

PS6: Backpropagation Part II

To begin, I found similarities in the problem from PS5 and this problem, so I was able to re-use some code fragments from the last project for this project. Since this project deals with financial data like PS1, I was also able to reuse some scripts and code fragments from that project as well in order to correctly get the financial training and validation data read in for use with the Matlab neural net toolbox. Please note that my runnable code is in 'ps6_financial_backprop.m.'

Training Setup:

First I will provide a brief explanation of some things I did to initially get the feed forward net with backpropagation training successfully.

As mentioned, I reused the code for reading in the training data from the financial training data set provided for PS1. I also reused the code for normalizing the training data, i.e. 'find_minmax_feature_vals.m' and 'scale_all_feature_values.' I made the targets for the neural net be the attributes of the provided training data, i.e. the rates of return for all of the companies. I used a 2-dimensional input with 10 interneurons in the first layer and a single output neuron for the second layer. I initialized the feed forward net with the given number of interneurons and then set up the training data divide function.

For the divide function, I ended up using the 'dividerand' option to get the most general output from the neural net, given that random is better than pre-determined. I set the train ratio on the divide parameter to 50% and the test ratio on the same parameter to 50%, while I let the validation data ratio be 0%. I divided the training data in this way in order to make sure the neural net was not being over-trained, i.e. the fit is good on both training and test data, but also to make sure that the net was not being validated either. I wanted to make sure the net was not being validated because I will manually validate the net using another set of validation data.

Carrying on, I set the initial training function to 'traincgf', the transfer function on the first neuron layer to 'tansig' and the transfer function on the second layer to 'purelin.' The input to the net were the 'feature_scaled_vals' (the transpose of the scaled feature values) and the targets were the 'attributes' (transposed attributes matrix). The reason why the feature values and the attributes matrices had to be transposed is because the Matlab neural net toolbox uses the input format:

X R-by-Q matrix

T U-by-Q matrix

where X is the input to the net and T is the target data. Hence, in order to properly correspond the multi-dimensional input-data to the one-dimensional target data, a transpose was necessary. I was then able to train the net on these values.

Validation Setup:

Once again, I determined the re-using fragments of the code provided in PS1 was the most efficient way of tackling this problem. Therefore, I used the validation data set provided in PS1, read in as 'raw_features' and the return rates as 'val_attributes.' I calculated the index return average, scaled all feature values, and then simulated the neural net I previously trained. Once again, the required that the input be the transpose of 'feature_scaled_vals.' The simulation produces many outputs, but the most important one is the vector of one-dimensional return rate predictions for each pattern called 'simOutputs.' The return rate of investment of each company can be accessed using the pattern index for that company.

Calculate Return on Investment:

In this area I changed up my code from PS1 to make it more logical. Basically, what it does is iterate through all of the patterns and get their equivalent output from the simulation on the neural net, i.e. the simulation returns a one-dimensional vector of predicted return rates on investment. For each of these patterns, it checks if the predicted rate of ROI is over 10% (which is a quite favorable return on investment) and if so, it adds that it is a pattern to choose and stores its actual return rate given the 'val_attributes' data for that pattern. This is my method of selecting favorable stocks given the output of the neural network simulation. It also tracks that the population is growing for the number of patterns chosen to invest in (used for average calculation later). Once this is done for all of the patterns, we have a vector of all the actual validation return rates for companies that we want to select, i.e. predicted rate of over 10% return. From this, I calculated the actual average rate of return on investment by taking the sum of all of the actual chosen pattern validation attributes divided by the population (number of patterns chosen to invest in). Then the program prints if the return on investment is good or bad as compared to the index fund return rate.

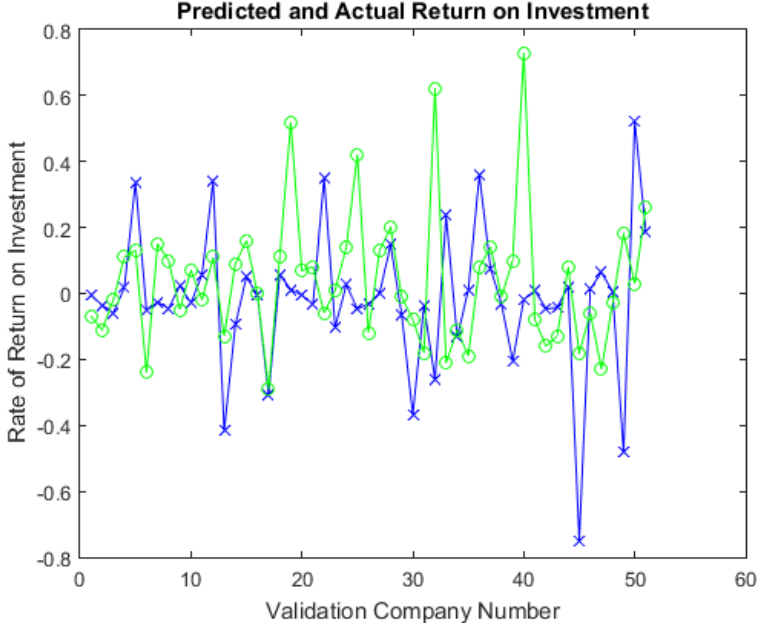
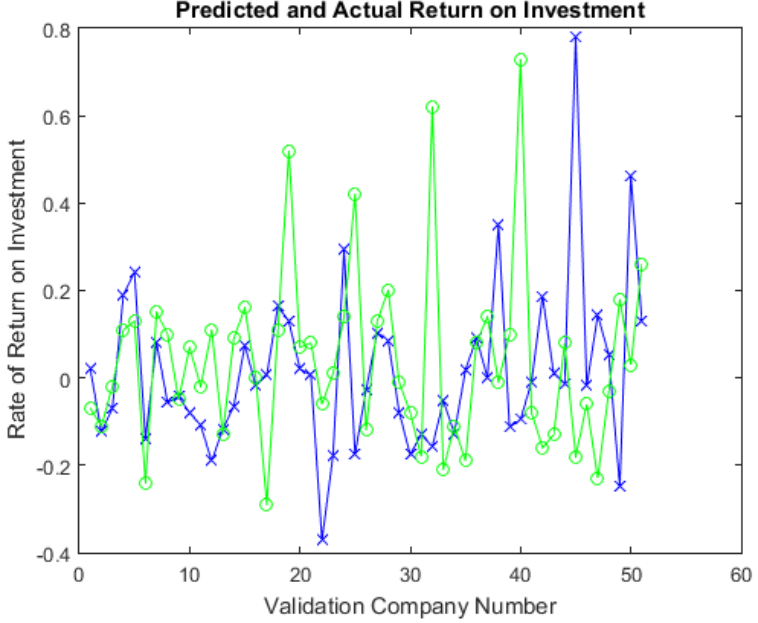
Graphical Comparison:

After the previous steps on calculating return on investment, I plot both the simulation outputs for all 51 validation companies as well as the actual rates of return for those 51 companies. This is in an effort to visual how well the neural network is training given the validation step. The predicted return rates are plotted along a solid blue line with x markers while the actual return rates are plotted along a green line with circle markers. The purpose of these visualization choices are to see how many of the x's for predicted data line up with or compare to the o's for actual data. Hence, this graph shows a comparison between the predicted and actual rates of return on investment for all 51 validation companies.

Experiments:

In my first experiment, I used the 'traincgf' training method with 10 interneurons and the random divide for training/test data. I used the default cap on the number of training iterations, 1000. My average rates of return on investment (ROI) for 4 consecutive runs along with their graphs were the following:

Run #	Avg. Rate of ROI	Graph
1	0.126316	<p>Predicted and Actual Return on Investment</p> <p>Rate of Return on Investment</p> <p>Validation Company Number</p>
2	0.077000	<p>Predicted and Actual Return on Investment</p> <p>Rate of Return on Investment</p> <p>Validation Company Number</p>

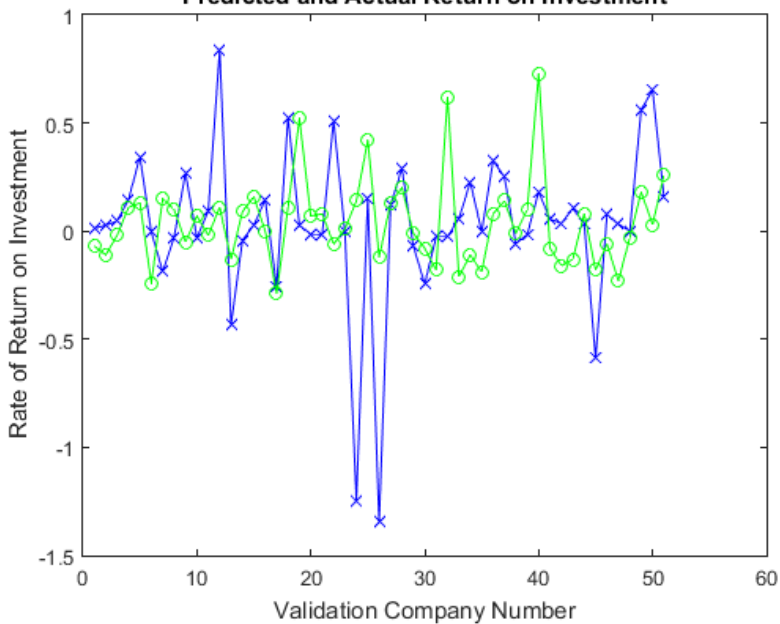
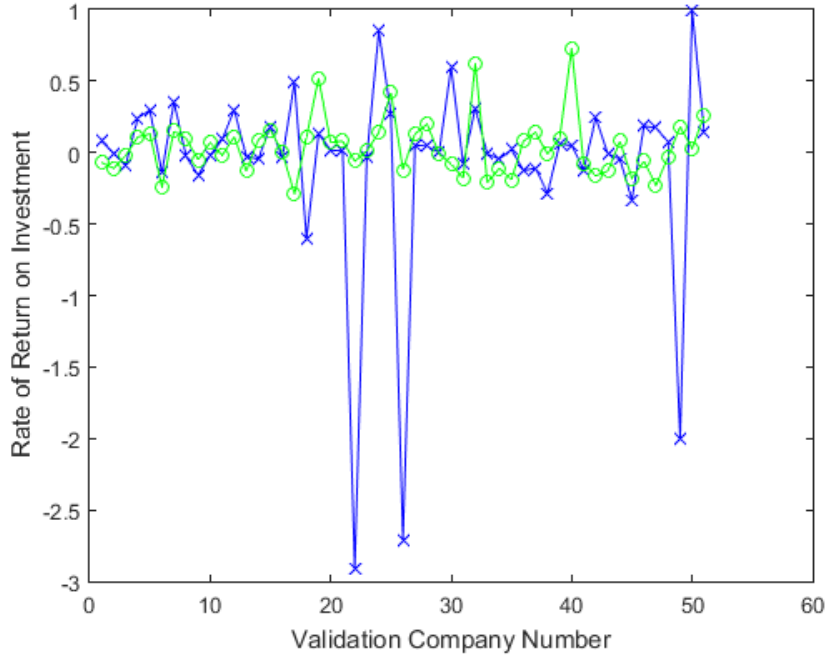
3	0.067500	 <p>The graph displays the Rate of Return on Investment (Y-axis, ranging from -0.8 to 0.8) against the Validation Company Number (X-axis, ranging from 0 to 60). The green line represents the Actual Return, and the blue line represents the Predicted Return. Both lines show significant fluctuations, with the predicted line generally tracking the actual line's movements, indicating a reasonably good model fit.</p>
4	0.070833	 <p>The graph displays the Rate of Return on Investment (Y-axis, ranging from -0.4 to 0.8) against the Validation Company Number (X-axis, ranging from 0 to 60). The green line represents the Actual Return, and the blue line represents the Predicted Return. Both lines show significant fluctuations, with the predicted line generally tracking the actual line's movements, indicating a reasonably good model fit.</p>

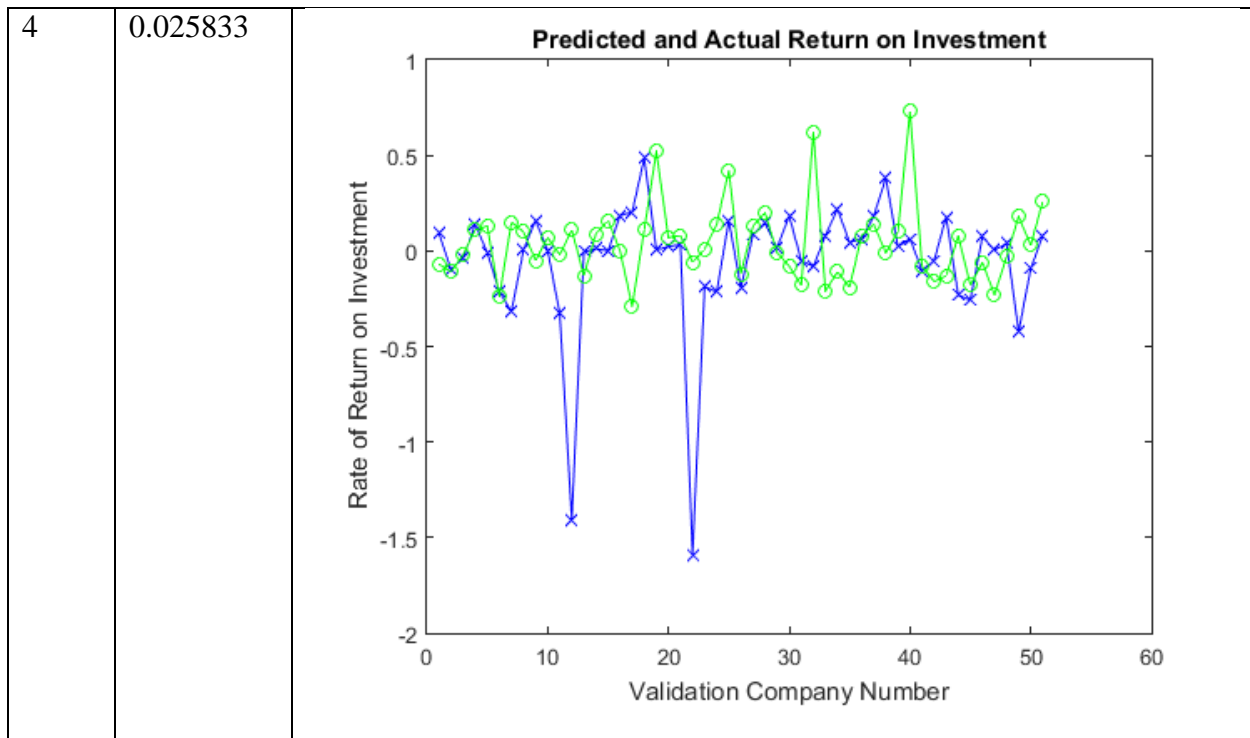
Given that the return index fund rate is ~4%, we can see that somehow all of the 4 predictions were favorable. This shows that 10 interneurons may be good enough for our neural net to be successful at predicting good companies to invest in. This amount of interneurons will be both general and accurate apparently. The training time with 10 interneurons and the given training method was very fast. Each run only took around 6 seconds or less, with the lowest being 2 seconds. I noticed that the accuracy (return rate) was lowest for the fast converging run, but the

results were still favorable. Notice that the correlation between predicted and actual rate of ROI is quite close. This shows that the network is favorably predicting outcomes. We can increase the number interneurons in our next experiment set to see if this gives us more favorable results than we just saw.

In my second experiment, I used the 'traincgf' training method with 100 interneurons and the random divide for training/test data. I used the default cap on the number of training iterations, 1000. My average rates of return on investment (ROI) for 4 consecutive runs along with their graphs were the following:

Run #	Avg. Rate of ROI	Graph
1	0.071667	

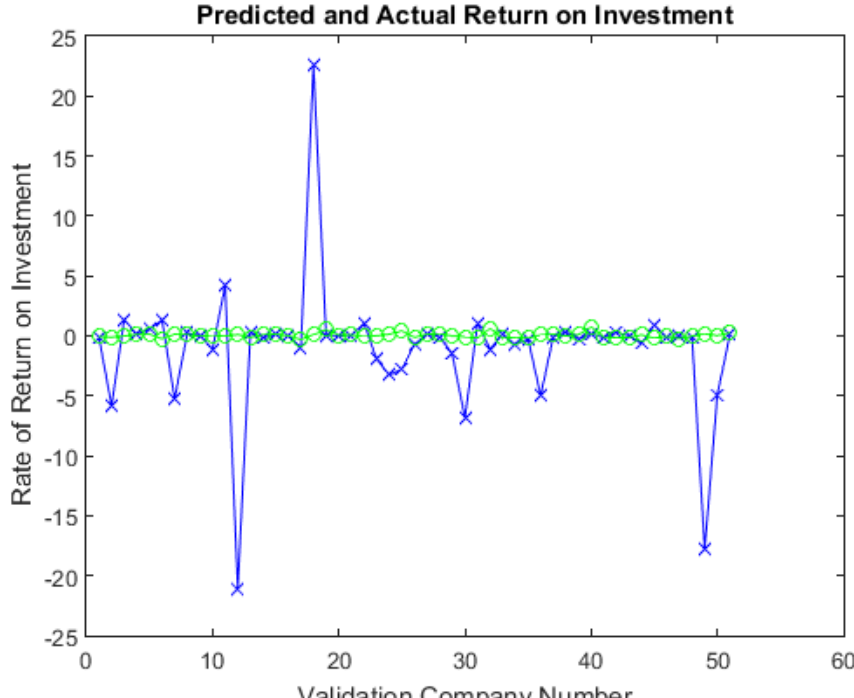
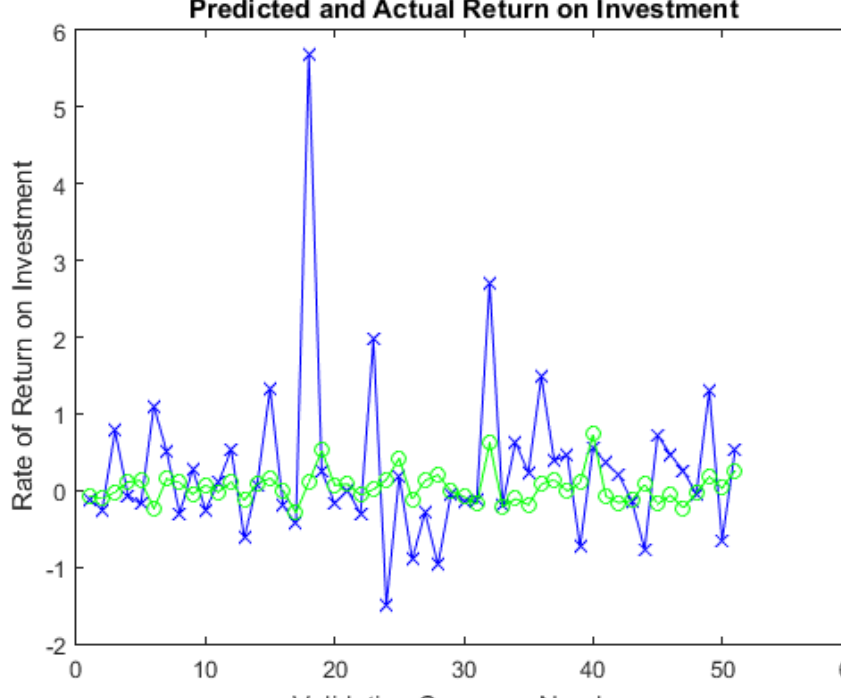
2	0.126667	<p>Predicted and Actual Return on Investment</p> 
3	0.114375	<p>Predicted and Actual Return on Investment</p> 

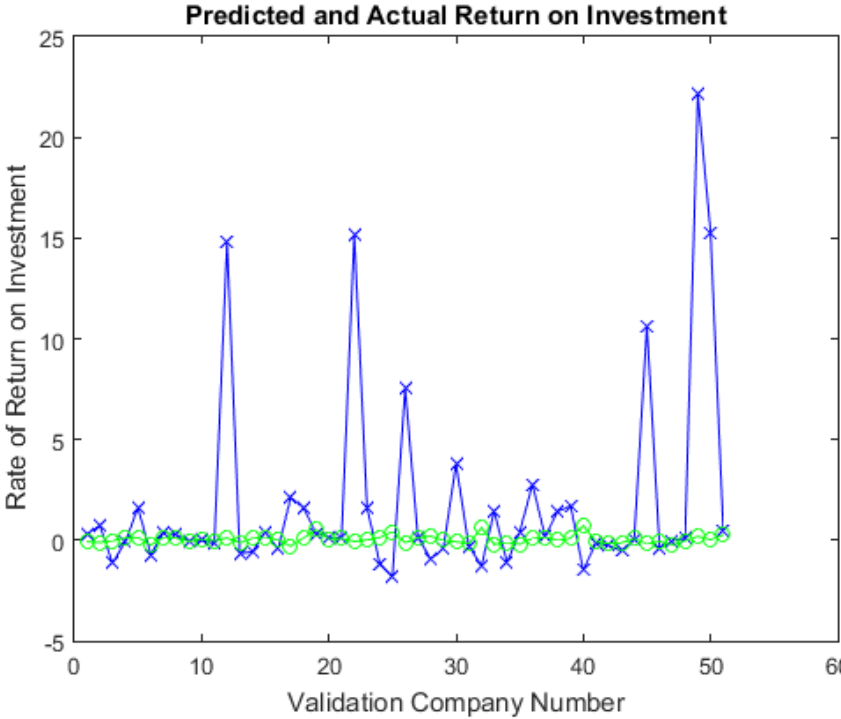
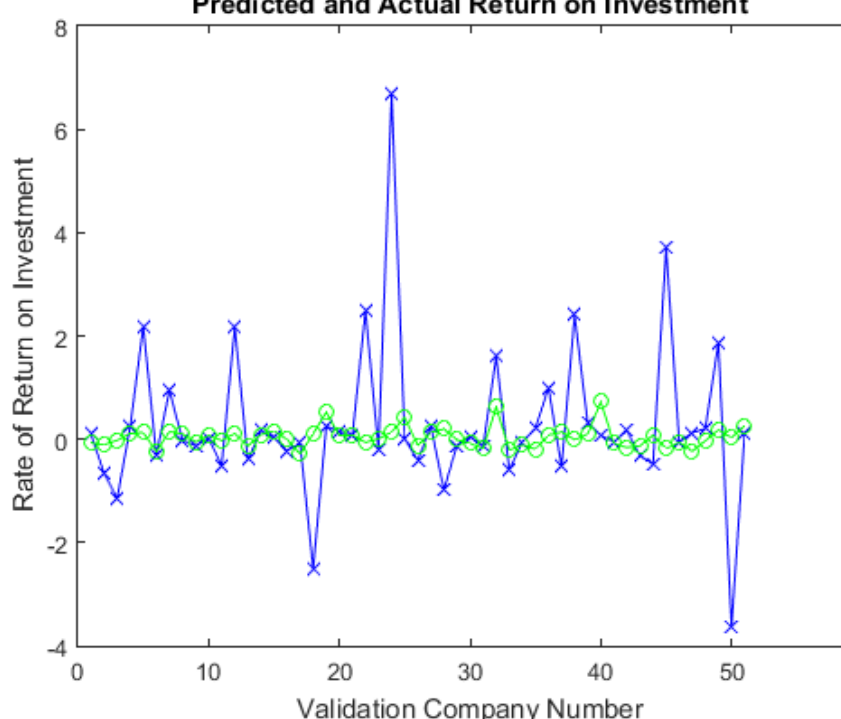


The returns for 100 interneurons have not necessarily proven to be better than the returns for 10 interneurons. The time it took to train on this many interneurons was only around 10 seconds per run, which is very efficient for such good predictions. The smallest amount of time was 1 second, which yielded the worst return rate for run 4. We can see that for the most part the graphs of predicted rates of ROI closely correspond to the graphs of the actual rates of ROI, which is a good indicator that our network is able to generalize its output for novel input. These graphs show better correspondence between predicted and actual than those for 10 interneurons, but it seems that the network begins to memorize patterns exactly, giving less generality. Hence, having a large number of neurons is less favorable overall than having a small number of neurons because as we can see, the network is being over-trained. Although the returns with more interneurons were high at points, they were also worse than the index fund at points, while the net with less interneurons always consistently generated a favorable (higher than index) return rate. This reveals that having too many interneurons in the network caused it to over-train, and the results produced were not as favorable for the network that was not trained as long.

Continuing, I experimented with another training method. This time I used the default 'trainlm' method with 10 interneurons and the random divide for training/test data. I used the default cap on the number of training iterations, 1000. My average rates of return on investment (ROI) for 4 consecutive runs along with their graphs were the following:

Run #	Avg. Rate of ROI	Graph

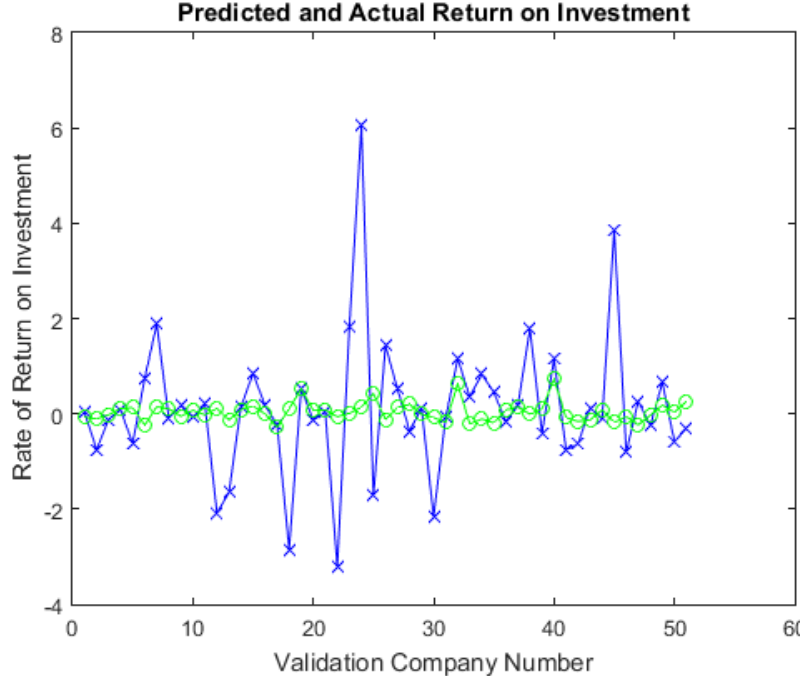
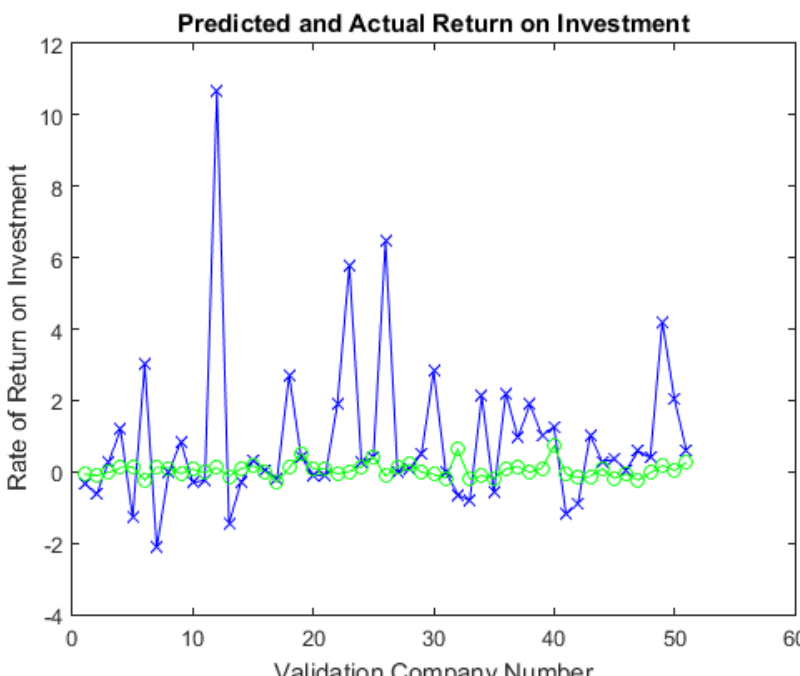
1	0.015294	<p>Predicted and Actual Return on Investment</p>  <p>Rate of Return on Investment</p> <p>Validation Company Number</p>
2	0.090000	<p>Predicted and Actual Return on Investment</p>  <p>Rate of Return on Investment</p> <p>Validation Company Number</p>

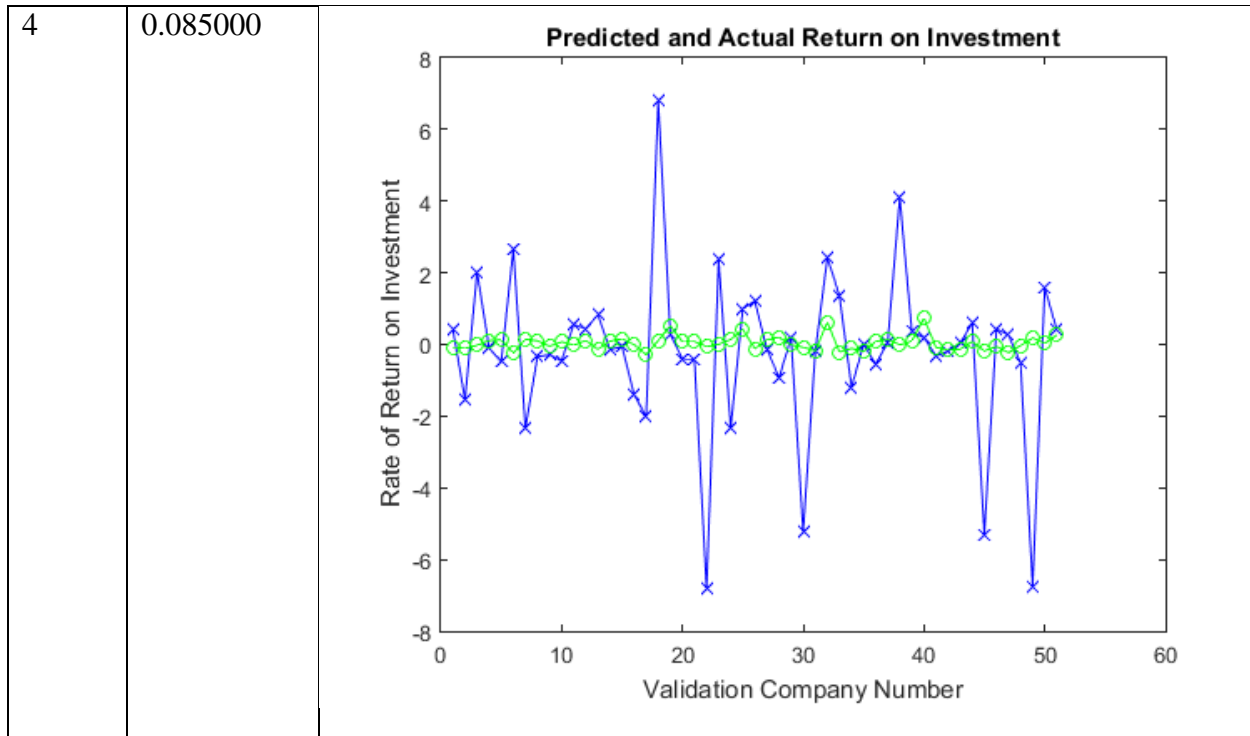
3	0.036071	<p>Predicted and Actual Return on Investment</p> 
4	0.087143	<p>Predicted and Actual Return on Investment</p> 

The training time taken for 10 interneurons with this method was usually about 15 seconds. As can be seen above, it seems that this method performed worse than the previous training method used. It revealed that only 50% of the time it will generate a favorable outcome. The graphs for this method also do not correlate predicted to actual data as well as the graphs for 10 interneurons in the previous method. Sometimes they show good correlation, but in general they do not. Perhaps this method of training is less favorable than the 'traincgf' method previously used. Let's see if increasing the number of interneurons will help this method produce more favorable results.

This time I used the default 'trainlm' method with 100 interneurons and the random divide for training/test data. I used the default cap on the number of training iterations, 1000. My average rates of return on investment (ROI) for 4 consecutive runs along with their graphs were the following:

Run #	Avg. Rate of ROI	Graph
1	0.072800	

2	0.057083	<p>Predicted and Actual Return on Investment</p> 
3	0.070333	<p>Predicted and Actual Return on Investment</p> 



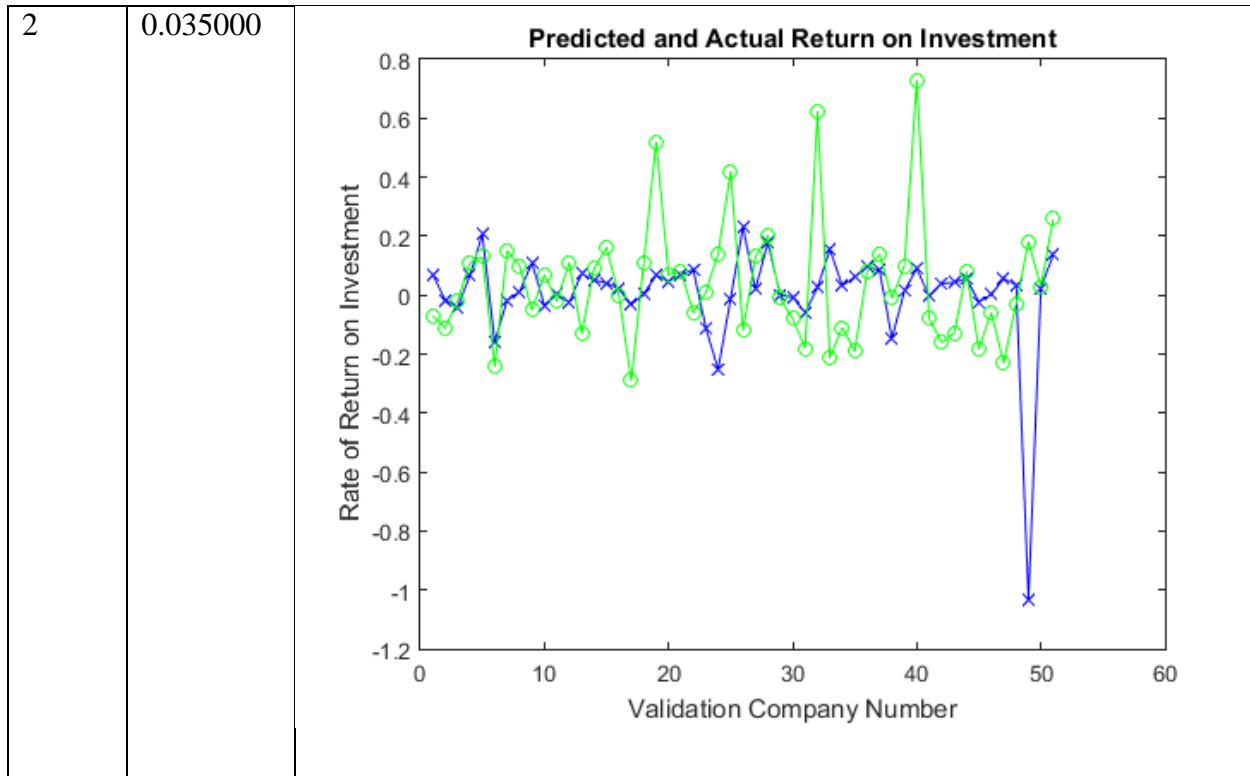
This method took much more time to train the network. The average time was around 50 seconds. The rate of ROI was always favorable though. This reveals that perhaps the method is more accurate with more interneurons rather than with less, as seen in the previous training method. However, we can still notice the poor correlation in the graphs with this method. It really seems as if this method of training is not favorable for our data type for some reason. The predicted rates of ROI are all over the place on the graph. This shows that although this method had good binary predictions (whether a stock will gain or lose value), it was not very accurate in the sense of predicting the actual rate of ROI as compared to the previous training method. However, it has been revealed that increasing the number of interneurons in the ‘trainlm’ method does in fact generate a more favorable outcome, whereas in the previous ‘traincgf’ method it does not. It does not seem that adding more interneurons cause much over-training for this training method, but it seems to all depend on the training method used. Perhaps some training methods respond better to more interneurons than others, as we just saw.

Overtraining:

The next thing I focused on was overtraining. It can be specified that a network will be over-trained given two conditions (or a combination of both). The first condition is that the network has too many interneurons which will lead to it purely memorizing its pattern inputs. The second condition is that it is trained for too many iterations and although it may have once found the most optimal answer, it overwrote that answer with a less favorable one. In the next experiment set, I examine the effect of overtraining a network.

In order to have a method of training that permits overtraining, I used the 'traingdx' method. This method uses gradient descent with momentum and adaptive learning rate backpropagation in order to train the network. From class, we know that gradient descent can sometimes overshoot the solution and keep percolating on training although it has already found potentially the best solution that it can. It will let you over-train the network permitting it does not reach the maximum number of iterations. Thus, I expanded the cap on the number of iterations to 2000 iterations using the 'trainParam.epochs' variable. I kept the random 50/50 divide of training and test data and set the number of interneurons to 200. My average rates of return on investment (ROI) for 2 consecutive runs along with their graphs were the following:

Run #	Avg. Rate of ROI	Graph
1	0.043333	



Hence we can see that as a result of over-training the network using both lots of interneurons and a large number of iterations, the results produced by training were not as favorable as the results produced on a network trained less but not under-trained, as in the previous experiment sets. Surprisingly, this method took an average of 10 seconds to complete with 200 interneurons, which shows that it is very fast. Despite the method used, it can be shown that over-training the network can cause it to wipe out good data that has been learnt by it, thus replacing the good knowledge with less-good knowledge and sometimes even bad knowledge.

Number of Interneurons:

Overall, it can be seen that the best number of interneurons varies per training method used. However, I have noticed that a number of interneurons between 10 and 200 has produced the best results for me. This is probably due to the fact that even at 200 interneurons, the network is still provided generalized prediction on output since there are approx. 230 companies in the training data. We can see that the number of interneurons should probably be less than the number of patterns in the training set given that the network will begin to memorize the data when the number of interneurons grows larger than the number of patterns, thus not making further generalized predictions. Sometimes there is a probably that a network will work great with more interneurons than the number of input patterns, but this would only happen if the data it would see in the future was very closely correlated with the input data it was trained on. Hence, I conclude that the best number of interneurons for a feed forward network with backpropagation should be between 10 and 200, provided the size of the training set is ~230 patterns. Narrowing in, I believe that the number of interneurons should be closer to the lower

bound of the range I provided in order to provide more generalizations given novel input data rather than trying to memorize the training data given a large amount of interneurons.

Once again, as in PS1, the network predicted very well the rate of return on the validation stocks when the number of interneurons was larger as compared to when the cluster count was larger, given the training method permitted this. This shows that the data provided is closely correlated between training and validation, and as the number of interneurons approaches the number of patterns in the training data, the network may perform better (obviously depending on the training method in this case). This rises mysterious questions as to why there is such accuracy in the predictions of the network with more interneurons along with the predictions in the k-means clustering method with more clusters for this financial dataset, but in general theory, fewer interneurons should provide better results on novel input than more interneurons should.