 0.01 and batches = 0.01 with an additional hidden layer (total 2 hidden layers) and changed the optimizer to Gradient Descent and recorded model performance (Model 6), the details of which are recorded below in the table -

**Table VI - FFNN Model Comparison post improvements**

| FFNN Model | Epochs | Learning Rate | Batches | Test Accuracy | Test Loss | Time (sec) |
|---|---|---|---|---|---|---|
| **Model 3** (original best model, 1 hidden layer, Adam Optimizer) | 1000 | 0.01 | 100 | 51.00% | 18.56 | 147.42 |
| **Model 6** (2 hidden layers, Gradient Descent Optimizer) | 1000 | 0.01 | 100 | 41.08% | 33.78 | 148.52 s |

We observe that adding an additional hidden layer and changing the optimizer didn't really improve the model performance. Below is the plot of Model 6 -




**References**

[1] A. Obizhaeva and J. Wang, "Optimal trading strategy and supply/demand dynamics," J. Financial Markets, vol. 16, pp. 1–32, 2013.

[2] R. Cont, A. Kukanov, and S. Stoikov, "The price impact of order book events," J. Financial Econometrics, vol. 12, pp. 47–88, 2014

[3] R. Cont, "Statistical modeling of high frequency financial data: Facts, models and challenges," IEEE Signal Process. Mag., vol. 28, no. 5, pp. 16–25, Sep. 2011.

[4] Tsantekidis, Avraam, Nikolaos Passalis, Anastasios Tefas, Juho Kanniainen, Moncef Gabbouj and Alexandros Iosifidis. "Using Deep Learning for price prediction by exploiting stationary limit order book features." *CoRR* abs/1810.09965 (2018): n. pag.