Long Term Stock Market Classification

Agamdeep S. Chopra

05/01/2021

BIA 610

Machine Learning Final Project

Motivation

- Short term forecasting has seen some significant improvements in recent years with the advent of text mining and sentiment analysis of social media such to aid stock price prediction. Yet, long term forecasting remains much challenging due to shear computation complexity and uncertainties.
- In this project, I will try to prove that tax returns of listed companies can be a good indicator for predicting long-term performance of stocks.
- The key hypothesis of this project is to predict whether a stock is safe for investment and will yield net profit after a year or not.
- This was done by analyzing and preprocessing historic tax returns of all the corporations listed on NYSE to train an ML model that predicts the likelihood of the stock yielding profit for the invertor in a year.

Dataset Explanation (Initial Setup)

- The original dataset was collected from Kaggle. It is my understanding that the dataset was populated by a script that scrapes information from publicly available tax returns of corporations listed on the NYSE stock exchange between and including the years 2014-2018.
- Data was manually cleaned using Python to replace empty features and headers from the dataset with an appropriate value for training. Note that features here represents the financial indicators scrubbed form tax returns.
- The dataset order was then randomized and split into Train, Test, and Validation sets with 95%, 2.5%, and 2.5% of the original dataset.
- These subsets were then exported in .csv format for further processing.

Dataset Explanation (Preprocessing)

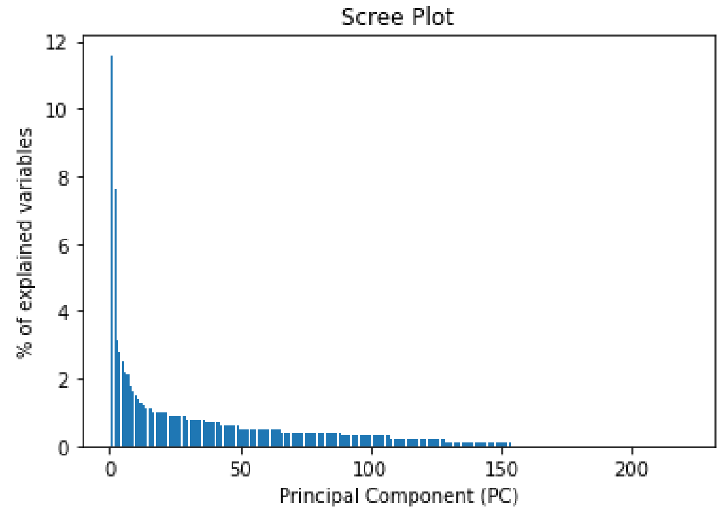
- The datasets were of the shape ***m*** *x* ***d*** where ***m*** in the # of examples in the set and **d** is the # of features + the expected binary output, which is always the last column the way I prepared the datasets.
- So, theoretically the shapes of the sets should be of the shapes:
  - Train -> $(0.95 \bullet \boldsymbol{m}) \times \boldsymbol{d}$
  - Test -> $(0.025 \bullet \boldsymbol{m}) \times \boldsymbol{d}$
  - Val -> $(0.025 \bullet \boldsymbol{m}) \times \boldsymbol{d}$
- The data was then processed as explained in the following slides.

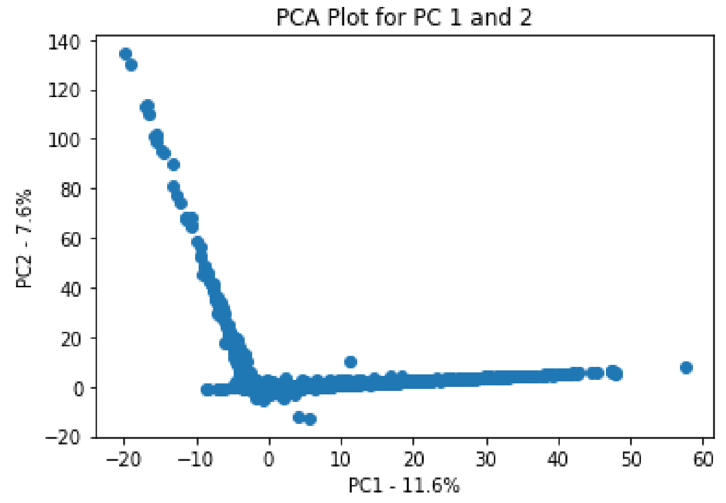
Outlier Elimination

- Data was passed through a custom outlier detection function that detected and eliminated any examples that contained values of features greater than the user specified standard deviation.
- Here, I assumed the datasets followed a gaussian distribution for simplicity.

Feature Dimensionality Reduction through Principal Component Analysis

- PCA was performed on the entire training dataset in order to reduce feature dimensionality and visualize the dataset's clustering patterns.
- Top k features of the model were then extracted from this information for model generation. Here, I used 14^2 top features. We will see the choice of features in later slides.
- PC1 and PC2 were then used to visualize the dataset.





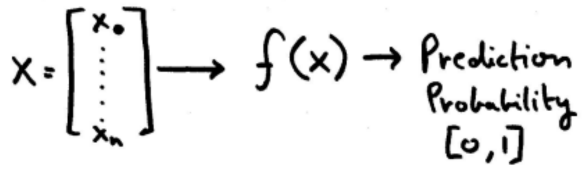
The Models

- 2 benchmark models were created. These were:
  - Multiple Linear Regression
  - Perceptron (MLR with a nonlinearity, in this case sigmoid)
- 2 standard and popular ANNs were then coded, and their nodes and layers were optimized to give the best performance on a subset of the dataset. These were:
  - Shallow NN
  - Deep NN
- Just out of curiosity and personal research interests, I coded a Recurrent Neural Network, and a Convolutional Neural Network.

The Models (Benchmark Models)

- Multiple Linear Regression
- Perceptron (MLR with a nonlinearity, in this case sigmoid)

MLR →

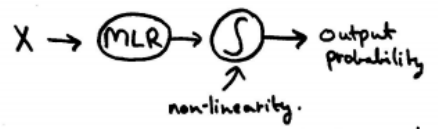


$$X = \begin{bmatrix} x_0 \\ \vdots \\ x_n \end{bmatrix} \longrightarrow f(x) \rightarrow \text{Prediction Probability } [0,1]$$

$$f(x) = \omega_0 x_0 + \omega_1 x_1 + \ldots \omega_n x_n + b$$

here, $\{\omega_0, \ldots \omega_n\}$ and b are trainable parameters.
weights ↑ bias ↗

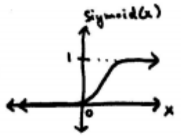
Perceptron →



$$X \rightarrow \text{MLR} \rightarrow \int \rightarrow \text{output probability}$$

↗ non-linearity.

ex- $\text{Sigmoid}(x) = \dfrac{1}{1+e^{-x}}$ →



So, expanding, (lets call sigmoid(x) as g(x)),

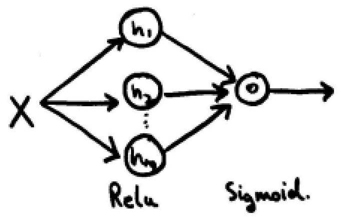
output prob → $y = g\left(\sum_n \omega_n x_n + b\right)$

where ω_n & b are trainable params.

The Models (Dense Networks)

For further insights and derivations, please refer to my GitHub repository titled deep-learning: *https://github.com/AgamChopra/deep-learning/tree/master/Intro%20to%20Deep%20Learning*

Shallow NN →

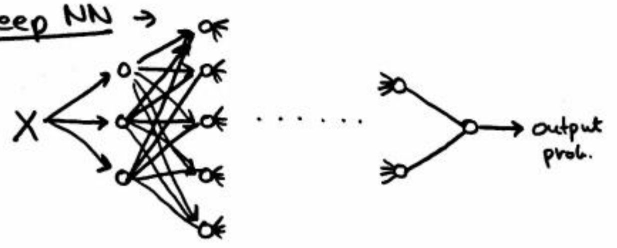


output probab. $= Y = g_o\left(\sum_m w_m g_m\left(\sum_n x_n + b_h\right) + b_o\right)$

Shallow NN introduces a hidden layer that comprises of "neurons"/nodes that themselves structurally the same as a Perceptron but usually have a different non linearity than Sigmoid. In our case, its ReLU non-lin.

The idea is that the individual nodes can "learn" to "specialize" in "seeing" different patterns & thus in theory improving performance.

Deep NN →



Same as shallow but with more than 1 hid. layers. Idea is that lower layers extract low level info & higher up you go, more complex information is being processed by the 'neurons'.

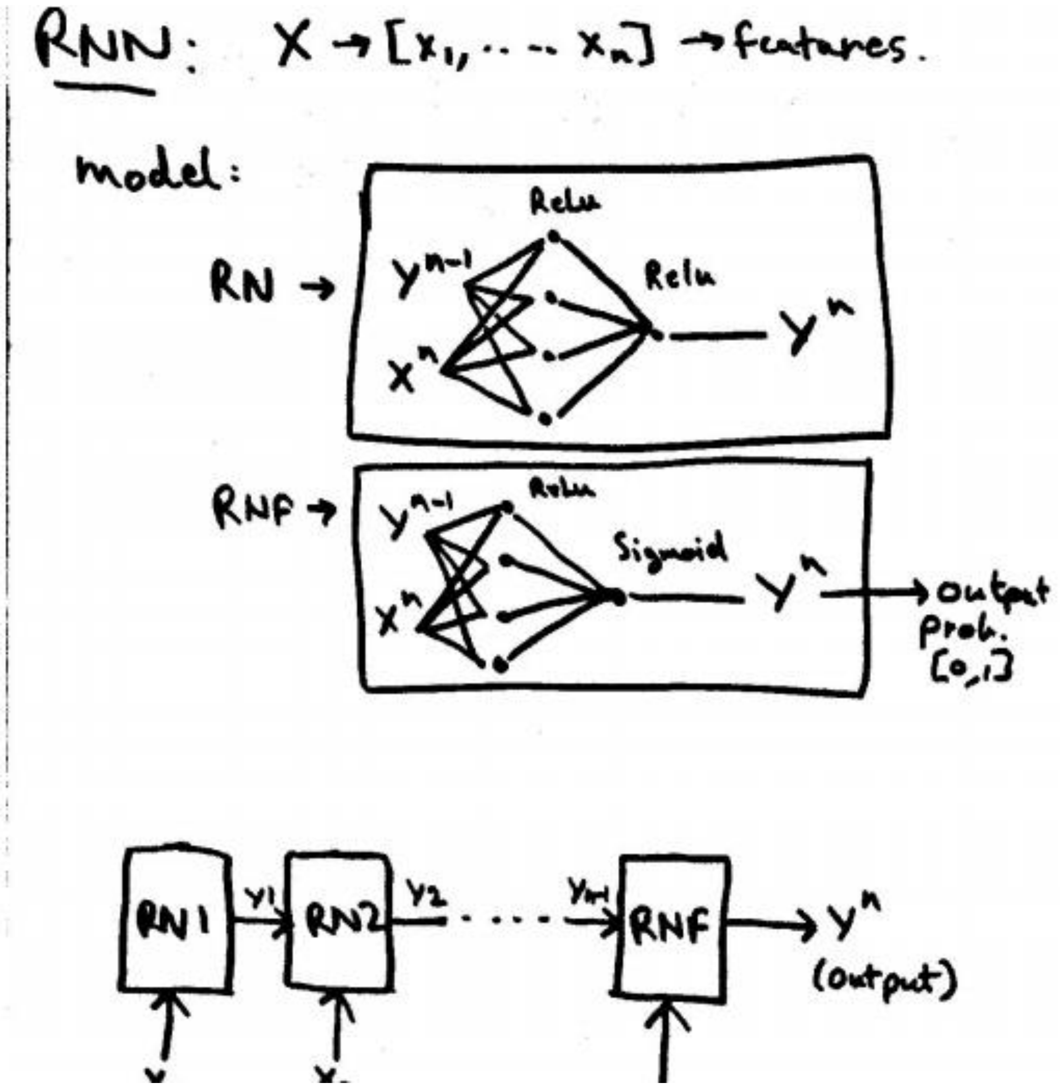
Deep NN can significantly improve a model's ability to extrapolate otherwise abstract patterns.

General rule of thumb, as L ↑, # nodes per layer can ↓ since individual neuron specialization is not required. Thus we are able to extract more abstract information.

output $Y = g_o\left(\sum w^{*} g^{*}_{-1}(\cdots\cdots) + b_o\right)$

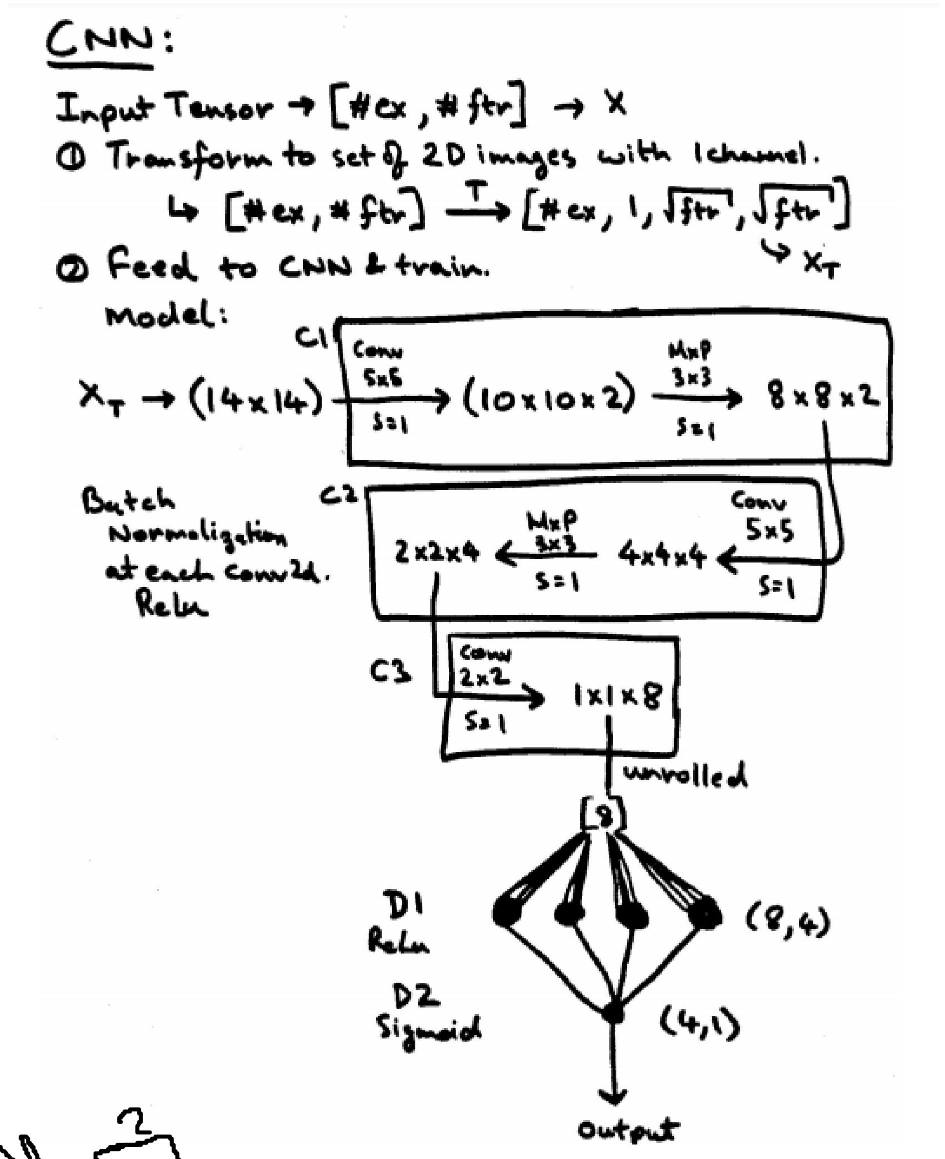
The Models (RNN)

- The features are sent one by one through each Residual Node.
- Each Residual Node except the first node has “memory”. That is, the nodes take both current feature and the previous node’s transformation.



The Models (CNN)

- Converted 1D data to 14x14x1 images and fed through a CNN.
- To keep the CNN architecture easy, I retroactively adjusted the PCA filtered dataset to $14^2 = 196$ features.
- Key Takeaway: This dataset can be transformed into an image classification problem with very promising results.
- Surprisingly performed the best.



CNN:

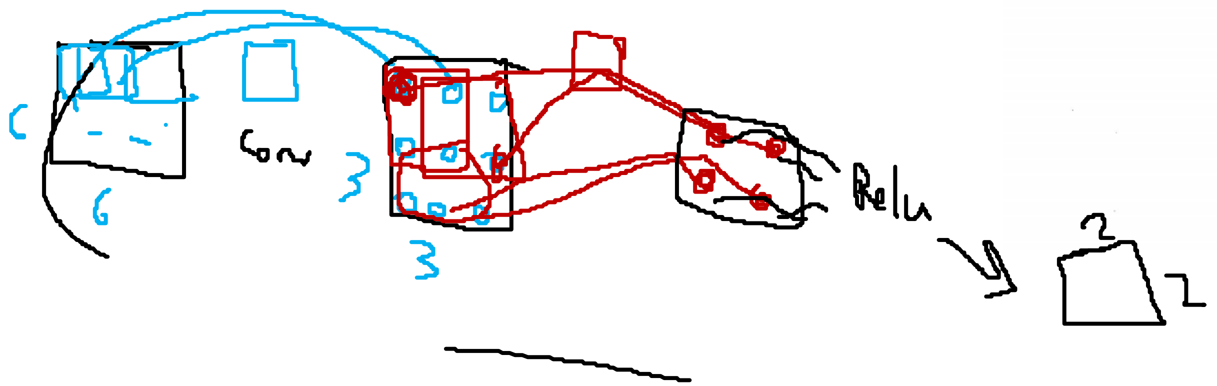
Input Tensor → [#ex, #ftr] → X

① Transform to set of 2D images with 1channel.

↳ $[\#ex, \#ftr] \xrightarrow{T} [\#ex, 1, \sqrt{ftr}, \sqrt{ftr}]$ ↪ X_T

② Feed to CNN & train.

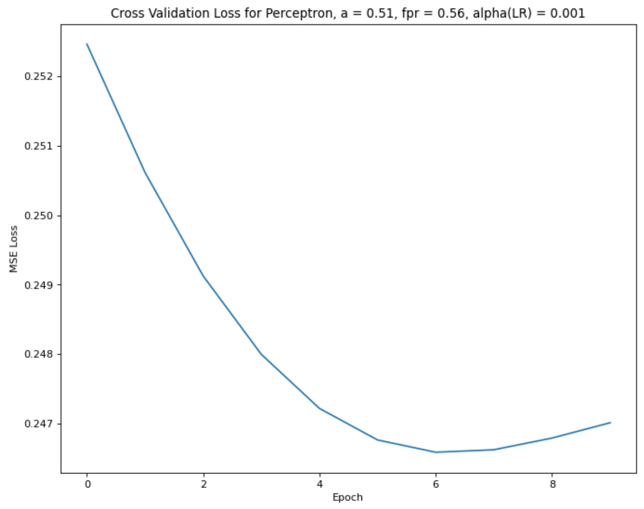
Model:

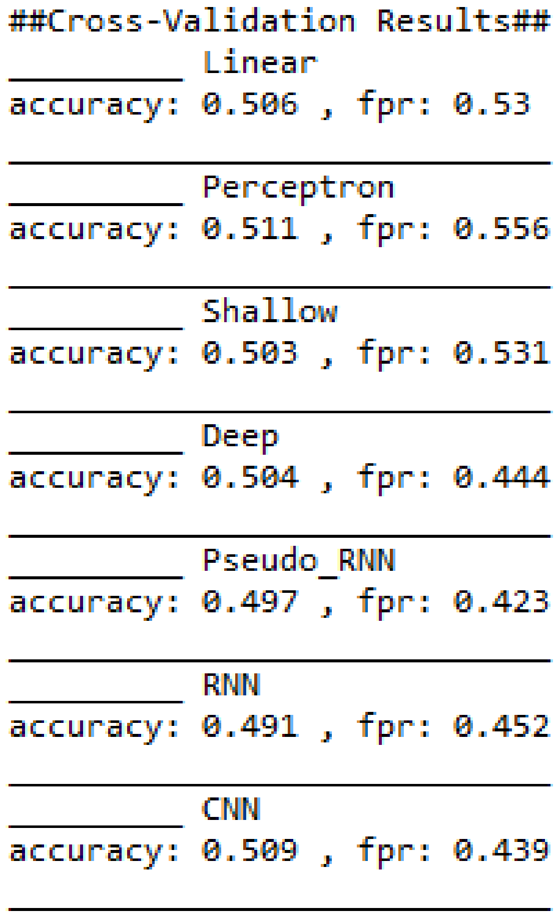


Cross-Validation

- I created a function that inputs a dataset in ***m*** *x* ***d*** format and splits it into k batches.
- It is assumed that the dataset was already randomized.
- The function then returns a dataset in ***k*** *x* ***m/k*** *x* ***d*** format.
- The dataset was then passed to a cross validation function.

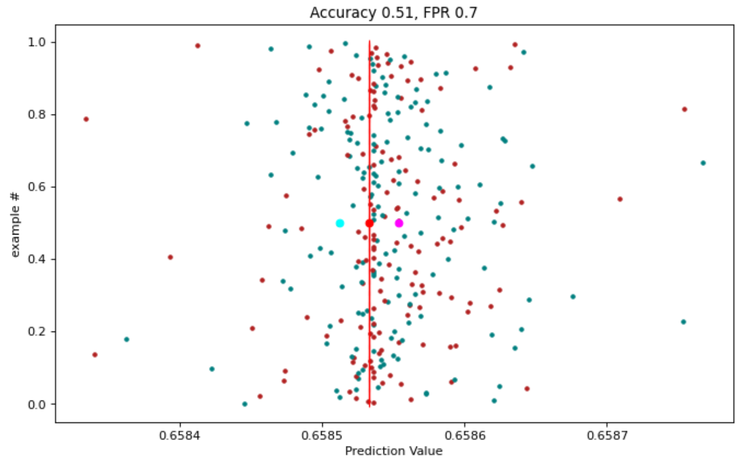
Cross-Validation (Continued)

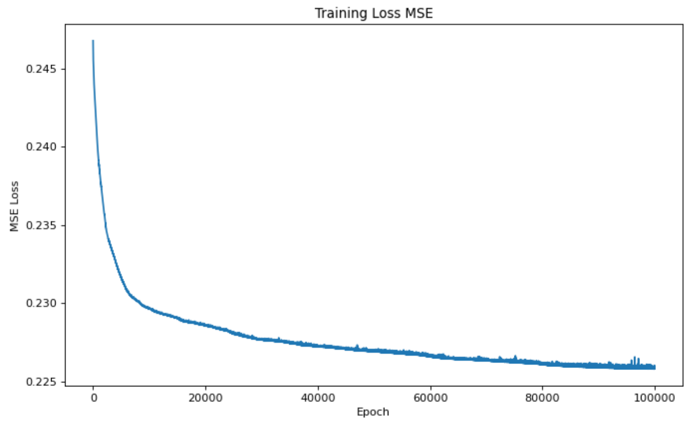


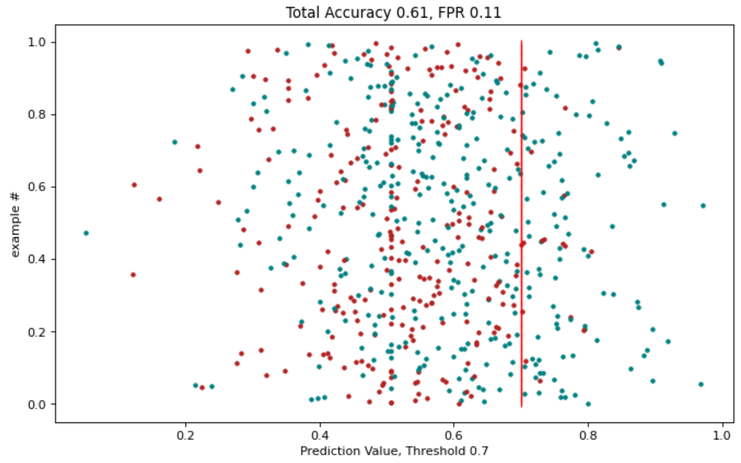


```
##Cross-Validation Results##
__________ Linear
accuracy: 0.506 , fpr: 0.53
_______________________________
__________ Perceptron
accuracy: 0.511 , fpr: 0.556
_______________________________
__________ Shallow
accuracy: 0.503 , fpr: 0.531
_______________________________
__________ Deep
accuracy: 0.504 , fpr: 0.444
_______________________________
__________ Pseudo_RNN
accuracy: 0.497 , fpr: 0.423
_______________________________
__________ RNN
accuracy: 0.491 , fpr: 0.452
_______________________________
__________ CNN
accuracy: 0.509 , fpr: 0.439
_______________________________
```

- The k batches of the dataset was then trained and evaluated against each model for 10 epochs.
- Preliminary metrics for each k-fold were calculated and averaged for each model.
- These metrics alongside the training loss were plotted as shown.
- Model with the most optimal performance in Accuracy and FPR was then selected for training. In this case it was the CNN model.







Training, Evaluation/Observation, and Conclusion

- The CNN model was trained with MSE Loss for 1000 epochs with ADAM optimization, LR = 0.0001, and regularization = LR/100.
- Similarly, the model was then trained with BCE Loss with the same parameters.
- Interestingly, MSE loss model performed slightly better than BCE loss model.
- Finally, the model was then trained with the same parameters and MSE loss for 100,000 epochs. This took quite a while… The trained model was saved as cnn.pth and cnn.txt for future access.
- Model threshold was optimized against the test dataset and test metrics.
- Finally, the model was evaluated on the validation set and the metrics were averaged over.
- The final model gives an accuracy of ≈ 60% and FPR of ≈ 10%. This means that, **for any positive prediction of the model, we can say with 90% confidence that the predicted stock will return a net profit after 1 year.**

Insight, Outcome, and Suggestions

- We prove that tax returns can be a decent indicator for predicting long term stock performance.
- Thus, it is my suggestion that by adding tax return information to a dataset for stock price prediction might be a good way to boost a model's predictive performance for both short-term and long-term forecasting.
- One thing I would do differently is to dedicate more time for data preparation and preprocessing.
- It would be interesting to see more complex models applied to this dataset.
- Perhaps use GPU instead of CPU and C++ instead of Python to speed up training and overall program efficiency.

Thank you!

If you have any questions regarding anything in this project, please feel free to reach out to me at achopra4@stevens.edu