Comp 380

Fall Semester

Stock Propagation

Giacomo Radaelli, Liam Sefton, Dalton Kohl

Professor Jiang

12/15/21

Abstract

In this report we detail the data mining, data preprocessing, network architecture, experimentation and results from our implementation of a backpropagation neural network designed to predict one year price increases for a stock. We mined the stock data from the yfinance python library that interacts with the Yahoo Finance API. Data preprocessing included scaling of values, cleaning of data and testing set selection. The neural network employs 2 hidden layers, the tanh activation function and the Adam learning algorithm. It was trained on 700 samples from 400+ stocks in the S&P 500 and tested on 200 stocks in the S&P 500. The net had an average absolute difference from the target % increases of around .2 and could predict a 10% increase in a stock 73% of the time.

Keywords

Machine Learning, Data Mining, Finance, Backpropagation, Adam, Stock Price Prediction, Neural Network

Introduction

The goal of this project was to investigate a value investing approach to stock price prediction using neural networks. Most algorithmic trading done today is done by trading huge volumes based on technical analysis and Long Short Term Memory chain sequence analysis. Although technical analysis does give some hints to intraday, day to day and week to week price movements, the market remains unpredictable in the short term to a great degree. Many institutional investors engage in value investing, which follows the philosophy that the market price always trends towards the true value of an equity. We thought that if we input the correct data for a company (from annual reports and price history), the neural net could act as a sort of advanced modelling algorithm that would make comparable value predictions to things such as the Discounted Cash Flow model. If the markets do trend towards the true value of an equity over time, the net should experience a reduced degree of randomness when compared to short term price movements, which is the purpose of the value investing approach. This approach yielded impressive (to us) results and was able to predict a 10% increase in the price of stock 73% of the time.

Background

We began by extracting stock data from the Yahoo! Finance database. We decided to focus on the S&P 500 companies, selecting specific inputs from each stock's cashflow, balance sheet, and financial statement. To do so (in a pythonic manner), we utilized the yfinance API, an open-source tool that uses Yahoo's publicly available APIs and allows users to download market data from Yahoo! Finance. Within this API, several attributes and methods proved useful to extract our desired datapoints from each financial document. A module called the Ticker module implemented these methods, and using this module, we created a miner program in which we outputted a csv file containing 1500 samples (each sample contained data for 20 inputs). Each stock was described by 3 samples, where each sample represented data from a different year.

After mining the raw information of the S&P 500 stocks, we began cleaning our data. There were several cases within our dataset in which lines were too long, too short, or contained numbers with multiple decimal points. After removing these lines, we had to run through our dataset once more as there were still cases in which lines contained “nan” values.

Once we obtained clean data (which resulted in approximately 900 samples remaining), we began preprocessing our data. This involved categorizing our data by rescaling each datapoint, creating a much smaller (and more feasible) range of values. In doing so, we preserved the information provided by each value but also drastically reduced the variance of our data.

After preprocessing our data, we began implementing our neural network, experimenting with the machine learning framework PyTorch. Familiarizing ourselves with this framework involved a very steep learning curve, particularly due to implicit lines of code, making it difficult to decipher what was going on.

Approach

Once we collected our data and outputted it into our csv file, we needed to begin preprocessing the data to format our information in a more useful and efficient manner. We decided to convert our data into more manageable values. Some examples include:

$$Net\ assets\ \ -> \frac{net\ assets}{total\ assets}$$

$$Cash\ \ -> \frac{cash}{total\ assets}$$

$$Current\ liabilities\ \ -> \frac{current\ liabilities}{total\ liabilities}$$

$$Gross\ profit\ \ -> \frac{gross\ profit}{total\ revenue}$$

We converted our pricing data as well, transforming it into percent change values. For example:

$$Price\ 1\ yr\ before\ "current"\ \ -> \frac{"current" - previous\ price}{"current"}$$

$$SPY\ price\ 1\ yr\ before\ “current”\ \ -> \frac{"current" - previous\ SPY\ price}{"current"}$$

$$Price\ 1\ yr\ after\ “current”\ \ -> \frac{(1\ yr\ after\ "current") - "current"}{1\ yr\ after\ "current"}$$

In doing so, we preserved the complete original information while also diminishing the range of our values from [-1000000,1000000] to [-1,2].

Our approach to creating the NN was to decide on an initial set of layers and number of neurons per layer, activation functions, initial weights, learning rate, loss function, and learning algorithm and iterate through changes of these variables choosing the combination that yielded the best results. At first we started with four layers consisting of the input layer having 19 neurons, to the first hidden layers 15 input neurons, to the second hidden layers 10 input neurons, to the output layers 5 input neurons, and finally having a single output neuron, after changing and experimenting with the architecture of the network we found this setup to yield the best results. Initially we used the ReLU activation function, however with experimentation with sigmoid and tanh we found that tanh yielded the best and most consistent results among all activation functions that were tested.

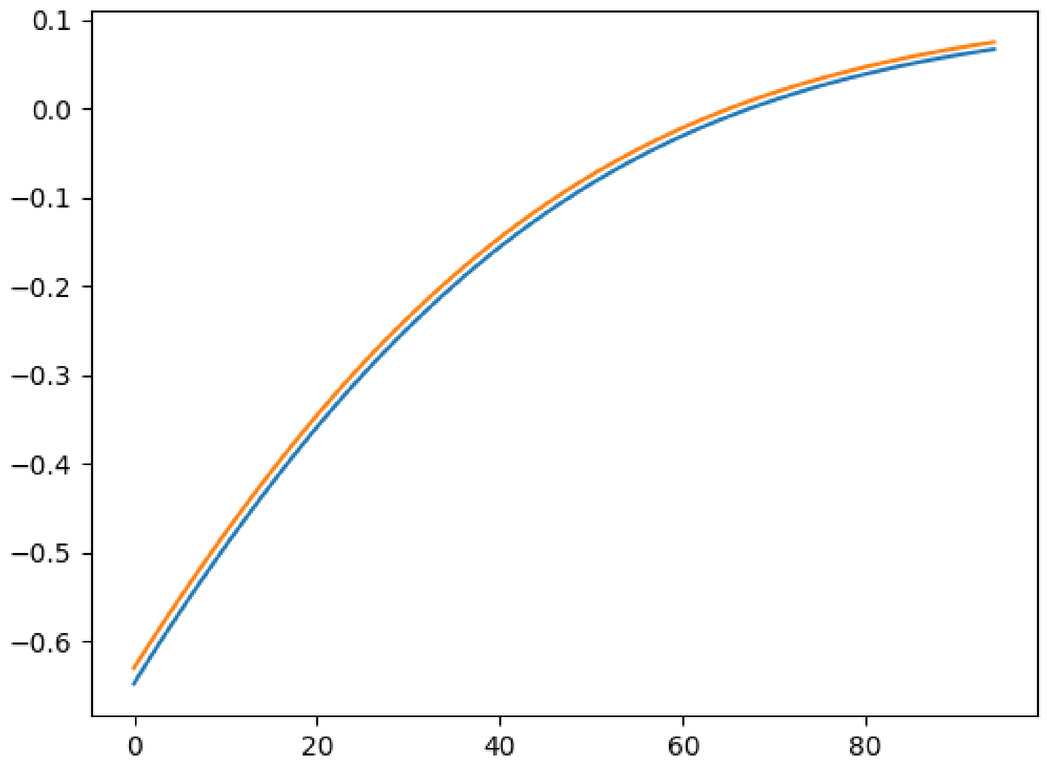

Figure 1: Sigmoid activation function output variance

Figure 1 shows the output variance graph that the Sigmoid activation function produces. The bottom, blue line, represents the minimum value output by the net during an epoch. The top, orange line, represents the maximum value output by the net during an epoch. This graph shows that the maximum and minimum values are stuck together during the training, which we call output clumping. With ReLU, Sigmoid and Leaky ReLU we experienced output clumping.

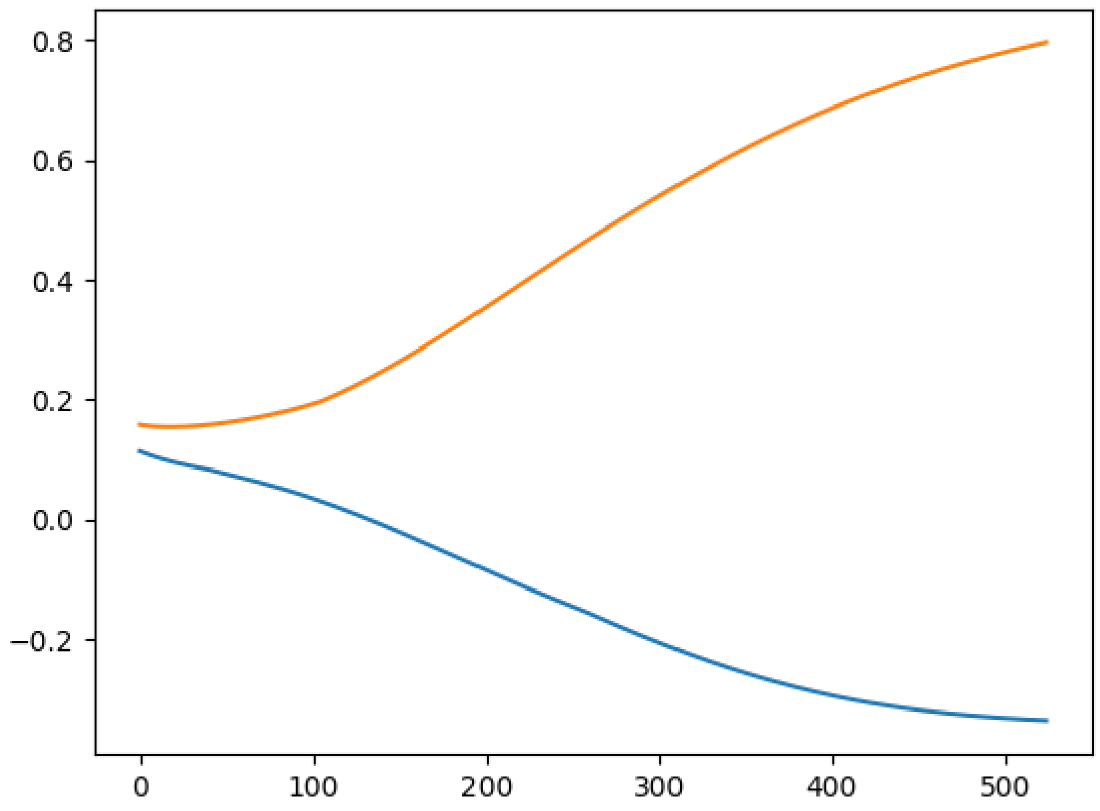

Figure 2: Tanh activation function output variance

Figure 2 shows the output variance for the Tanh activation function, which shows a much wider spread than the Sigmoid variance. This is what we looked for when selecting an activation function to use, as we wanted the range of output values to match that of the dataset. We think that whatever is causing the output clumping from Sigmoid, ReLU and Leaky ReLU has to do with over incentivizing safe predictions that trend towards the mean target value of the dataset.

Throughout this process of testing we switched between setting our initial weights to zero and randomizing our initial weights and found that randomizing our initial weights yielded the best results. We wanted to explore pretraining the weights, but did not have the time to finish that implementation. When choosing our learning rate for the network, we decided upon .00005, we found that any learning rate higher than that would lead to the network diverging quickly and anything lower than that would yield no increase in performance, but a longer time to converge. These results were constant throughout all architectures we tested for the network. When choosing the loss function we started with mean squared error and after testing cross entropy and a manual implementation of mean errors raised to the fourth power which aimed to de-incentivize clumping, however we found cross entropy to have lower results and the manual implementation failed to reduce clumping so in the end we chose mean squared error. Initially we implemented the Stochastic Gradient Descent learning algorithm followed by the AdaMax and Adam learning algorithms, concluding that Adam yielded the most promising results.

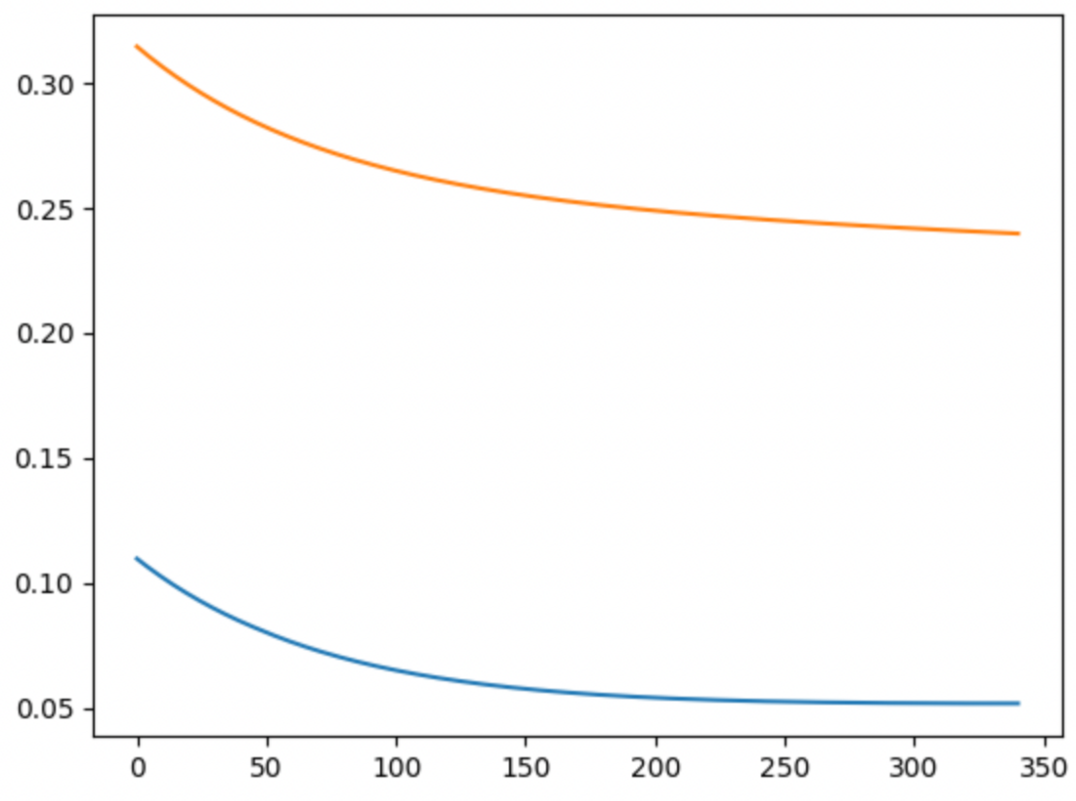

Figure 3: Stochastic Gradient Descent with Tanh activation function output variance

Figure 3 shows the output variance for a fairly long run using tanh activation functions and the SGD learning algorithm. The output lines move in parallel, meaning that the outputs themselves are most likely moving in unison (a form of output clumping). Figure 2 represents the same architecture and activation functions as Figure 3, but with the added change from SGD to the Adam learning algorithm.

In addition to basic validation set overfitting prevention, we also attempted neuron dropout to discourage overfitting. The results were generally poor, with most runs ending early or producing statistically insignificant results (50% accuracy which could be called random guessing).

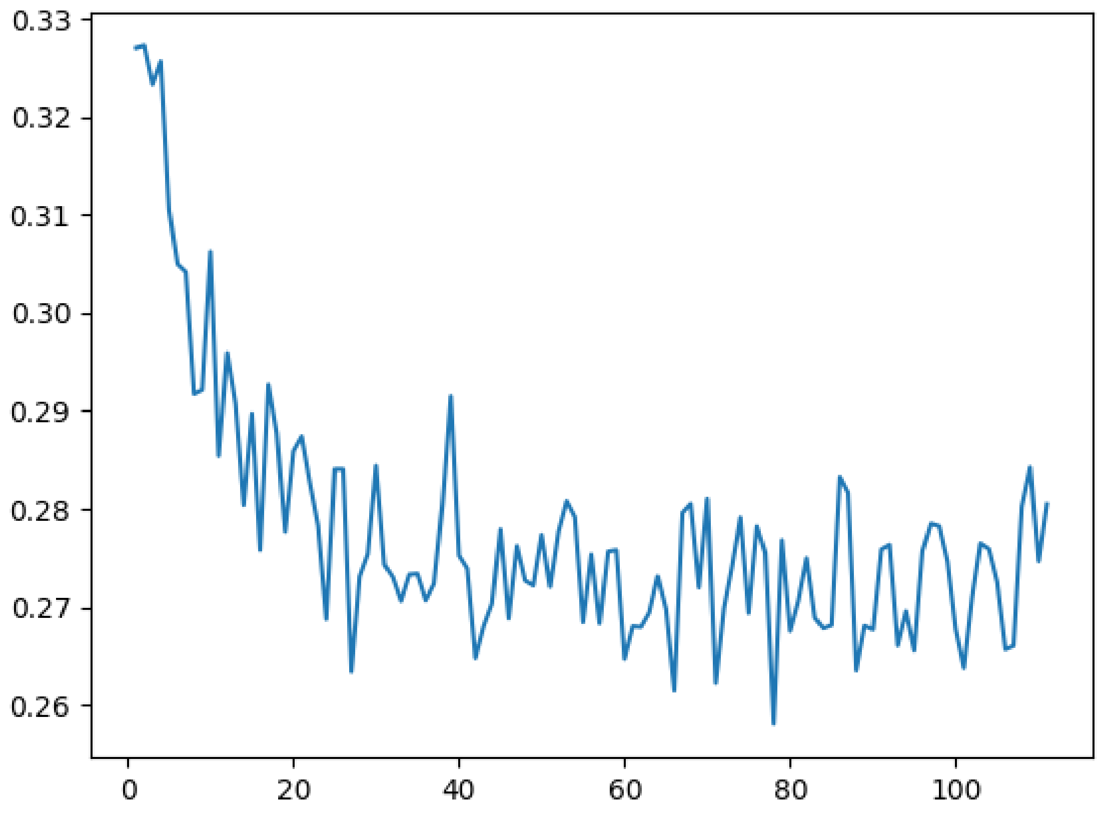

Figure 4: Output of main architecture when dropout is added

Figure 4 shows the average absolute difference between our net's outputs and the validation set target values by epoch. The large degree of variability made it more difficult to detect overfitting and the net generally produced a worse performance than without dropout. We also tried dropout of varying degrees, experimenting with dropout rates as low as .1 for only 1 layer, but at that point the results were barely measurable and any higher dropout rates would decrease performance.

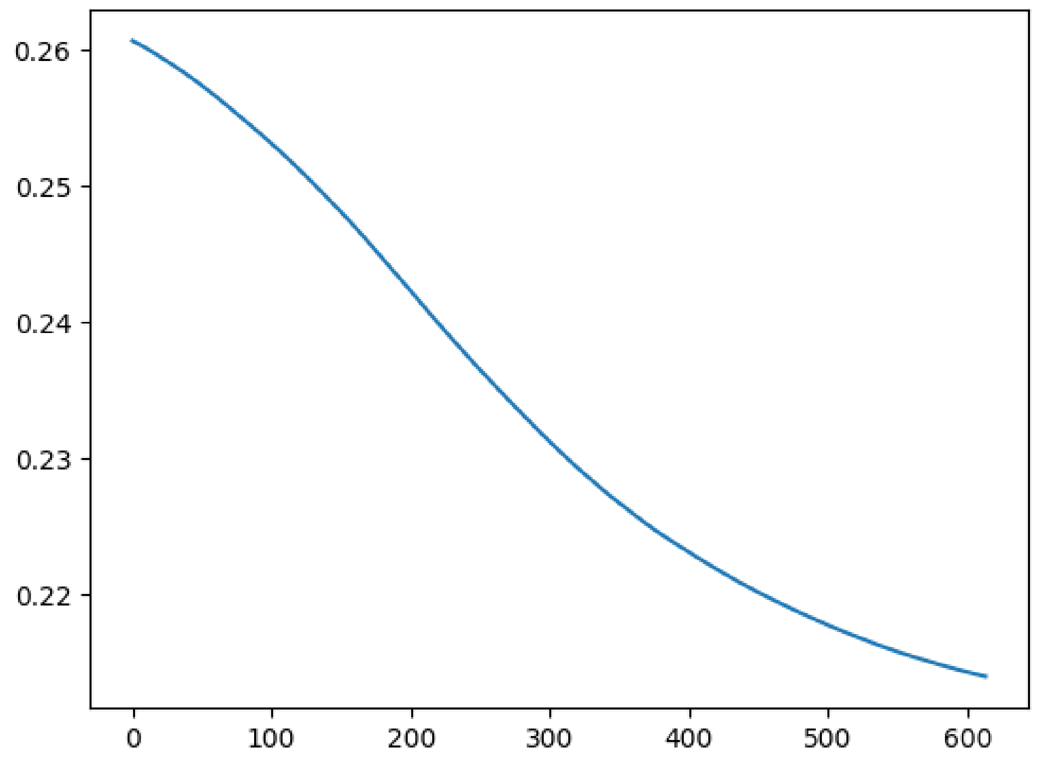

Figure 5: Output of main architecture

Figure 5 shows the average absolute difference between our final net's outputs and the validation set's target values. This is our best looking graph, showing some of the momentum of the Adam algorithm at play, starting off slow and gaining steepness towards the middle before tapering off again. The net converges to somewhere around .2-.21, meaning the net is off by an average of .2 from the actual increases in price at the end. When applied to the binary datasets though, it shows what kind of accuracy an average difference of .2 can result in.

With the final implementation of the network we found that the network was able to predict a 10% year to year increase in stock price with 73% accuracy. After doing 100 test runs we were able to achieve a 99% confidence interval on [0.7291, 0.7299]. We created another binary data set with any target values above 0 being set to 1 and any target values set below 0 to 0 allowing for a more broad acceptance rate, compared to our usual bounds of anything above .1 being set to 1 and anything below set to 0. With this new data set we did the same tests finding that the accuracy increased to about 80%.

Conclusion

Overall, over the course of a month we mined 19 stock attributes of the S&P 500 companies from annual reports and price data using using Yahoo! Finance through the finance API. Using these attributes we were able to create a data set with each line being given an output value (or last column) of the year to year percent increase of stock price. Using this data set we created three subsets of the data consisting of training data, testing data, and binary testing data. With the training data we trained a neural network to predict the year to year percent increase of stock price. The network architecture consisted of 4 layers with 19 neurons, to 15 neurons, to 10 neurons, to 5 neurons, finally having one output neuron which was the predicting percent stock price increase. We found that using random initial weights, the tanh activation function, learning rate of .000005, mean squared error loss function, and Adam learning algorithm yielded the most accurate prediction of stock increase. We predicted its accuracy by first converting anything target value and output above .1 to 1 and below .1 to 0, after doing so we compared the target value to the output value of the network finding that they were equal about 73% of the time after training.

To increase accuracy we plan to make changes to the data, explore pytorch much deeper, and use more computing power. Possible changes to the data include getting more data, consisting of more year to year data of existing companies within the dataset and year to year data of companies that are not in the S&P 500. Getting more year to year data of the existing companies within our data set will lead to an increase in accuracy for the network where adding more companies to our data set will increase the generality of the network allowing for better accuracy across a wide range of companies. With more exploration of the pytorch library we will be able to test a wider range of optimizers, loss functions, and activation functions. In the future we aim to automate the testing of these features which creates the issue of computing power. To attempt future testing more computing power would be needed to get significant results in a more timely manner. Currently it takes about twenty to thirty minutes to do a single run of training and testing, making getting the one hundred run confidence interval very time consuming being approximately fourteen hours. Testing multiple combinations of all the variables that the net consists of would be logistically cumbersome without an increase in computational power to decrease the average run time for the network. This logistical nightmare is why we aim to get more computing power to test the network, possibly buying time through AWS.

Appendix- Process

Github**:** https://github.com/liamsefton/stockpropagation

1. Download preprocessed stock data which can be found within the_chonker.txt file found in the aforementioned github.
2. Create python scripts to separate the data into training and testing datasets.
3. Install pytorch, matplotlib.
4. Copy source code from Appendix A inputting the training and testing data files into file open lines.

5. Run tests on neural the network, varying the number of hidden layers and layer activation functions.
6. Investigate results found and conclude which combination optimizes our neural network.
7. Profit?

Appendix A (Full Code)

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.modules.loss import MSELoss
import torch.optim as optim
import matplotlib.pyplot as plt

class StockNet(nn.Module):
    def __init__(self):
        super(StockNet, self).__init__()
        #This creates the layers
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(19, 15),
            #nn.ReLU(),
            #nn.ReLU(),
            nn.Tanh(),
            #nn.Sigmoid(),
            #nn.Dropout(.4),
            #nn.Linear(20, 15),
            #nn.ReLU(),
            #nn.ReLU(),
            #nn.Linear(15, 15),
            #nn.Tanh(),
            nn.Linear(15, 10),
            #nn.ReLU(),
            nn.Tanh(),
            #nn.Sigmoid(),
            #nn.Dropout(.2),
            nn.Linear(10, 5),
            #nn.ReLU(),
            #nn.ReLU(),
            nn.Tanh(),
            #nn.Sigmoid(),
            #nn.Dropout(.2),
            nn.Linear(5, 1)
            #nn.Tanh()
        )

    def forward(self, x):
        #forward function must be defined, so we give it the layers we
created above
        return self.linear_relu_stack(x)

#this function is used to initialize the weights to a set value to
reduce randomness
line_num = 1
def init_weights(layer):
    if type(layer) == nn.Linear:
        layer.weight.data.fill_(0.0)
        layer.bias.data.fill_(0.0)

sum_of_runs = 0
num_of_counted_runs = 0
for z in range(10):
    model = StockNet()
    #model.apply(init_weights)
    num_epochs = 5000
    optimizer = optim.Adam(model.parameters(), lr=.000005) #changed
from adam
    testing_differences = -1
    curr_epoch = 0

    #variables for pyplots
    testing_error_y_vals = []
    training_error_y_vals = []
    variation_y_vals_high = []
    variation_y_vals_low = []
    x_vals_training = []
    x_vals_testing = []
    x_vals_variation = []
    convergence_counter = 0
    divergence_counter = 0

    plt.xlim(1, 50)
    plt.ylim(0, .5)
    plt.title("Error on validation set")
    plt.xlabel("Epochs")
    plt.ylabel("Average error on samples")


    #Training
    for outer_loop in range(num_epochs):
        if outer_loop % 10 == 0:
            #for group in optimizer.param_groups:
            #    group['lr'] = group['lr'] - (group['lr']/10) #lowers
weights slowly over time
            print(outer_loop)
        #for group in optimizer.param_groups:
        #    group['lr'] = group['lr'] - (group['lr']/100)
        f = open("trainingnew.txt", "r")
        sum_correct = 0
        num_samples = 0
        line_num = 1
        first_run =True
        largest_val = 0
        smallest_val = 0
        for line in f:
            line = line.split(",")
            line = list(map(float, line))
            target = [line[-1]] #target must be in list in order to be
converted to tensor
            target = torch.FloatTensor(target)
            line = line[:-1]

            output = model(torch.FloatTensor(line))

            if not first_run:
                if float(output.item()) > largest_val:
                    largest_val = float(output.item())
                if float(output.item()) < smallest_val:
                    smallest_val = float(output.item())
            else:
                largest_val = float(output.item())
                smallest_val = float(output.item())

            optimizer.zero_grad()
            loss_func = nn.MSELoss()
            loss = loss_func(output, target)
            loss.backward()
            optimizer.step()

            sum_correct += abs(target - output)
            num_samples += 1
            line_num += 1
            first_run = False
        f.close()

        training_error_y_vals.append(float(sum_correct)/num_samples)
        x_vals_training.append(curr_epoch)
        x_vals_variation.append(curr_epoch)
        variation_y_vals_high.append(largest_val)
        variation_y_vals_low.append(smallest_val)


        #Overfitting prevention here
        divergence_threshold = .001 #if (new average error) - (prev
average error) > divergence_threshold
        convergence_threshold = .00001 #if abs(new average error -
prev average error) < convergence_threshold
        f = open("testingnew.txt", "r")
        sum_correct = 0
        num_samples = 0
        first_run = True
        variation = 0
        prev_output = 0
        for line in f:
            line = line.split(",")
            line = list(map(float, line))
            target = [line[-1]]
            target = torch.FloatTensor(target)
            line = line[:-1]
            output = model(torch.FloatTensor(line))
            if not first_run:
                variation += abs((float(output.item()) -
prev_output)/prev_output)
            prev_output = float(output.item())
            sum_correct += abs(target - output)
            num_samples += 1
            first_run = False

        f.close()
        exit_code = -1
        #checking for convergence and divergence
        if testing_differences == -1: #if it is first iteration
            testing_differences = float(sum_correct)/num_samples
        elif float(sum_correct)/num_samples <= testing_differences:
            divergence_counter = 0
            if abs(testing_differences -
float(sum_correct)/num_samples) < convergence_threshold:
                convergence_counter += 1
                if convergence_counter == 10:
                    exit_code = 0
                    break
            else:
                convergence_counter = 0
                testing_differences = float(sum_correct)/num_samples
        else:
            if float(sum_correct)/num_samples - testing_differences >
divergence_threshold:
                divergence_counter += 1
                if divergence_counter == 5:
                    exit_code = 1
                    break
            else:
                divergence_counter = 0

        curr_epoch += 1
        line_num += 1

        testing_error_y_vals.append(float(sum_correct)/num_samples)
        x_vals_testing.append(curr_epoch)



        ylim_multiplier = 2
        if(max(x_vals_testing) > 39):
            plt.xlim(1, max(x_vals_testing) + curr_epoch*.75)
        if(float(sum_correct)/num_samples >= .8):
            plt.ylim(0,float(sum_correct)/num_samples *
ylim_multiplier)
        elif(float(sum_correct)/num_samples < .8):
            plt.ylim(float(sum_correct)/num_samples *
float(sum_correct)/num_samples,.5)
        plt.plot(x_vals_testing, testing_error_y_vals)
        ylim_multiplier -= .01
        plt.pause(.01)


    if exit_code == -1:
        print("Training stopped after " + str(curr_epoch) + " epochs
from reaching max epochs.")
    elif exit_code == 0:
        print("Training stopped after " + str(curr_epoch) + " epochs
from achieving convergence.")
    elif exit_code == 1:
        print("Training stopped after " + str(curr_epoch) + " epochs
from overfitting prevention.")

    if curr_epoch > 99:
        num_of_counted_runs += 1
    #Testing
    f = open("binarytesting.txt", "r")
    sum_correct = 0
    num_samples = 0
    for line in f:
        line = line.split(",")
        line = list(map(float, line))
        target = [line[-1]]
        target = torch.FloatTensor(target)
        line = line[:-1]
        #line = [float(i)/max(line) for i in line] #this does
normalization
        output = model(torch.FloatTensor(line))
        #print(output)
        if float(output) > .1:
            if float(target) == 1:
                sum_correct += 1
            num_samples += 1

    print("Testing result: ", end="")
    if num_samples > 0:
        print(float(sum_correct)/num_samples)
        if curr_epoch > 99:
            sum_of_runs += float(sum_correct)/num_samples
    else:
        print("0.0")
        if curr_epoch > 99:
            sum_of_runs += 0.0

    #Look at this graph

    #plt.show()

    plt.close()
    plt.plot(x_vals_testing, testing_error_y_vals)
    plt.savefig("testing_differences_by_epoch.png")
    plt.close()
    plt.plot(x_vals_training, training_error_y_vals)
    plt.savefig("training_differences_by_epoch.png")
    plt.close()
    plt.plot(x_vals_variation, variation_y_vals_low)
    plt.plot(x_vals_variation, variation_y_vals_high)
    plt.savefig("variation-be-like.png")
    plt.close()

    f.close()


if num_of_counted_runs == 0:
    avg_of_runs = 0
    print("10 run average: " + str(avg_of_runs))
else:
    avg_of_runs = sum_of_runs/num_of_counted_runs
    print("10 run average: " + str(avg_of_runs))
```

Appendix B

Responsibilites:

Data mining, data preprocessing and testing set creation - Liam, Giacomo

Network architecture implementation - Liam, Dalton

Experimentation and iterative development - Liam, Dalton

Slides - All

Report - All