MATH 6373 Spring 2021 MSDS
Homework 1: Neural Networks MLP

Jacob Gogan
Sara Nafaryeh
Thomas Su

All authors played an equal part

Table of Contents

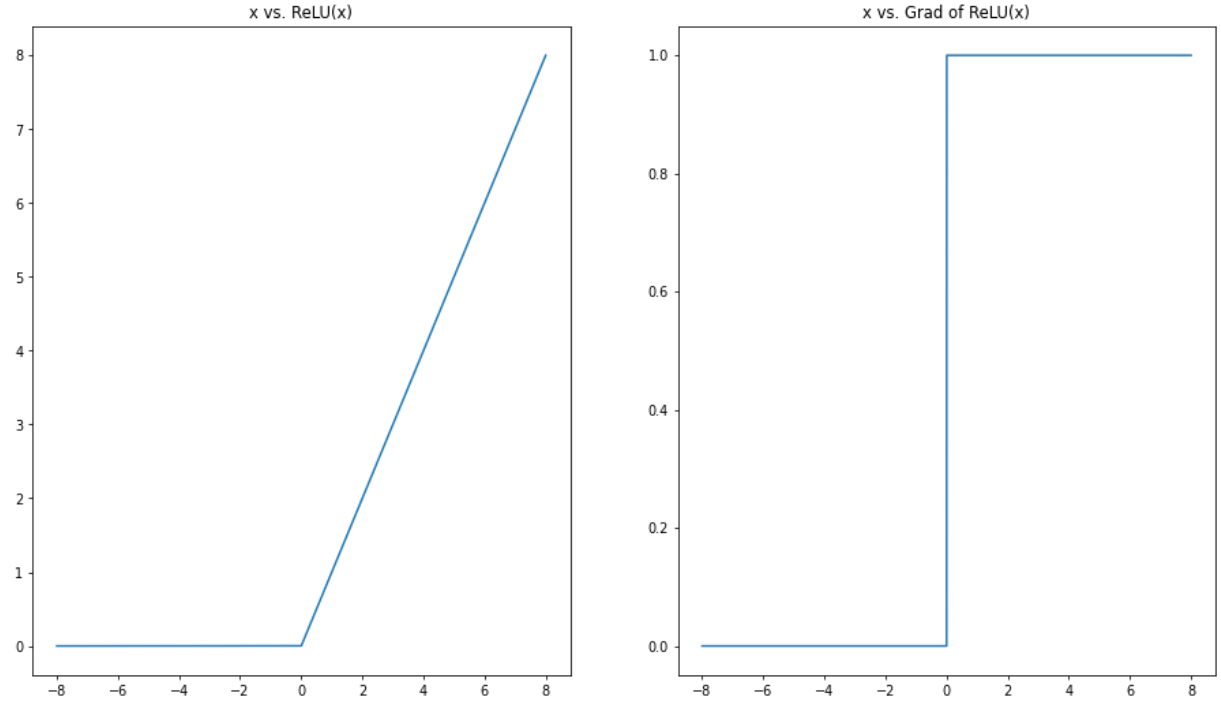
Introduction

Multilayer Perceptron (MLP) is a type of feedforward Artificial Neural Network Model that introduces one or more hidden layers on the basis of a single-layer neural network. The hidden layer is located between the input layer and the output layer. Therefore, the neurons in the hidden layer are fully connected to each input in the input layer, and the neurons in the output layer and each neuron in the hidden layer are also fully connected. Also, the input layer doesn't involve any calculation.

The output of each hidden layer is transformed by an activation function, such as ReLU (rectified linear unit) function, sigmoid function, and hyperbolic tangent (tanh) function. In this report, we are going to apply only the ReLU $f(x) = max(0, x)$ as an activation function that keeps only positive values and sets all the negative values to zero. Without an activation function, The output layer of MLP is equivalent to one of a single-layer neural network, because the fully connected layer performs the affine transformation (that is any transformation that preserves collinearity), and the superposition of multiple affine transformations is still the same as an affine transformation. The solution is to introduce the nonlinear transformation, which is the activation function.

In this report, we will be exploring stock's closing price using MLP. The files were all downloaded from the following link: *finance.yahoo.com*. The stocks that we chose were: **SPY ETF, Apple, Microsoft, Amazon.** According to YahooFinance as of February 3, 2021, Apple, Microsoft, and Amazon are the top 3 holdings of SPY; therefore, our goal for this report is to predict the future SPY closing price for the next day using these three.

```
import numpy as np
x = np.linspace(start = -8, stop = 8, num = 10**5)
y = np.maximum(0, x) # ReLU(x)
dx = x[1]-x[0]
dydx = np.gradient(y, dx)
import matplotlib.pyplot as plt
fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (16, 9), dpi = 70)
ax1.plot(x, y)
ax1.set_title('x vs. ReLU(x)')
ax2.plot(x, dydx)
ax2.set_title('x vs. Grad of ReLU(x)')
```



- Obviously, the ReLU function is a Piecewise Linear Function.
- Its derivative f'(x) = 0 at x < 0, and f'(x) = 1 at x > 0. Although the derivative is undefined when input x = 0, we still take zero as the derivative of the ReLU function at x = 0 here.

STEP 0: DATA Set up

The following steps are taken in python to clean up and organize our data. Once downloaded, each stock came with the following columns:

{Date, Open, High, Low, Close, Adj_Close, and Volume}

Since we are only interested in the **Date** and **Closing price**, we extract those columns and ignore the rest. The first steps to clean up the data we extract and organize the year, month, and day from the DateTime column, then concatenate and combine multiple data frames. We do this by replacing the calendar date values by a sequence t = 1,2,3,4 of indices since the stock market is closed 2 days per week and on certain holidays. On day "t" we denote ***V(t)*** = vector of the prices from the 4 stocks we selected: **[SPY(t), AAPL(t), MSFT(t), AMZN(t)].** Set day "t" input vector **X(t) = [V(t), V(t-1), V(t-2), V(t-3)]** we form a line vector of 16 dimensions (16 features per case).

Our goal in this report is to train an MLP to predict, on the day "t", the next day SPY price **SPY(t+1)**. Therefore, the true value **Y(t) = SPY(t+1)** is unknown on day "t" and we plan to predict Y(t) by using Multi-Layer Perceptron: Z_t = MLP(X(t)) of our data set = {X(1), X(2)....X(N)} , where N = total number of cases (stock dates). After cleaning the data frame and removing any missing values, we set N = 1005 cases from the dates Year/Month/Day: 2017/2/6 to 2021/2/2.

We will attach the .csv files along with this report, but for a visual understanding, below we display the first 5 dates of our data frame with our true value **Y(t) = SPY(t+1)** followed by the first 8 out of the 16 total input vector **X(t)** stocks.

			SPY(t+1)	SPY(t)	AAPL(t)	MSFT(t)	AMZN(t)	SPY(t-1)	AAPL(t-1)	MSFT(t-1)	AMZN(t-1)
Year	Month	Day									
2017	**2**	**6**	228.94	228.93	32.57	63.64	807.64	229.34	32.27	63.68	810.20
		7	229.24	228.94	32.88	63.43	812.50	228.93	32.57	63.64	807.64
		8	230.60	229.24	33.01	63.34	819.71	228.94	32.88	63.43	812.50
		9	231.51	230.60	33.10	64.06	821.36	229.24	33.01	63.34	819.71
		10	232.77	231.51	33.03	64.00	827.46	230.60	33.10	64.06	821.36

Along with the correlation matrix of our 16 features:

```
print(df.iloc[:,:].corr())
```

```
           SPY(t+1)    SPY(t)   AAPL(t)  ...  AAPL(t-3)  MSFT(t-3)  AMZN(t-3)
SPY(t+1)   1.000000  0.994855  0.910212  ...   0.905795   0.910136   0.896517
SPY(t)     0.994855  1.000000  0.911714  ...   0.906664   0.911402   0.896874
AAPL(t)    0.910212  0.911714  1.000000  ...   0.995937   0.939131   0.931693
MSFT(t)    0.913735  0.915537  0.941027  ...   0.937175   0.996727   0.953803
AMZN(t)    0.897965  0.898848  0.933809  ...   0.930049   0.955835   0.995385
SPY(t-1)   0.991756  0.994831  0.909211  ...   0.907685   0.912435   0.897234
AAPL(t-1)  0.909015  0.909538  0.998453  ...   0.997249   0.939499   0.932161
MSFT(t-1)  0.912998  0.913410  0.940037  ...   0.938159   0.997703   0.955015
AMZN(t-1)  0.897624  0.897673  0.932829  ...   0.930942   0.956372   0.996890
SPY(t-2)   0.986986  0.991730  0.907729  ...   0.908238   0.912812   0.897232
AAPL(t-2)  0.907331  0.908342  0.997262  ...   0.998436   0.939623   0.932517
MSFT(t-2)  0.911642  0.912668  0.939654  ...   0.938935   0.998375   0.956016
AMZN(t-2)  0.896991  0.897349  0.932240  ...   0.931874   0.956730   0.998335
SPY(t-3)   0.982190  0.986982  0.905649  ...   0.909864   0.914601   0.898059
AAPL(t-3)  0.905795  0.906664  0.995937  ...   1.000000   0.940350   0.933193
MSFT(t-3)  0.910136  0.911402  0.939131  ...   0.940350   1.000000   0.957497
AMZN(t-3)  0.896517  0.896874  0.931693  ...   0.933193   0.957497   1.000000

[17 rows x 17 columns]
```

Right away we see that our correlation matrix has high correlation (strong relationship) between its features with values ranging from 0.80 to 0.99. This can cause

problems with the way the model handles the data. Such a problem can be removing a feature, it can have a large impact on the prediction. Another problem could be that it can make the learning algorithm slower. We can also see that the correlations between the target stock SPY(t+1) shows an extremely high/ strong relationship.

STEP 1: Multi-Layer Perceptron (MLP): 3 layers Architecture

In this section, we will briefly go over what it means for MLP to be an artificial neural network. They are composed of input, hidden, and output layers. In our report, the input layer represents the 16 feature dimensions of the stocks and the output layer is of dimension 1, also known as the closing price we are trying to determine. The hidden layer is dimension H = h, at the moment we do not know what value integer h; however, further down in the report we try various values to find the best results for our data. Each one of these layers has perceptrons that are sending multiple signals to the next layer until it reaches the output. For each signal, the perceptron uses different weights of arrows, w(j,k), where input neuron k → goes to hidden neuron j in dimension H. In our report, we have 16 neurons from the input layer. They are densely connected by 1 connection to each neuron to the first hidden layer, h. Therefore, we define this as W = h * 16 matrix of weights w(j,k) with B = column vector [bj] of all the thresholds for the hidden neuron j to H. From the hidden neuron j to the output neuron 1, we define M = column matrix of weights m(1,j) with its C = single threshold for single output neuron. It is important to note that between each layer, we will be using the activation function “ReLU”, where f(x) = max(0,x). This function will help improve the neural network by speeding up the training process.

In conclusion, with 16 input layers, h hidden layers, and bj thresholds, we expect the total number of MLP parameters to be 16h+h+h+1 = 18h+1. Once we identify our best-hidden layer h* we can observe our model’s best performing MLP* and record our data-sets W, B, M, and C.

To get to that point, we need to extract the true value set from our data, also known as the closing price we hope to predict **Y(t) = SPY(t+1)**. We define this data frame as **Y**, a vector column with 1005 cases. Also, we create a new data frame without the true value set and define this as **X**, a matrix of 1005 cases and 16 features.

```
# extract the true set
Y = df.loc[:,'SPY(t+1)']
Y # true_set
```

```
Year  Month  Day
2017  2      6      228.940002
             7      229.240005
             8      230.600006
             9      231.509995
             10     232.770004
                       ...
2021  1      27     377.630005
             28     370.070007
             29     376.230011
      2      1      381.549988
             2      381.850006
Name: SPY(t+1), Length: 1005, dtype: float64
```

```
# create a new df as X that is removed the column of true set
X = df.drop(['SPY(t+1)'], axis = 1)
X
print(X)
```

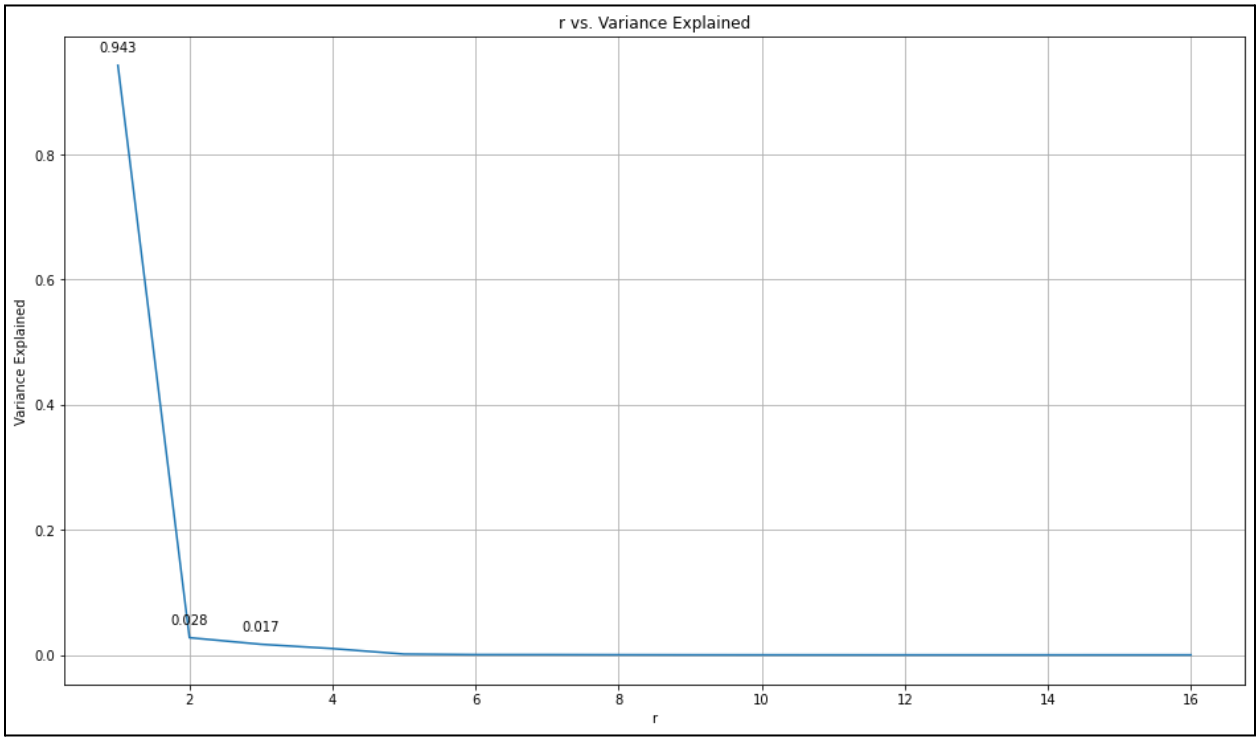
```
                    SPY(t)     AAPL(t)  ...   MSFT(t-3)    AMZN(t-3)
Year Month Day                          ...
2017 2     6    228.929993   32.572498  ...   63.580002   832.349976
           7    228.940002   32.882500  ...   63.169998   839.950012
           8    229.240005   33.009998  ...   63.680000   810.200012
           9    230.600006   33.105000  ...   63.639999   807.640015
           10   231.509995   33.029999  ...   63.430000   812.500000
...                    ...         ...  ...         ...          ...
2021 1     27   374.410004  142.059998  ...  225.949997  3292.229980
           28   377.630005  137.089996  ...  229.529999  3294.000000
           29   370.070007  131.960007  ...  232.330002  3326.129883
     2     1    376.230011  134.139999  ...  232.899994  3232.580078
           2    381.549988  134.990005  ...  238.929993  3237.620117

[1005 rows x 16 columns]
```

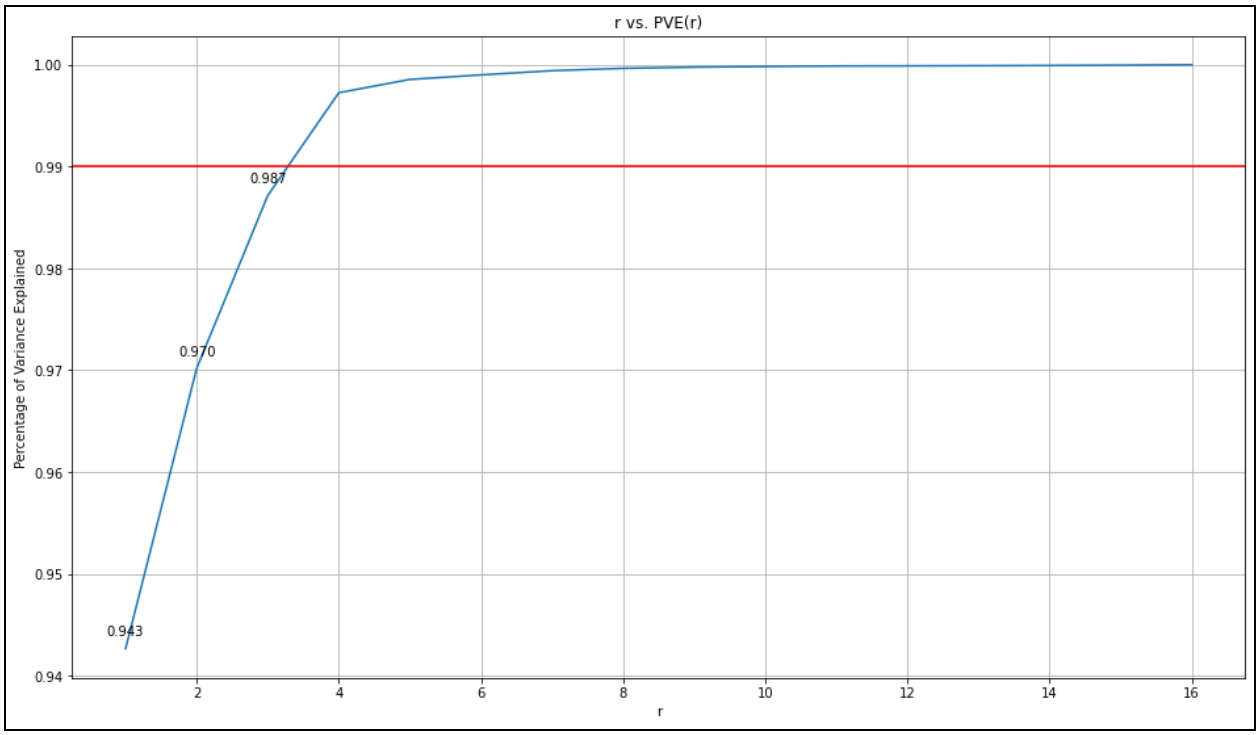
STEP 2: Find the Size Range of Hidden Layers

In this section, we are going to select four values for h = size of hidden layers. We will use PCA, also known as the Principal component, to find the lowest value our hidden layer can be and define it as h95. Next, we will find the upper bound hidden layer, hmax, through the equation 18h + 1 ≤ size(TRAIN data set); h2 and h3 hidden layers will have values between the lower and upper hidden layer.

Thus our goal is **h95 < h2 < h3 < hmax**

To find h95 we take our (1005) * (16) INPUT data frame **X** which we created above, we center and rescale it, and define it as **SX**. After applying PCA to SX, we get the following graphs. R vs. Variance Explained and r vs. PVE(r)





Using the python code ***round(np.interp(0.99, ys,xs),3)*** we get r = 3.281, so we round and select number of principal components for 95% explained variance to be **h95 = 3**.

Moving on to finding the hmax, we take an 80/20, train/test split of our 1005 data cases to have a training = 804 cases and test = 201 cases. To find the number of parameters we take the formula: 18h+1≤ size(TRAIN) and get **hmax = 44**.

For our h2 and h3 we select equally spanned range values between h95 and hmax:

```
# hmin = number of principal components for proportion of explained variance
# dim_input * h + h + h + 1 = number of MLP parameters <= number of cases of training set
hmin = len(X_pca.columns)
hmax = int((X_train.shape[0] - 1)/(X_train.shape[1] + 1 + 1))
# Select 4 values for h as a size of hidden layer
from random import sample
h = sample(list(range(hmin + 1, hmax + 1)), k = 2)

# hyperparameters
n_input = len(X.columns) # dimension of input layer
# dimension of hidden layer
n_h1 = hmin
n_h2 = 14
n_h3 = 28
n_h4 = hmax
#n_hidden = sample([n_h1, n_h2, n_h3, n_h4], k = 1)[0]
n_output = 1 # dimension of output layer
```

```
[n_h1, n_h2, n_h3, n_h4]
```

```
[3, 14, 28, 44]
```

Therefore, we have the following MLP 3 layers Architecture:

input defined as n_input = 16, **output** defined as n_output = 1, and hidden layers

h95 we define as n_h1 = 3

h2 defined as n_h2 = 14

h3 defined as n_h3 = 28

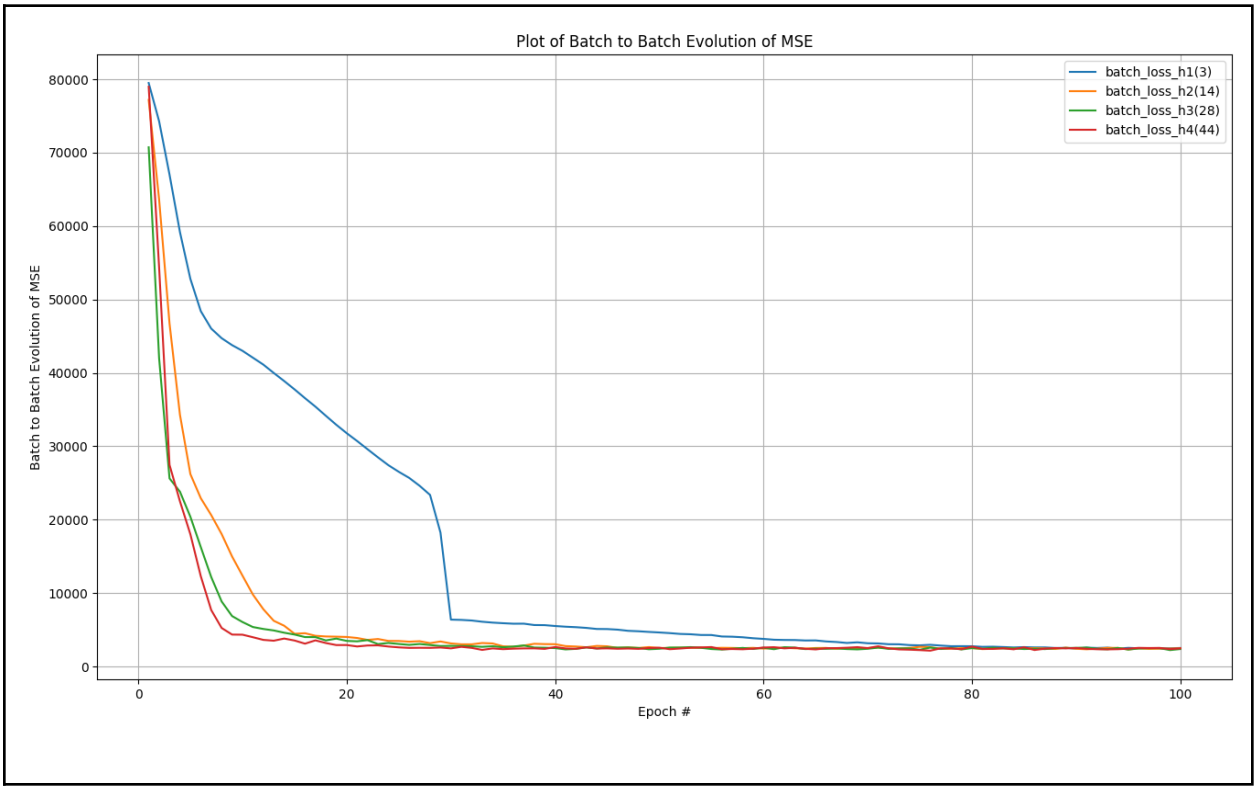
hmax defined as n_h4 = 44

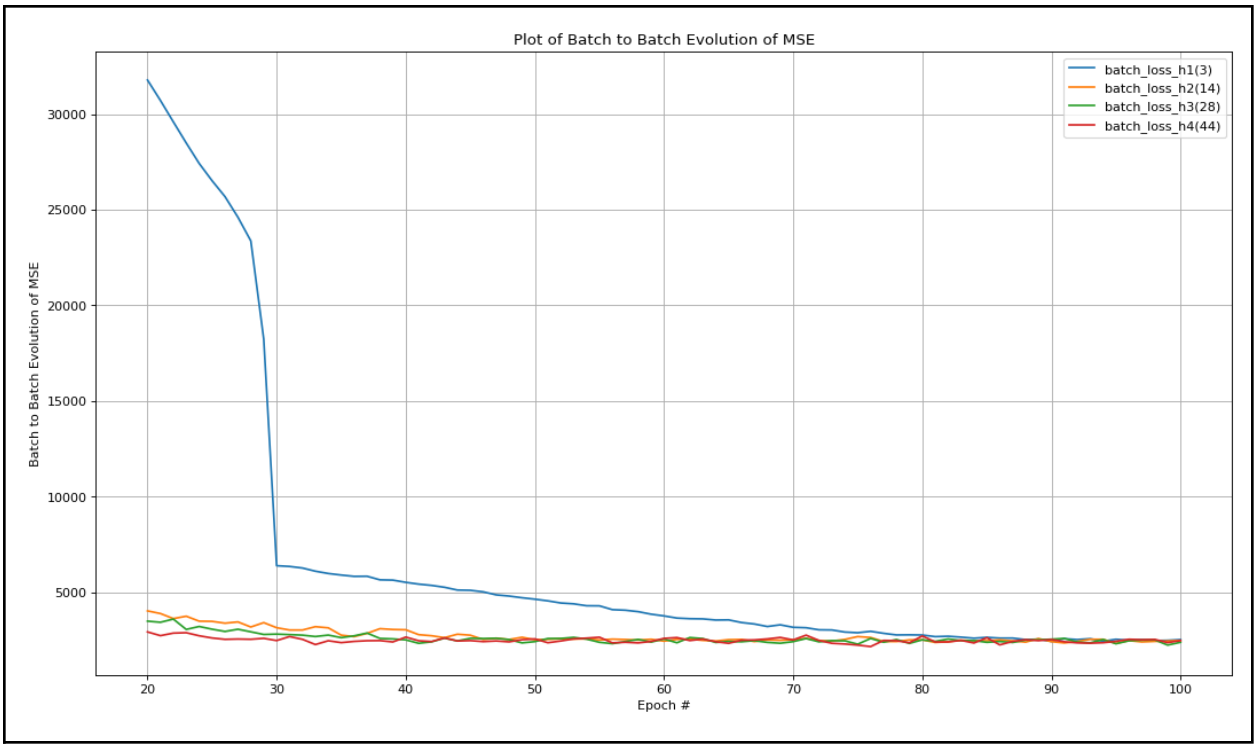
STEP 3: Automatic Training Via Tensorflow

For each of these hidden layer-values, we implement automatic learning via Tensorflow. Using a batch size of 25 cases and an epoch of 100, we observe the following ADAM gradient descent, BATCH learning, and MSE loss function. First, we have to compile the model because we want to specify parameters for the training. Defining the optimizer = optimizer.Adam(learning_rate = 0.01). The default of the learning rate is set to 0.001, we want to fasten the training procedure and change it to 0.01. Also, we compile loss_fn = losses.MeanSquareError(). To save the best model during the whole training process, we make sure to use the function, callbacks.ModelCheckpoint(). This allows us to keep the model of least steps and save the weights at a frequency to be used later in our report to continue the train or achieve the best performance. Naming the file path as 'BestModel.h5' we can easily go back to our best model, and set monitor = 'val_loss' in order to monitor our model's total loss.

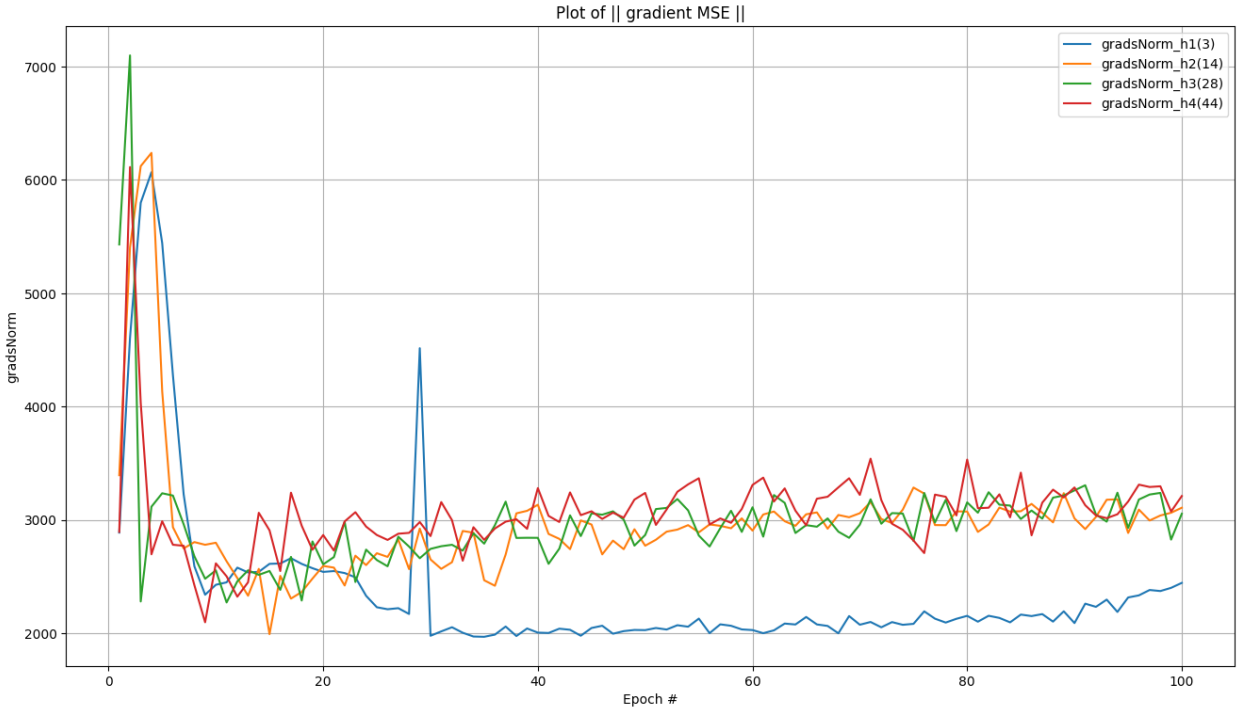
The batch size is a hyperparameter that we must test and tune based on how our specific model is performing during training. By grouping the data in batches, the quicker our model will complete each epoch during training. This way, we are able to process much more than one single sample at a time. As stated before, we are using a batch size of 25, and with our training set of 804 cases, we have about 32 iterations (804/25) to complete 1 epoch. Epochs are one forward pass and one backward pass of all the training sets in our data. Passing the complete data set through a neural network only once is not sufficient; therefore, we need to pass this full data set multiple times to the same neural network to cause one epoch list to underfit. This is why we have set our epoch to 100. The following is the plot batch to batch evolution of the Epoch # vs.

MSE for each of the four h values. Here we are calculating the loss function at the end of each epoch during training. The value for the loss will constantly be changing since the weights of our model are constantly being updated. The objective is to minimize the loss function and see our loss continuously decrease as we run more epochs and we can see that in the following graph. Overall the batch loss of all the hidden layers is decreasing after each epoch. We observe a sharp decrease followed by a horizontal trend for batch_loss of h2, h3, h4 hidden layers at around 10 epochs. However, h1 does not show signs of a horizontal trend until after about 40 epochs.





Next, we will take a look at the norm of the gradient descent MSE. Since gradients measure the change in all weights with regards to the change in error, we can observe how much our model learns after each epoch. Overall the gradient descent shows steep decreasing slopes for the “gradsNorms” of all hidden layers and comes to a steady converged ||gradient MSE|| value between 3000 and 3500 after 50 epochs. This indicates that the model has a fairly good learning rate. However, we can see once again that the h1 hidden layer does not follow the other layers until after a few epochs. It also appears to have a few steep increasing and decreasing slopes, indicating that a hidden layer of 1 might not be as effective.

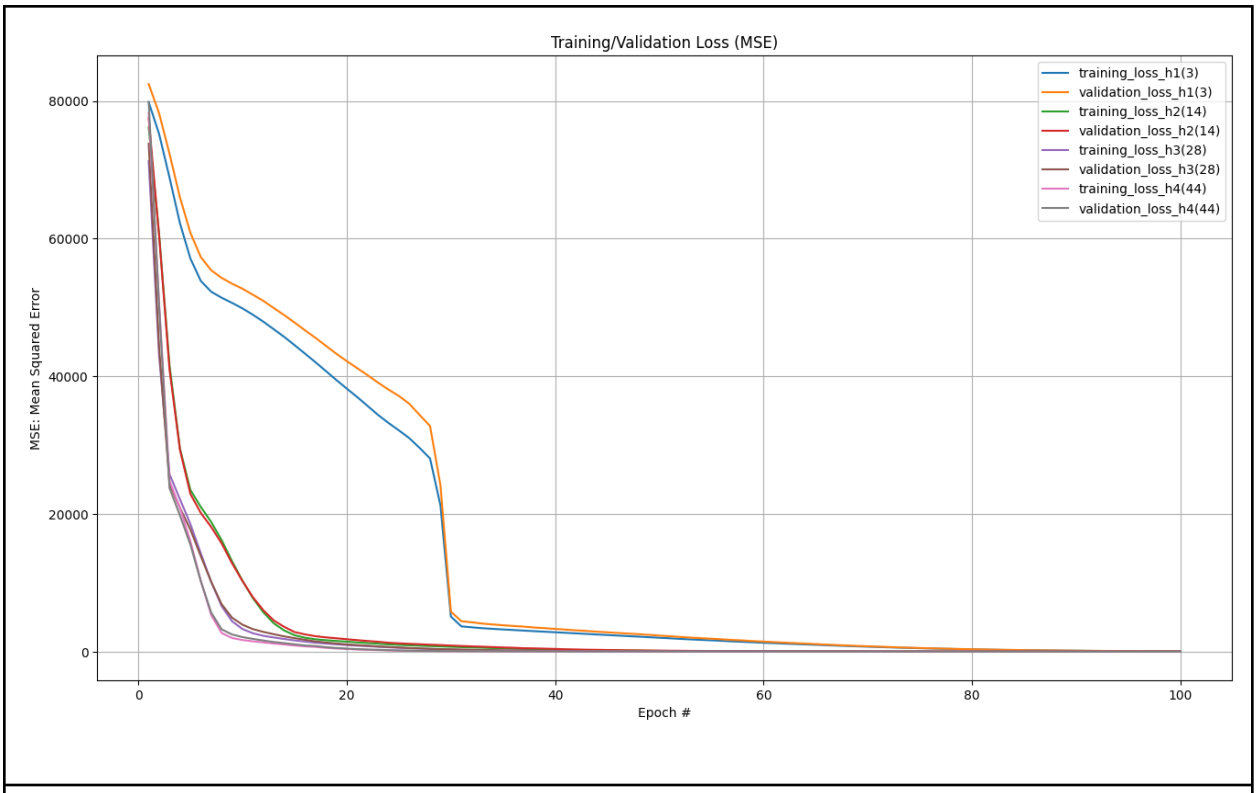

Plot of || gradient MSE ||
gradsNorm_h1(3)
gradsNorm_h2(14)
gradsNorm_h3(28)
gradsNorm_h4(44)
gradsNorm
Epoch #
7000
6000
5000
4000
3000
2000
0
20
40
60
80
100

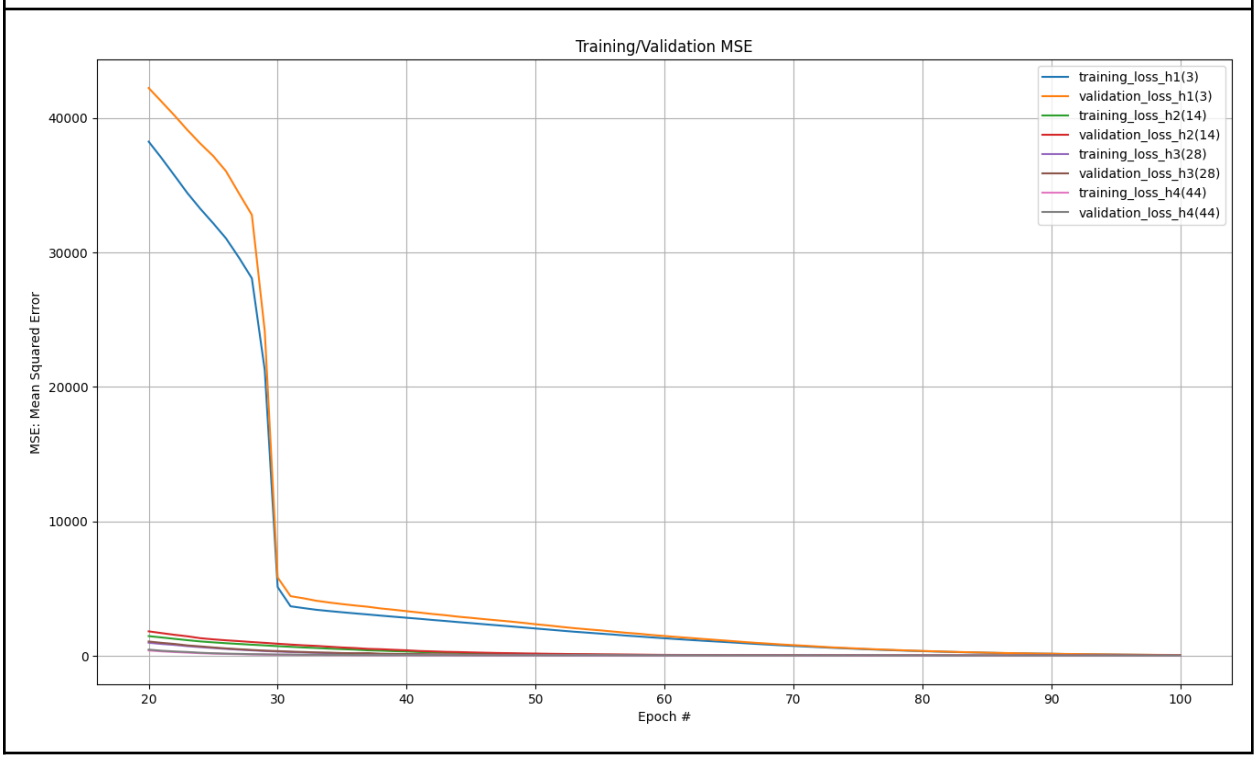
After each epoch, we computed the trainMSE (real loss) on the full training set and testMSE (val_loss) on the full test set. The table below shows the train_loss and val_loss for each hidden layer. Since the training loss is the average of the losses over each batch of training data, the model is changing over time. This results in the loss over the first batches of an epoch are generally higher than over the last batches. The val_loss for an epoch is computed using the model as it is at the end of the epoch, resulting in a lower validation loss. This can be observed in the highlighted part of our table; for each h(n).

	train_loss_h1	val_loss_h1	train_loss_h2	val_loss_h2	train_loss_h3	val_loss_h3	train_loss_h4	val_loss_h4
0	79839.953125	82470.898438	76180.773438	77513.226562	71259.054688	73800.609375	78255.820312	79867.000000
1	75209.648438	78210.406250	60650.230469	60443.296875	43012.570312	44207.273438	50670.886719	50193.050781
2	68877.250000	72327.593750	41631.414062	40968.773438	25785.892578	24371.507812	24819.320312	23771.072266
3	62249.421875	66003.976562	29532.296875	29334.335938	22156.988281	20897.800781	20885.484375	19801.443359
4	57121.011719	60868.050781	23510.054688	22924.714844	18559.527344	17748.361328	16065.084961	15567.115234
...	...	...	...	...	...	...	...	...
95	84.179642	91.777084	17.127085	20.361814	14.175061	16.506943	15.950694	18.435843
96	76.399483	84.250862	17.662964	19.472635	14.425071	17.076702	15.582515	16.865574
97	68.175102	76.787910	16.907173	18.814045	14.540498	16.889261	15.629103	18.367348
98	62.046375	70.295425	16.650818	18.642294	25.823471	27.415977	16.853920	17.540077
99	57.367649	66.334831	16.623348	19.060925	15.002465	16.612886	14.213573	16.543114

100 rows × 8 columns

The following plot is trainMSE and testMSE of each h(n) value. The only sign of possible overfitting is for h1, where the training_loss (trainMSE) is separate from the val_loss (testMSE) between about 10 to 30 epochs. Other than that, there is only a small difference between the two losses for each h, as they are moving in the same direction. As we can see in both the table and graphs, we are able to monitor the training to know when to stop. The MSE is useful performance in helping to drive the learning due to gradient descent to decrease our MSE as much as possible and we see that after 30 epochs.

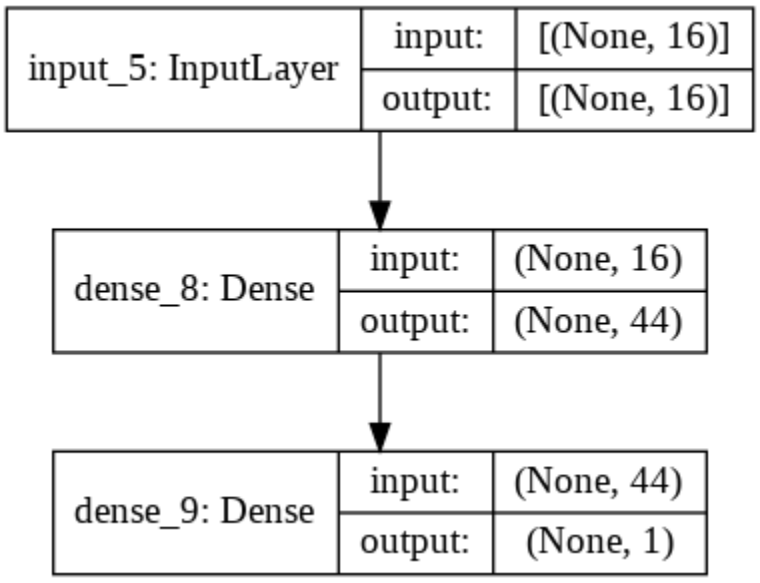

Training/Validation Loss (MSE)
MSE: Mean Squared Error
Epoch #
training_loss_h1(3)
validation_loss_h1(3)
training_loss_h2(14)
validation_loss_h2(14)
training_loss_h3(28)
validation_loss_h3(28)
training_loss_h4(44)
validation_loss_h4(44)


Training/Validation MSE
MSE: Mean Squared Error
Epoch #
training_loss_h1(3)
validation_loss_h1(3)
training_loss_h2(14)
validation_loss_h2(14)
training_loss_h3(28)
validation_loss_h3(28)
training_loss_h4(44)
validation_loss_h4(44)

STEP 4: Pick h* and Perform Best MLP

Given the information we have about our data, we can finally select the size of the hidden layer for the best performing MLP* to be h* = 44, also known as our h3. This hidden layer has shown to have the lowest trainMSE (training_loss) and testMSE (val_loss) out of the 4 hidden layers, as well as showing less fluctuation in normalized gradient MSE. As seen in our Training/Validation Mean Squared Error (MSE) plot, h2, h3, h4 often show mostly the same performances, but h3 performs more steadily.

We repeat STEP 3 using h* = 44. Now the weights and biases are now fixed for this MLP*, we can observe our plot model: input = 16, hidden layer = 44, and output= 1. As well as the model's weights and biases as follow:



The weights from the input layer to the hidden layer defined as:

W = model.layers[0].get_weights()[0]

The column vector of all thresholds for hidden neurons j in the hidden layer:

B = model.layers[0].get_weights()[1]

Column matrix of weights m(1, j) from hidden neuron j to output neuron 1:

M = model.layers[1].get_weights()[0]

And Single threshold for a single output neuron defined as:

C = model.layers[1].get_weights()[1]

And their respective shapes:

```
print(W.shape, B.shape, M.shape, C.shape)
```

```
(16, 44) (44,) (44, 1) (1,)
```

After each epoch, trainMSE/testMSE on the full train/test set we also observe the model loss to be 22.9950 and our Best Model loss to be 11.2118 Our loss has overall decreased and we know we are headed in the right direction.

```
model.evaluate(X_test,Y_test)
```

```
7/7 [==============================] - 0s 2ms/step - loss: 22.9950
22.99501609802246
```

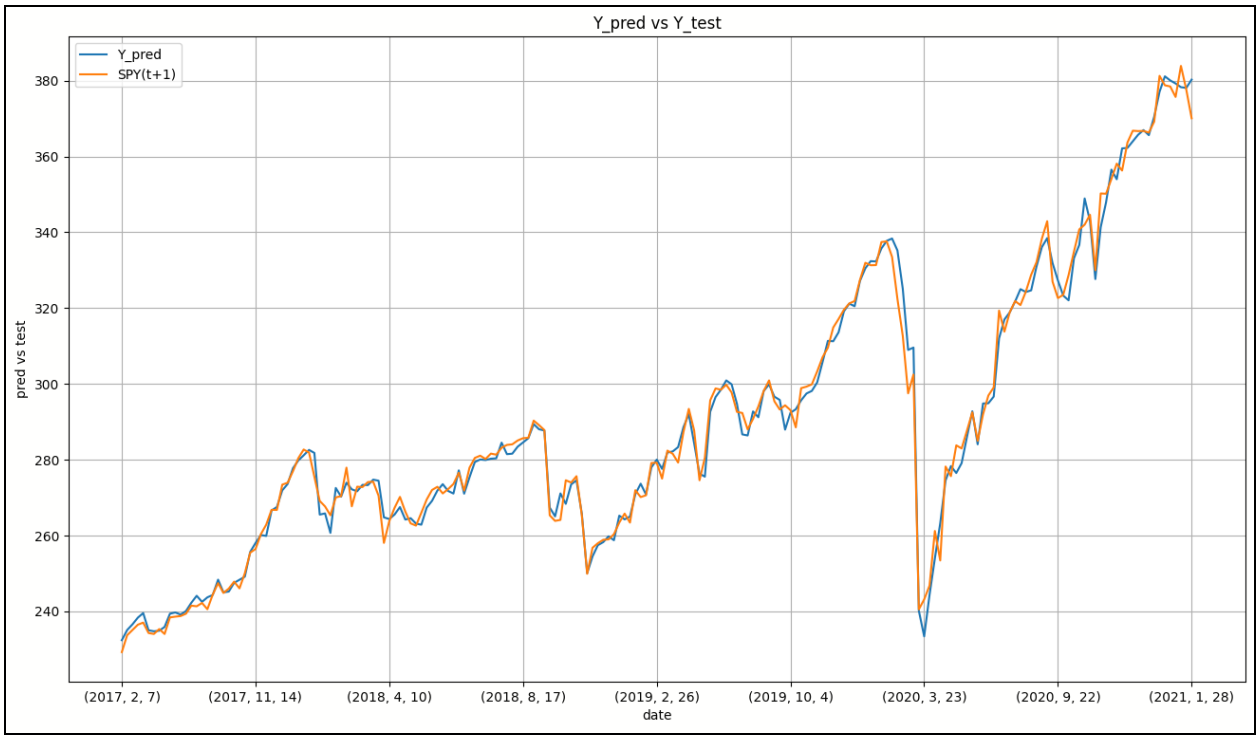
```
# Restore the best model

from tensorflow.keras.models import load_model

best_model = load_model('BestModel.h5')
best_model.evaluate(X_test, Y_test)
```

```
7/7 [==============================] - 0s 2ms/step - loss: 11.2118
11.211752891540527
```

Now that we have watched our training set and have noticed a decrease in the MSE, at a fast speed of 2ms/step, we can now evaluated it at its practical performance ERR(t) = | PredY(t) - TrueY(t) |, the table and graph below demonstrate this performance.

			SPY(t+1)	Y_pred	ERR
Year	Month	Day			
2017	2	7	229.240005	232.378922	3.138917
		13	233.699997	235.067764	1.367767
		16	235.089996	236.579910	1.489914
		27	236.470001	238.332306	1.862305
	3	16	237.029999	239.544418	2.514419
...	...	...	...	...	...
2021	1	13	378.459991	380.049805	1.589814
		14	375.700012	379.252441	3.552429
		19	383.890015	378.273285	5.616730
		27	377.630005	378.084290	0.454285
		28	370.070007	380.270630	10.200623

201 rows × 3 columns



Overall, we see that the predicted value and true value follow the same path minus a few errors such as the dates: [2020.3.23] and [2021.1.28]. These are not large errors and might not have to be too concerned, but to be sure we will also review the accuracy percentage performance. Evaluating the performance, we calculate the RMSE/ average(target stocks) to come out to be 1.15%. We want a performance to be less than 5%; ours was a good result. Therefore we are not too concerned about our error rate and conclude this to be a good prediction model.

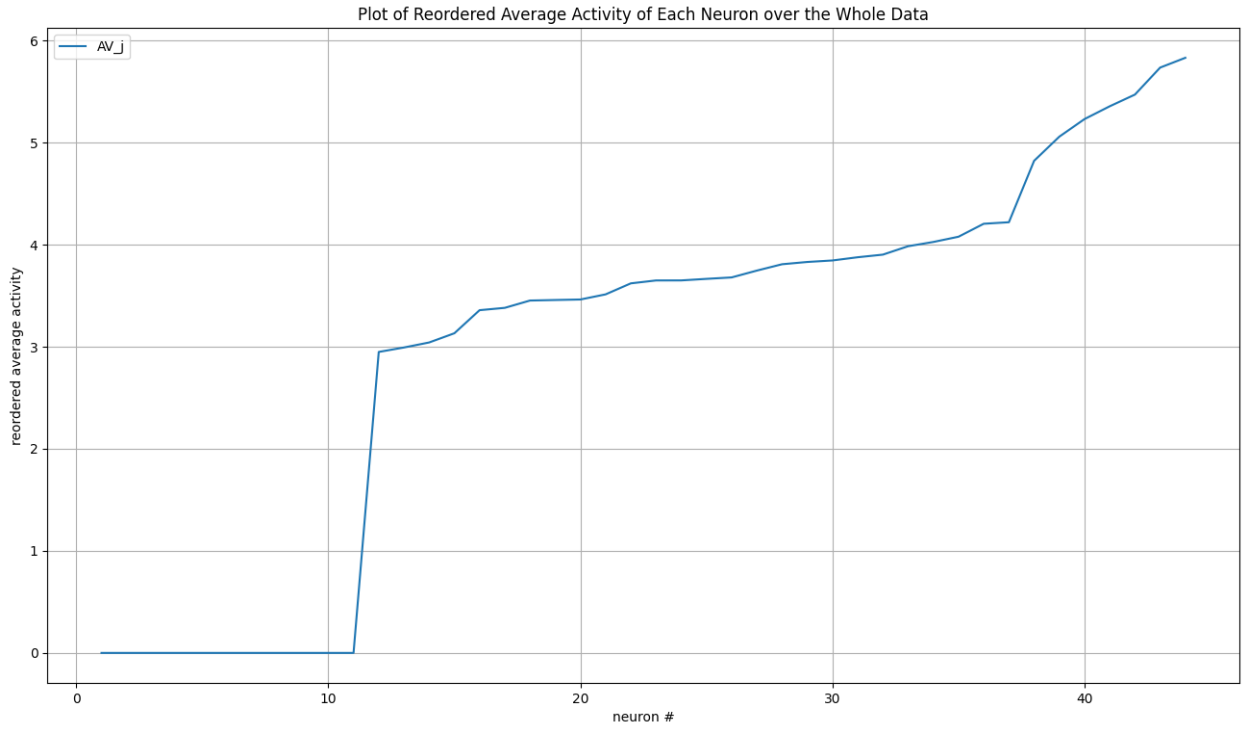
```
RMSE = ( sum( Y_test.subtract( Y_pred.squeeze().sort_index(ascending = True) )**2 )
 / len(Y_pred) )**0.5
print(RMSE, "\n")
Performance = RMSE / np.mean(Y_test)
print(Performance)
```

```
3.348395430033154

0.011566742965840213
```

To access the hidden layer activity, we use the bestModel.layers[0] as a function. We can now compute the average activity of each neuron over the whole data set. For each case Xt presents to the MLP, each neuron NRj computes its state NRJ(t) for that case Xt. Define AVj as the average activity of the neuron NRj, we take the average value of NRj(t) over the whole data set as follows:

AVj = (1/1005) [NRj(1) + NRj(2) + … + NRJ(1005)] since total cases = 1005

With h*= 44 hidden layers, we compute 44 average activities AV1, AV2 … AV28, and reorder the activities AVj for all the hidden Newson's NRj in ascending order. The following is the plot of these recorded activities:



First, we will take a look at the 3 most active hidden **NRj** defined as *most_avg_act*, as well as the most active hidden neuron denoted as **NRa**, defined as *a*=10.

```
most_avg_act = av_j[-3:][::-1] # three most active hidden NR_j
most_avg_act
```

	avg_activity
10	5.829034
25	5.733062
26	5.468699

```
a = most_avg_act.index[0] # denote NR_a as the most active hidden neuron NR_j
a
```

```
10
```

Next we take a look at the list of 16 weights w(1,a) w(2,a) … w(16,a) linking NRa to the input layer already in order by decreasing absolute values. Here we can see that for the most active hidden NRj, the importance of the input feature is AMZN(t-1). The weight of this w(i1, a) is the largest. Therefore, this weight has the most influence on the output. The feature MSFT(t-2) weights is the 2nd most important and so on down the list.

```
# order these "len(X.columns)" weights by decreasing values w(i1, a) > w(i2, a) >
w.sort_values(by = ['weights'], ascending = False, inplace =True)
#w.T
w
```

	weights
AMZN(t-1)	1.605843
MSFT(t-2)	1.554536
MSFT(t-1)	1.409757
AMZN(t-3)	1.368609
AMZN(t-2)	1.366659
MSFT(t-3)	1.363173
AMZN(t)	1.266179
MSFT(t)	1.061402
SPY(t-3)	0.647701
AAPL(t-3)	0.625713
AAPL(t-1)	0.529549
SPY(t-2)	0.503415
AAPL(t-2)	0.471743
AAPL(t)	0.459353
SPY(t-1)	0.456900
SPY(t)	0.250774

Mean value of the real input, Wj is the weight. Taking the product of these two and ranking then, we can identify from the strongest to weakest impact. We can see that for these most active hidden NRj, AMZN(t-1) is the most important input and SPY(t) is the least.

Most active hidden layer compare the 16 numbers | w(LA, j) | x mean[| inp(j) |] for j=1 ... 16 where mean[| inp(j) |] is the average of | inp(j) | over all cases

```
[82] abs(w) * np.mean(abs(X_test))
```

	weights
AMZN(t-1)	1.289218
MSFT(t-2)	1.248028
MSFT(t-1)	1.131795
AMZN(t-3)	1.098760
AMZN(t-2)	1.097194
MSFT(t-3)	1.094396
AMZN(t)	1.016526
MSFT(t)	0.852125
SPY(t-3)	0.519994
AAPL(t-3)	0.502341
AAPL(t-1)	0.425137
SPY(t-2)	0.404156
AAPL(t-2)	0.378729
AAPL(t)	0.368782
SPY(t-1)	0.366813
SPY(t)	0.201329

Now, we do the same for most inactive NRj. we find that the least active hidden NRj is b = 44 at the value of 0, and the weights w(LA,1)...w(LA,10) are small.

```
least_avg_act = av_j[0:3] # three most inactive hidden NR_j
least_avg_act
```

	avg_activity
44	0.0
36	0.0
14	0.0

```
b = least_avg_act.index[0] # denote NR_b as the most inactive hidden neuron NR_j
b
```

```
44
```

The list of 16 weights w(1,b) w(2,b) … w(16,b) links NRj to the input layer and order them by decreasing absolute values. Here we can see that for the most inactive hidden NRj, the importance of the input feature is SPY(t-1). The weight of this w(i1,b) is the largest. Therefore, this weight has the most influence on the output. The feature MSFT(t-3) weights is the 2nd most important and so on down the list.

	weights
SPY(t-1)	0.328734
MSFT(t-3)	0.302766
SPY(t-3)	0.252328
AMZN(t-2)	0.186708
SPY(t)	0.184303
AAPL(t-3)	0.180884
SPY(t-2)	0.146546
AAPL(t-2)	0.136881
MSFT(t-1)	0.124946
MSFT(t-2)	0.113240
MSFT(t)	0.094213
AAPL(t)	0.075053
AMZN(t-3)	0.072929
AMZN(t)	0.066403
AMZN(t-1)	0.049136
AAPL(t-1)	0.029136

Once again we look at the mean values of the real input, where Wj is the weight. Taking the product of these two and ranking then, we can identify from the strongest to weakest impact. We can see that for these most inactive hidden NRj, SPY(t-1) at a weight of 0.2639 is the most important input and AAPL(t-1) at weight of 0.0233 is the least.

least active hidden compare the 16 numbers | w(LA, j) | x mean[| inp(j) |] for j=1 ... 16

where mean[| inp(j) |] is the average of | inp(j) | over all cases

```
[89] abs(w1) * np.mean(abs(X_test))
```

	weights
SPY(t-1)	0.263918
MSFT(t-3)	0.243070
SPY(t-3)	0.202576
AMZN(t-2)	0.149894
SPY(t)	0.147964
AAPL(t-3)	0.145219
SPY(t-2)	0.117652
AAPL(t-2)	0.109892
MSFT(t-1)	0.100311
MSFT(t-2)	0.090913
MSFT(t)	0.075637
AAPL(t)	0.060255
AMZN(t-3)	0.058549
AMZN(t)	0.053310
AMZN(t-1)	0.039448
AAPL(t-1)	0.023391

The following is a specific [stock + time delay] corresponding to the features i1, i2, … i16 from start to end:

```
t1_stop = process_time()
print(f"\nElapsed time (mins): {(t1_stop - t1_start)/60:.6f}")


Elapsed time (mins): 3.593340
```

Conclusion

Pros:

- MLP performs very well on non-linear data-sets.
- Our MLP Model exhibits a small difference between training and validation error.
- Although the computational complexity is proportional to the network complexity, MLP can be remarkably efficient if trained properly.

Cons:

- Typically, MLP is also very easy to overfit on the training data, resulting in poor generalization on the test set. To reduce overfitting, we need to constrain the model complexity by reducing the number of cases of the data-set. On the contrary, our MLP was somewhat underfitting instead, because the model fails to learn sufficiently with only 1005 cases, then the model will perform poorly on the training sets. It's concluded that we should've increased extra cases to optimize the model.

Since there is no straightforward answer to what is the best number of hidden layers to help improve a model's performance, It would be interesting to explore other hidden layer values of h other than the four we looked into h =1, 26, 40, and 44. Also, it would be interesting to see if the addition of more than 1 hidden layer helps with the model's performance in future reports of Multiple Layer Perceptrons.

Reference

- machinelearningmastery.com/introduction-to-regularization-to-reduce-overfitting-and-improve-generalization-error/
- arxiv.org/abs/1611.03530
- builtin.com/data-science/gradient-descent
- medium.com/data-science-bootcamp/multilayer-perceptron-mlp-vs-convolutional-neural-network-in-deep-learning-c890f487a8f1#:~:text=Disadvantage%20is%20that%20the%20number,that%20it%20disregards%20spatial%20information.
- stats.stackexchange.com/questions/383489/is-there-any-relation-between-number-of-hidden-layers-in-a-neural-network-and-pe
- en.wikipedia.org/wiki/Multilayer_perceptron
- en.wikipedia.org/wiki/Rectifier_(neural_networks)