Comparing Algorithmic and Human Trading

Justin Chudley

William Paterson University

Cognitive Science Honors Thesis

Abstract

Trading on the stock market is moving away from humans trading stocks and towards computers making and executing trading decisions, known as algorithmic trading. While many traders attempt to define sets of rules for algorithmic trading, we propose using popular psychologically plausible algorithms - Linear Regression, Reinforcement Learning, and a Long Short Term Memory Neural Network - to determine rules for algorithmic trading. These algorithms were chosen due to their relationship to various models of learning in the human brain; each algorithm seeks to emulate a different theory of human learning. The effectiveness of each algorithm is compared to participants trading on a simulated stock market. Human results were mixed, with some participants being able to outperform the trading algorithms, but overall humans were outperformed by all computer algorithms. Reinforcement learning was the most profitable algorithm, followed by the neural network and then the linear regression.

Introduction

Traditionally, trading on the stock market was done by humans putting in trade orders on the wall street floor. As computers and networks became exponentially faster, algorithmic trading, or the use of computers to complete the stock trading process at high frequencies, has become the predominant method of professional trading (Chaboud, Hjalmarsson, Vega, & Chiquoine, 2011). Algorithmic trading strategies typically are defined by sets of rules that tell a computer program when to execute a trade (Ticknor, 2013). Popular strategies usually consist of various technical indicators, news on companies, world news, etc. (Dash & Dash, 2016). This study instead seeks to study various psychologically plausible statistical algorithms and apply them to the stock market. The effectiveness of these algorithms is then compared to the effectiveness of humans trading on a simulated stock market. The three algorithms used in this study were linear regression, reinforcement learning, and a neural network. These three were chosen due to their relationship with theories of human learning. While these algorithms will be explained in depth later in this paper, a brief summary of each can be provided in order to show their relationship to learning in the human brain.

A linear regression takes a series of independent variables and applies a unique weight to each. The resulting products are then summed up to generate a prediction for a dependent variable. When a human is shown a series of data and is trying to use this to make a prediction, a naive approach would follow a similar thought process. Assuming the human does not have any previous knowledge on a strategy for predicting values in this data set, a plausible method would be to would use this series of data, make assumptions on the importance of each data point (similar to weights in a linear regression), and use this to make a prediction. This strategy is similar to what a linear regression does, a linear regression does not have a method for learning

strategies, it instead takes a naive approach like a human who does not know a good strategy may.

Reinforcement learning is centered around the core idea in nearly all learning theories in which humans learn by interaction with their environment (Sutton, 1998). As humans interact with their environment, their actions have causes and consequences (Sutton, 1998). Every action taken produces an outcome. The model suggests that humans will always learn to maximize the amount of positivity an action produces for them and minimize the amount of negativity produced. As a reinforcement learning agent takes actions in an environment, it receives rewards. The previous rewards it has received help determine proper actions to take, and since the goal is to maximize reward, overtime the agent will learn which actions to take and which not to take, just like humans do (Sutton, 1998).

While the vast majority of the human brain functions are still unknown, neural networks seek to emulate the theorized function of neurons in the human brain (Arbib, 2003). Inside the human brain are roughly 100 billion neurons (Arbib, 2003). Neurons in the human brain are thought to receive and process information. Information is passed between neurons as electrical impulses, which are dampened or amplified as the brain learns which neurons are important for specific tasks (Arbib, 2003). Artificial neural networks attempt to replicate this system. An artificial neuron takes in input, processes it, and applies a weight to it before sending it off to the next neuron (Arbib, 2003). As the artificial neural network progresses, it will adjust the weights of its neurons over time, attempting to find the optimal weight for each neuron to learn how to complete whatever task it was designed to solve (Arbib, 2003).

Three Psychologically Plausible Algorithms

Linear Regression

When using data such as stock prices to predict future stock prices, finding trends in the data is necessary to make future predictions (Ticknor, 2013). Regression analysis is a subfield of statistics that seeks to find relationships between variables. One type of regression analysis, linear regression, seeks to model the linear relationship between a dependent variable and independent variables. In this model, the independent variables is denoted as $x_1$through x_n , and the dependent variable is denoted as y. The $x$variables can be thought of as the "predictor" variables, that is that it does not rely on the value of y to determine its value, whereas the $y$variable depends on x to determine its' value, thus $y$can be thought as the outcome variable. A linear regression aims to find a line of best fit, or hyperplane, for a graph of (**X,y**) values. Once the line of best is found, future x values can be substituted into the equation of a hyper plane, $\boldsymbol{Y} = \boldsymbol{X}\beta + \varepsilon$ which will predict future y values (stock prices).

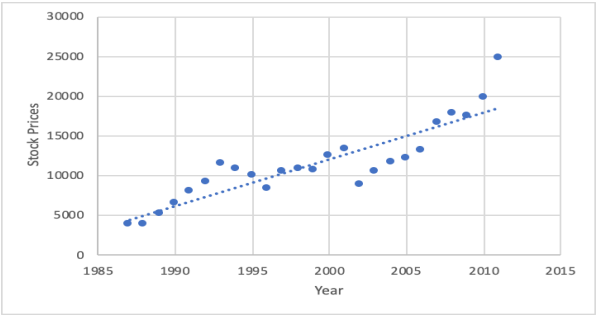

FIGURE 1. A Linear Regression on Dow Jones Data from 1985 - 2012. *By using the dates as the independent variable and the price on each date as the dependent variable, a line of best fit for the data is created. In this diagram,* $y = b + m(year)$.

The equation for a linear regression is: $\hat{y} = b_0 + b_1x_1 + b_2x_2 \ldots + b_nx_n$, where $\hat{y}$ is the expected value of the dependent variable y, x_1 through x_n are the independent variables, b_0 is

equal to the value of $y$when x_1 through x_n are equal to zero, and b_1 through b_n are the regression coefficients. A regression coefficient b_i represents the change in y relative to a change in x_i (Bremer, 2012).

A linear regression can also be thought of as a series of input nodes (independent variables and the intercept), which each get an individual weight (regression coefficients) applied to them. After weights are applied, the resulting values can then all be summed together, producing a predicted value.

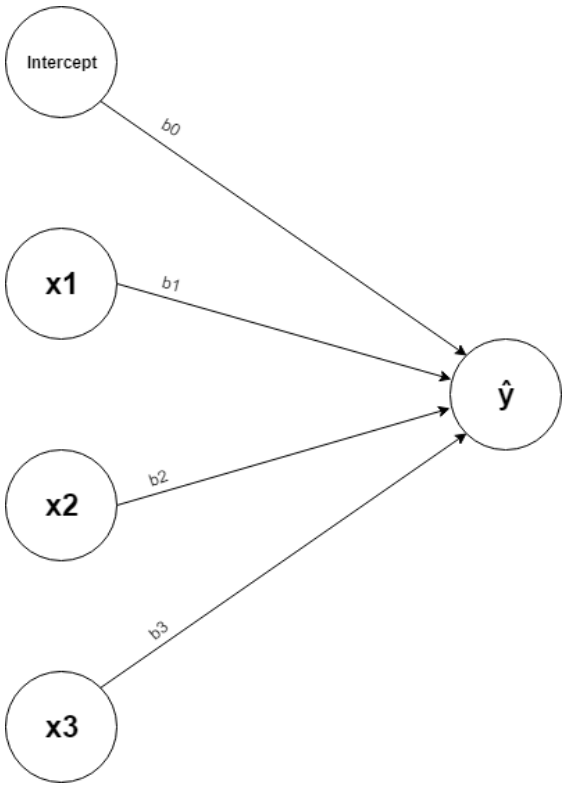

FIGURE 2. Visualizing a linear regression as a series of inputs nodes. *Each input node is multiplied by a weight, and then the resulting values are summed up to create a prediction.*

A linear regression for predicting stock prices could take in the three previous price points as independent variables and determine regression coefficients for each previous price

point. The independent variables are then multiplied by their respective coefficients and summed up to create a price prediction. This is similar to what can occur in the human brain; a human could feasibly look at the three previous prices and determine the importance of each individual previous price. Based off of the priorities applied to each previous price and the values of the previous prices, a prediction on the future price can be determined.

Reinforcement Learning

In 1972, Robert Rescorla and Allan Wagner created a mathematical model to describe the rate of learning in each trial in a series of trials (Rescorla & Wagner, 1972). Their research focused on learning through classical conditioning, in which a conditioned stimulus is associated with an unconditioned stimulus. For example, if a dog is known to salivate (unconditioned response) at the smell of meat (unconditioned stimulus), the unconditioned stimulus can be paired with a conditioned stimulus such as a bell ringing. To do this, one could ring a bell right before they present a dog with meat each time, and eventually just the sound of the bell alone would cause the dog to salivate. The Rescorla Wagner model describes the association of the conditioned stimulus and the unconditioned stimulus (Rescorla & Wagner, 1972). A strong association between the conditioned and unconditioned stimuli implies that the conditioned stimulus can predict the unconditioned stimulus. This model points out two general principles regarding learning:

1) If the expected outcome does not match the actual outcome, then the subject will learn.
2) The expected outcome of any trial is based upon the experiences from previous trials.

The equation for the Rescorla Wagner model is as follows: $\Delta V_x = \alpha_x \beta(\lambda - \sum_{t=0}^{x} V_t)$, where ΔV_x is the change in the predictive value of a stimulus V_x (the amount of learning that has occurred on a trial), α is salience of X (a value between 0 and 1 which describes the weight to give to the learning of previous trials), β is the association value (a value between 0 and 1) which is a predictor of how easy or difficult it is to learn, $\lambda$is the maximum value that the conditioned stimulus can have, and $\sum_{t=0}^{x} V_t$ is the total associative strength over all trials that have occured before V_x (Rescorla & Wagner, 1972). The model can be viewed as a simple reinforcement learning model, in which a reward is given at each step to encourage proper learning. With this, The equation can be rewritten to look like: $V_{s,t} = V_{s,t-1} + \alpha(r_t - V_{s,t-1})$, where $V_{s,t}$ is the value of stimulus $s$on trial t, r_{t-1} is the reward given at trial $t - 1$, and α is the learning rate. This equation shows that the value of the stimulus is updated based on the prediction error and the difference between the reward received and the predicted value from the previous trial (Schultz, Dayan, & Montague, 1997).

Reinforcement learning is a method for using an algorithm (known as the agent in reinforcement learning) to arrive at a desired output (known as the reward in reinforcement learning), without telling the algorithm how to arrive at the proper reward (Sutton, 1998). This method plays off the idea of reinforcement learning in psychology, such as teaching someone to do a trick via rewarding them for doing the proper trick and punishing them for improper execution of the trick (Watkins, 1992). It is up to the person being rewarded or punished to figure out what went wrong or right, and the person will then continue to adjust the trick based on reward and punishment until the trick is executed perfectly (Watkins, 1992).

In machine learning, the reinforcement learning agent's goal is to find its' maximum reward, self-adjusting over some number (typically a very large number) of iterations of trial and

error in an attempt to achieve this (Lee, 2001). The reinforcement learning strategy attempts to "learn" the solution to a problem without "knowing" the proper method for arriving at the solution (Lee & O, 2002). Reinforcement learning consists of an "agent" and an "environment" in which the agent acts. The agent chooses an action to take which changes the state of the environment; by choosing an action, the state of an environment is changed, and based on the outcome of the change in the state of an environment, the agent is provided with a reward. Since an agent's goal is to maximize the amount of reward it receives, rewards are used by the agent to encourage the agent's choice of action to move in a direction towards the proper solution to a problem.

Building off of the previously discussed Rescorla Wagner model of learning is a reinforcement learning model known as *Q*-learning. *Q*-learning is a model free reinforcement learning algorithm (Watkins, 1989). A reinforcement learning algorithm is considered to be model-free when the algorithm does not attempt to generate predictions of the future state of the environment and the future reward, but instead relies solely on data from the environment (Watkins, 1989). The goal of a *Q*-learning algorithm is to learn an optimal policy that can be used to determine the best action for an agent to take (Watkins, 1989). *Q*-learning is especially well-suited for learning policies for temporal sets of data, such as stock prices over time (Lee, & O, 2002, Lee, Hong, & Park, 2004, Lee, Park, O, Lee, & Hong, 2007). It is mathematically guaranteed that given sufficient data and training time on any policy, *Q*-learning will converge with a probability of one to the best action function for a policy (Watkins, 1992).

In *Q*-learning, a *Q*-table is used to keep track of each state of the algorithm and each action taken at said state. A *Q*-table is a matrix array with states as the new column labels and

actions at the new row labels. For each action taken at each state, a reward is given, which is encoded into the corresponding location in the *Q*-table.

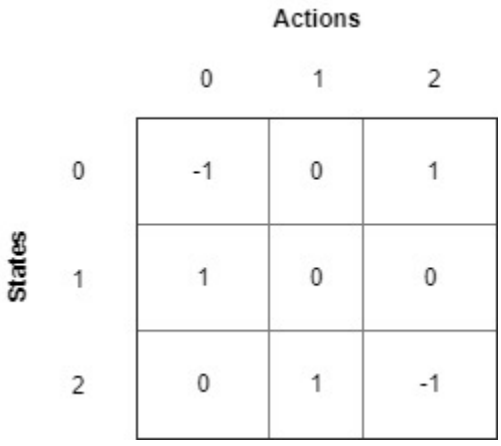

FIGURE 4. A simple Q-table with three possible actions and three states. *The reward in each state and action is represented inside the table. In each state, an action is chosen. The chosen action is then used to calculate a reward, which is then stored in the corresponding row and column.*

As shown in Fig. 4, for each state, some action was taken. Based on that action, a reward was calculated and stored into the corresponding row and column in the matrix. In order to choose an action for each state as well as calculate a reward, the Q-learning algorithm must be used.

The equation for a Q value at a given state and action t is defined by: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[r_{t+1} + \gamma maxQ(S_{t+1}, a) - Q(S_t, A_t)]$ (Watkins, 1989). However, this algorithm implies a future state, $\gamma maxQ(S_{t+1}, a)$. In this study's stock market Q-Learning situation, there are not future states. Because of this the researchers propose editing the Q-learning equation to account for this. When this is done, the equation can be defined as: $Q(S_t, A_t) = Q(S_{t-1}, A_t) + \alpha[r_t - Q(S_{t-1}, A_t)]$.

This equation now looks almost identical to the Rescorla Wagner model, however a Q value with a state and action is used instead of an association between stimuli. By abstracting our equation even further, the similarities to the linear regression equation and Q-learning can also be clearly seen if the equation is represented as the value function proposed in Glimcher, 2011 $Q(S_t, A_t) = \alpha^1 r_t + \alpha^2 r_{t-1} + \alpha^3 r_{t-2} + \alpha^4 r_{t-3} \ldots + \alpha^t r_0$ (Glimcher, 2011).
In this form, the Q-learning equation is almost the exact same equation as linear regression with a fixed learning rate, α, instead of regression coefficients.

Reinforcement learning for predicting stock prices could have two possible actions, buy and sell. Each time the algorithm chooses an action, a reward will be determined based on the profitability of the chosen action. The reward helps determine which action will be chosen next. In the human brain, a similar form of learning occurs. A human learning to trade on the stock market will have the same possible actions. If the action they choose is not profitable, then they will learn to not to take that same action in a similar situation. If the action was instead profitable, then they will learn that to take that action again in a similar situation.

Neural Network

Where simple rules are ineffective at finding a pattern, neural networks can excel (Hochreiter & Schmidhuber, 1997). These networks use algorithms that are designed to mimic the structure of neurons in the human brain (Hinton, 1992). Artificial neural networks "learn" the relationships between inputs (dependent variables) and the related outputs (independent variable). To do this, a neural network trains on a subset of the total data set. It is given some amount of the total dependent variables as inputs and the associated independent variable which should be produced by the inputs as outputs. The network trains on this input data to adjust its'

internal weights with the goal of finding the correct weights that will take in inputs and multiply them by the weights to produce the proper outputs given to the network in training (Tu, 1996).

To understand in depth how a neural network accomplishes these tasks, it is best to break the structure of a neural network down into its layers. Artificial neural networks are made up of layers of neurons. There are three types of layers in the structure of an artificial neural network: the input layer, the hidden layer(s), and the output layer, as shown in Fig. 3.

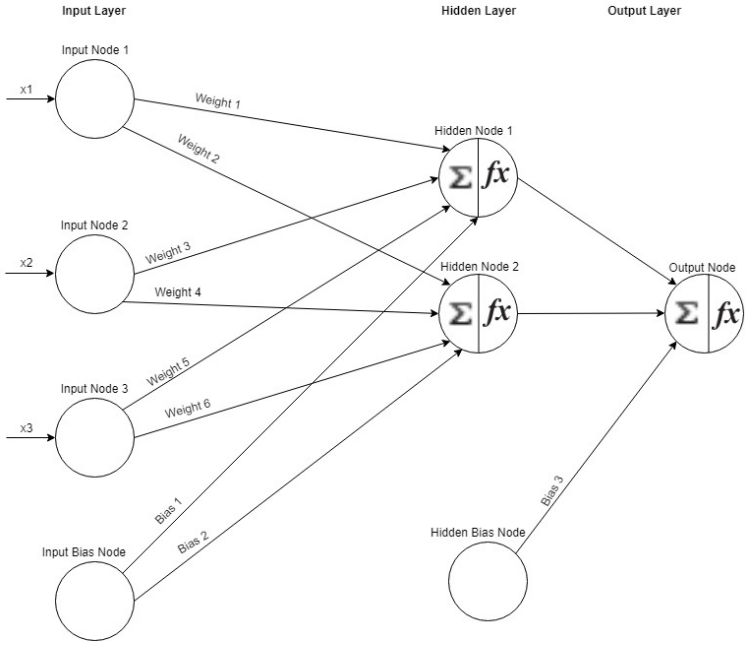

FIGURE 3. Visual representation of an artificial neural network. *Inputs are passed to each hidden node alongside a weight. Hidden nodes multiply each input by its weight and sum them. An activation function (f(x)) is applied to the summed results plus the biases. Results are passed to output node, summed again, another bias is added, independent variable is predicted (Tu, 1996).*

The input layer of a neural network is defined with *n*number of nodes, where *n* is equal to the number of independent variables. Each input node accepts one dependent variable, which

it then passes a copy of to each neuron in the hidden layer. During the passing of the independent variable to each neuron, a randomly initialized weight is applied to the independent variable. As the algorithm progresses through training iterations, these weights will change over time, with the goal of finding the combination of weights that will result in the proper (or as close to proper as possible) output. The resulting weighted inputs are then summed by each hidden neuron in the hidden layer, and then some randomly initialized bias (bias will adjust itself over time, just like weights) is added to the summation result (Nielsen, 2015). The hidden layer then applies an activation function (see: Activation Function section) to the number generated by the sum of the summation of weighted inputs and the bias and produces an output. If there is only one hidden layer in the neural network, then these outputs are all passed to the output node(s), otherwise they all pass each of their outputs to each node in the next hidden layer in the same manner that the input nodes passed their inputs to this hidden layer. Once the final hidden layer passes its' output to the output node(s), the output node(s) sums them, applies a bias to them, and then applies an activation function. The resulting value(s) are the predicted value(s) for the dependent variable(s). The equation for calculating the results of a neuron is as follows: $a_j^l = f(b_j^l + \sum_{k=0}^{m}(w_{j,k}^l a_k^{l-1}))$ where l is the current layer, j is the current neuron, $w_{j,k}^l$ is the weight on layer l going from neuron k on layer $l-1$ to neuron j on layer l, a_k^{l-1} is neuron k in layer $l-1$, f is an activation function, and b_j^l is the bias on neuron j in layer l (Nielsen, 2015).

In order to adjust weights, the network calculates the difference between the predicted output from the training input and the known proper output from the training data. To do this, the network calculates what is known as the error of the network (Nielsen, 2015). There are many different error calculation functions, however the method generally assumed to be most the effective (and the one used in this study) is the mean squared error (MSE) (Nielsen, 2015). The

mean squared error can be calculated by the following equation: $MSE = (1/n)\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$, where n is the number of predictions made by a network, y_i is the ith actual value from the training data, and $\hat{y}_i$ is the ith predicted value from the training inputs. A neural network can then use the resulting mean squared error to "teach" itself how to properly adjust its weights through a method known as backpropagation. The exact methodology of backpropagation is deeply complex, with entire chapters of textbooks devoted to explaining it, and out of the scope of this study (see: Nielsen, 2015, chapter 2 for an in-depth explanation). If the mean squared error does not equal 0 -- that is, if there is a difference between the predicted value and the actual value -- then an error signal is generated. The error signal is the result of equations involved in backpropagation which tells the weights in the network how much to adjust by to decrease the difference between the predicted values and the actual values. After a network is trained, it can then take in dependent variables as input (separate from those used in the training set) and predict what the independent variable should be based on the input.

While neural network algorithms are more complex than the linear regression algorithm, that is not to say they do not share similarities. As can be seen when comparing Fig. 2 and Fig. 3, both a linear regression and a neural network take in independent variables, apply some weights to them, and produce a predicted value for a dependent variable. A linear regression has a "bias", b_0, similar to a neural network. A neural network essentially expands on the idea of a linear regression, by adding different activation functions, hidden layer(s), and adjusting its weights over time as the network trains. As discussed earlier, the Q-learning algorithm is extremely similar to linear regression, which also makes it highly related to the neural network algorithm.

Humans trading on the stock market is a continual learning process as the stock market is ever changing. Each trade a human makes on the stock market affects the decisions they make on

future trades. In addition, humans consider past information from the market to inform their future decisions. Inside the human brain, neurons learn over time how to make the best decisions for stock market trading. Linear regression, reinforcement learning, and neural networks all seek to model methods of human learning. Linear regression makes a prediction based on past information. Reinforcement learning evaluates the results of past decisions to make better decisions in the future. Artificial neural networks seek to model the neural networks in a human brain to learn how to predict stock prices. A human brain uses all of these components in combination. This study isolates each component into a separate algorithm and compares them to each other as well as to humans trading on the stock market.

Methods

Participants

This study was participated in by 11 university students from William Paterson University. No demographics were collected from participants such as race, gender, age, etc. All participants participated voluntarily and were informed that no information about them would be collected, and that only their responses to the study and their reaction times would be recorded. Participants were recruited for this study by being asked in person if they would take part in the study. This study was approved by the William Paterson Institutional Review Board.

Materials and Procedure

This study was conducted on the same machine for all participants. Participants performed the study on a 13-inch Macbook Pro. The study was programmed with HTML, CSS,

Javascript, and a Javascript library known as JSpsych. At the start of the study, a new Google Chrome window opened on the Macbook Pro.

The window was an all-white background and the center of the window stated: “The experiment will switch to full screen mode when you press the button below” with a clickable “continue” button below the message. After clicking continue, a new all white screen displayed: “Wait until you are told to begin, then hit any key for the instructions.” Participants were then told they could continue when ready. Upon hitting any key to continue, an instruction screen appeared. The instruction key stated the follow instructions:

> *During this experiment you will be buying and selling shares on a simulated stock market. You will start with $10,000 and 5 shares. You will be shown dates and the price of the stock on that date. You will then either buy or sell a share on that date. Place your left index finger on the 'F' key and your right index finger on the 'J' key and leave them there for the entire experiment. Press the 'Space Key' to begin.*

After the participant read the instructions and hitting the space key, the experiment began. The first screen shown in the experiment was a screen which showed the initial portfolio balances of the participant. “Account Balance: $10,000” was displayed in the center of the screen, and then directly underneath that message on a new line the screen would show “Shares: 5.” This screen appears for 4000 milliseconds (4 seconds) and cannot be skipped earlier; the screen will automatically change to the next screen after 4000 milliseconds.

The second screen of the experiment shows in the center of the screen a date (the first time this screen is shown it shows the date 12/31/1914) and below the date the closing price of the Dow Jones Industrial Average on the date was shown (the first time this screen was shown it showed the price $54.63). At the bottom center of this screen, the message “Press Space to

continue" was shown. The length of time participants spent on this screen was recorded into the study's data.

Once space is pressed, a new screen appears. This new screen shows the word "Buy" with the phrase "(F)" below it to indicate the 'F' key is used to purchase a share of the stock at the previously shown price. Next to the "Buy" word is the word "Sell" with the phrase "(J)" underneath it to indicate the 'J' key is used to sell an already owned share at the previously shown price. Once either option is chosen, the response is recorded in the study's data.

After either the 'F' or 'J' keys are pressed, a new screen will appear showing updated portfolio balances. This screen will look similar to the previously mentioned portfolio screen with a few new additions. Under the "Account Balance: $..." text is a new set of text that states the change in the participant's account balance, i.e. "+54.63" or "-54.63". If the participant lost money, then the change in balance will be shown in red text, and if the participant gained money, the change in balance will be shown in green. Under the "Shares: …" message, is a new message that alerts the participant to their change in the number of shares owned, showing either "+1" in green text or "-1" in red text. This screen is shown for 4000 milliseconds and then the next screen appears.

The study repeats the same three previously mentioned screens until the end of the study. The screen that shows the date and the price of the Dow Jones changes to show the next year's date and price. The date is one year after the previous price, December 31st of the next year. For years that the stock market was closed on December 31st, the next closest date in December was used. Participants go through 104 iterations of this, starting with December 31st, 1914 to December 29th, 2017. After each choice of buy or sell, the portfolio balance and shares owned update and are reflected on the screen that shows them. After participants completed all 104

iterations of the study, the data from their iterations is saved into a comma separated value (CSV) file. The time it took participants to go through each screen, each buy and sell choice they made, and their account balance and shares owned for each iteration were recorded into the CSV file.

Models

Linear Regression

In this study, the dependent variable was the closing price of the Dow Jones on the last day the market was open on each year from 1914-2017. The linear regression in this study has 5 independent variables, x_1 through x_5, x_1 being the price the year before y, x_2 being the price the two years before y, etc. The goal of this linear regression for this study was to make predictions on the price of the Dow Jones, and then use that to simulate making trades on the stock market.

To accomplish this, the researchers wrote a computer program in the Python programming language. First, a portfolio object was defined in the program. The portfolio was defined to have a starting balance of $10,000 and 0 stocks initially owned. The portfolio object was also set up to have variables to keep track of the total assets (the amount of money the portfolio has plus the amount of stocks owned multiplied by the current price of the stocks). Instead of just taking a linear regression using all of the data available to the researchers and using that to predict future prices of the Dow Jones, linear regressions were taken on incremental sets of the data. At each price point, y, a linear regression was calculated using $y$and the corresponding price points of the five years before it, x_1 through x_5. For the early $y$values, y_1, y_2, y_3, y_4, y_5 where there was not a full five years of previous prices to use for x_1 through x_5, zero was substituted in for whichever x values were not available.

The linear regressions produce an equation, $\boldsymbol{Y} = \boldsymbol{X}\beta + \varepsilon$, which is the equation of the line of best fit. By plugging in the timestep $t + 1$ for x in the $y = mx + b$ (the linear regression function in the program will provide the values of **M** and b), the predicted price for y_{t+1} will be produced. If the predicted value for y_{t+1} from the line of best fit equation was higher than the actual y_t value from the data set, a stock was purchased. If the predicted value for y_{t+1} was instead less than the actual y_t value from the data set and the portfolio contains at least one stock share, a share was instead sold. If the predicted value for y_{t+1} was less than the value of y_t but no stocks were owned, then a share was neither sold nor bought. This process was then repeated for each timestep, t.

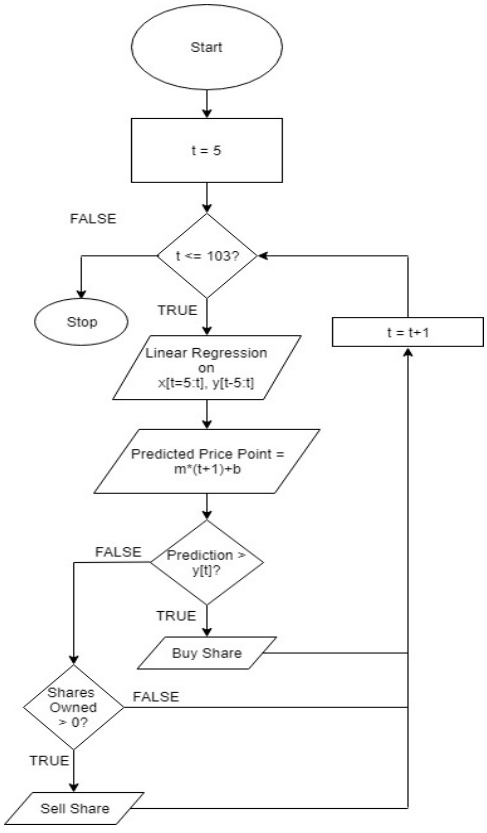

FIGURE 5. Flowchart of the linear regression program used in this study.

Once all linear regressions were completed, the total assets were calculated, and the initial starting assets of $10,000 were subtracted from them to calculate the overall profit of the algorithm.

Reinforcement Learning

To apply the Q-learning algorithm to stock prices, the environment, the agent, and the actions must first be defined. The environment is composed of a set of states; in this study, each state is equal to a last closing price for a year of the Dow Jones Industrial Average from 1914-2017. The actions that are executable on the environment (each individual price point) are one of two options, buy a share of the Dow Jones at particular state (price), or a sell a share. The agent in this study has a stock portfolio that contains the total net worth of the portfolio, the liquid cash available for buying shares, the total profits generated.

The implementation of the Q-learning algorithm for this study was written in the C++ programming language. While Python is typically the language of choice for machine learning and data science as it has many prebuilt libraries for these tasks, the researchers felt that the *Q*-learning algorithm was simple enough to code without having to use any prebuilt libraries. Combined with the fact that C++ has inherently much faster than Python, the researchers felt like it would be better to program this algorithm in C++.

The agent for Q-learning was defined to start with $10,000 and five open positions in its portfolio. The states were defined as the prices stored in a comma separated value (CSV) file which contained the historical closing price for the Dow Jones on the last day of the year the stock market was open December 1914 up to December 2017. The program started at the first state, with each successive state being the next year's last historical closing price.

At each state, the program must choose an action. To determine the action to use, the program uses a softmax function. A softmax function is a mathematical function which creates a probability distribution of some number of probabilities. In the case of this program, there are only two possible actions to choose from, thus $n$is equal to two. The softmax function used by this program can be defined as $P(sell) = 1/(1 + e^{(Q(sell)-Q(buy)/\beta)})$ where β is some predetermined number. By giving the softmax function the Q-values of each possible action, the probability of choosing each action as the next action can be determined. After obtaining the probability of choosing the sell action, the program can then generate a random number between zero and one, and then look to see if that randomly generated number is greater than the value of $P(sell)$. If the randomly generated number is higher than the value of $P(sell)$, then the buy action was chosen, otherwise the sell action was chosen.

Once an action is chosen, that action must be executed. If the chosen action was the buy action, then the program will "purchase" one share of the Dow Jones by subtracting the value of the current state (price) from the agent's liquid cash and increasing the amount of positions currently owned by one as well as storing the price that the share was "purchased" at. If the sell action was chosen, then the program will instead "sell" all its current positions and adding the cash value of those positions into the liquid cash the agent has.

After an action has been executed, the program has to determine a reward for taking that action. The reward of an action in a state in this program was defined as the change in total assets an action caused from one state to another. Thus, if an action was good and made a profit, a positive reward was given, and if the action was a poor choice and money was lost, a negative reward was given.

After calculating the reward for an action at a given state, the program updates the Q-values in the Q-table for each action at the current state. If the action was not the chosen action, then the Q-value from the last state simply carries over and becomes the Q-value at the current state. For the chosen action, however, the Q-value must be calculated using the equation discussed in the introduction.

Once new Q-values have been calculated, the program has reached the end of the algorithm for the current state. The program then will iterate over the same processes discussed above for each state, until it reaches the terminal state (the end of the states).

The reinforcement learning algorithm has two parameters, $\alpha$and β. Because of this, the reinforcement learning program was run 100 times on every combination of alpha and beta, with alpha incrementing from zero to one by 0.01 and beta incrementing from zero to ten by 0.1 each time. The average profits of each combination of alpha and beta were recorded.

Neural Network

Activation Functions

Activation functions in neural networks are used to determine if a specific neuron should be activated or not, and if so, how much the neuron should be activated (Karlik & Olgac, 2010). While there is a plethora of different activation functions that can be used for neurons, this study uses the sigmoid activation function and the hyperbolic tangent (tanh) activation function.

The sigmoid activation function is represented by the formula $f(x) = 1/(1 + e^{-x})$, where x is equal to the summed value of a neuron's weighted inputs plus their bias (Karlik & Olgac, 2010). A sigmoid activation function produces some value between zero and one, which can be used to not only determine if the neuron should be activated, but also how much weight

should be put on its activation; if the result is zero, then the neuron will not fire at all, if the result is 0.5, then the neuron will fire at 50%, etc.

The hyperbolic tangent function is defined as $tanh(x) = sinh(x)/cosh(x)$ (Karlik & Olgac, 2010). From this, we can represent the hyperbolic tangent function in a simpler to understand form: $tanh(x) = sinh(x)/cosh(x) = (e^x - e^{-x})/(e^x + e^{-x})$ (Karlik & Olgac, 2010). Visually these activation functions can be shown as:

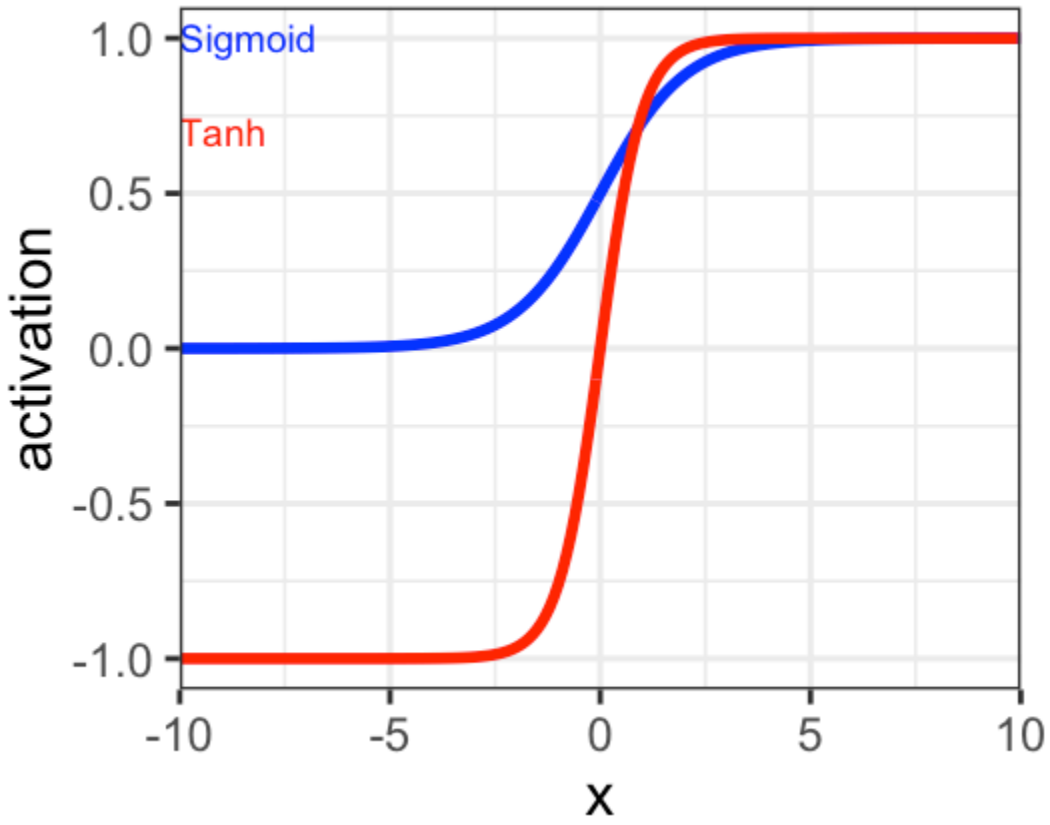

FIGURE 6. Hyperbolic tangent function (tanh) and Sigmoid activation function. *The sigmoid function is represented in blue, and the tanh function is represented in red. All values of y in the tanh function are bounded between -1 and 1. In the sigmoid function, all values of y are bounded between zero and one.*

The hyperbolic tangent activation function is similar to the sigmoid activation function; however, as shown in Fig. 6, the increase and decrease in activation values is steeper than that of the sigmoid function.

Recurrent Neural Network

The data used in this study can be defined as time-series data. Time-series data is a set of data that spans an amount of time, which makes the prediction problem in this study a time series prediction problem. Finding patterns in data that spans over long periods of time such as the 103 years looked at in this study can be a difficult task for a traditional neural network (Brezak, Bacek, Majetic, Kasac, & Novakovic, 2012, Khaldi, Chiheb, & Afia, 2018). In recent years, advancements have been made to the traditional neural network model to create newer models that better suit time series prediction problems, specifically a class of neural networks known as recurrent neural networks (Khaldi, Chiheb, & Afia, 2018).

Recurrent neural networks use all the same logic and layers as a traditional neural network; however, they also include what are known as hidden states inside the hidden layers (Mikolov, Karafiat, Burget, Cernock, & Khudanpur, 2010). A hidden state stores the output from a hidden node, which it then passes back into the hidden node at the next state alongside the hidden node's inputs. For a hidden node in state s, the hidden node receives inputs for state s and also receives as input its previous state, $s - 1$.

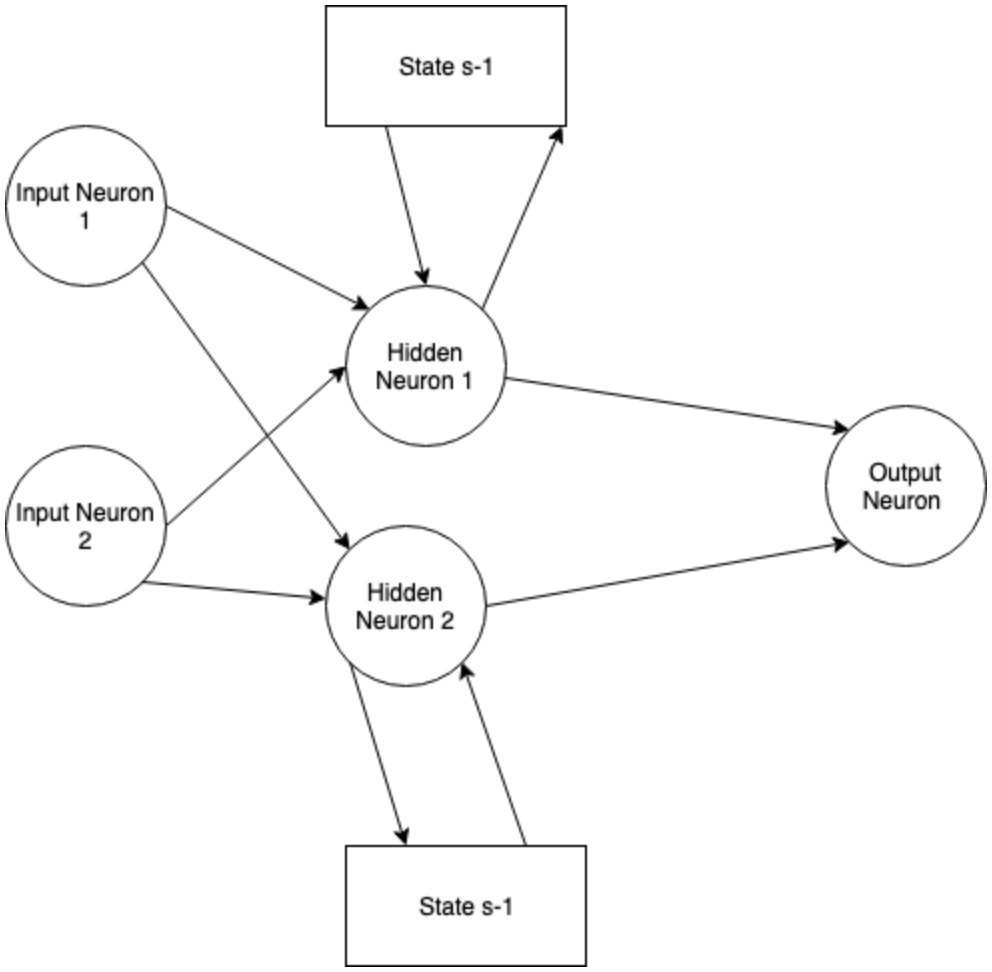

FIGURE 7. A simple recurrent neural network, in which the hidden nodes send their state to both the output as well as to a hidden state. *The hidden state is then given as input alongside the normal inputs to the next state of the hidden node.*

The addition of feeding previous states back into hidden layers is especially useful for time series data, as it allows the network to "remember" previous time steps to more accurately predict relationships within time series sets (Khaldi, Chiheb, & Afia, 2018).

Long Short Term Memory Recurrent Neural Network

When a recurrent neural network uses backpropagation over a long period of time series data, a problem known as the vanishing gradient problem arises (Nielsen, 2015). As training data which covers a long period of time is iterated through, backpropagation calculates gradients for each weight in the layer. Backpropagation starts calculating gradients at the output layer, and works backwards through the network, calculating the gradients for the input layers last (Nielsen, 2015). The gradients are a number between 0 and 1, and the gradient of a node in a layer is equal to the product of all gradients at prior levels (Nielsen, 2015). As numbers between 0 and 1 are multiplied together, their products become smaller and smaller. Because of this, as backpropagation progresses through the network calculating gradients, the gradients become smaller and smaller (Bengio, Simard, & Frasconi, 1994). By the time backpropagation is calculating the gradients for the input layers, the result can end up being an extremely small fraction (Bengio, Simard, & Frasconi, 1994). This causes the updating of weights in the input layer to barely occur, which in turn causes the network to have difficulty learning, since the input weights are essentially not updating, which leads to the vanishing gradient problem (Bengio, Simard, & Frasconi, 1994).

To avoid the vanishing gradient problem in time series prediction recurrent neural networks, a subfield of recurrent neural networks known as Long short term memory (LSTM) neural networks can be used (Hochreiter & Schmidhuber, 1997). Long short term memory neural networks were theorized by Hochreiter and Schmidhuber specifically to solve the vanishing gradient problem.

Instead of using hidden neurons, long short term memory neural networks replace them with what is known as memory cells (Hochreiter & Schmidhuber, 1997). Memory cells introduce

a second output to a neuron, cell state, which contains three gates: the input gate, forget gate, and output gate (Hochreiter & Schmidhuber, 1997). The input gate in a memory cell controls whether or not a cell is updated, the forget gate controls whether or not a cell's memory is set to zero, and the output gate controls whether or not the output of the current cell state is made visible (Hochreiter & Schmidhuber, 1997). All three of these gates use a sigmoid activation function. This causes the gates to all produce a value between zero and one. Since each gate is controlling whether or not something occurs, a value between 0 and 1 allows for either complete flow (value greater than zero) or no flow (a value of zero) (Hochreiter & Schmidhuber, 1997). The memory cell also contains a cell state modifier, which is a function that changes the state of the cell (assuming the forget gate is off and the input gate is on) (Hochreiter & Schmidhuber, 1997).

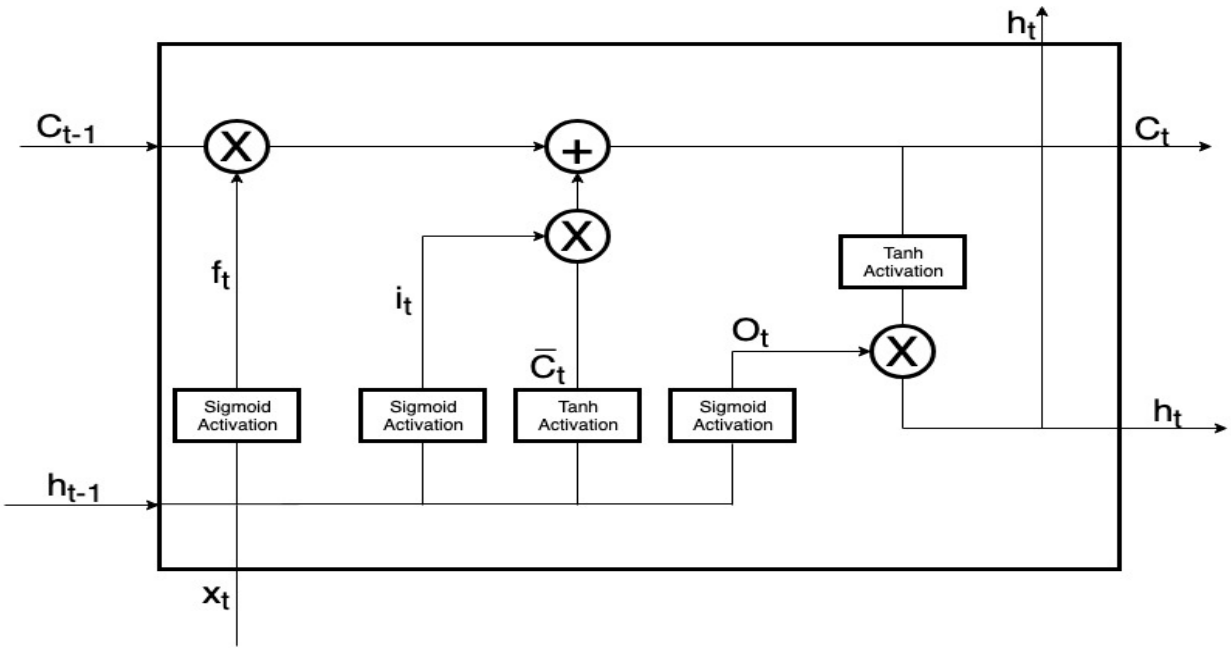

FIGURE 8. An LSTM memory cell. *Where C = cell state, h = hidden value, f = forget gate, i = input gate, O = output gate, $\bar{C}$ = the cell state modifier, x = input, and t = iteration number. The memory cell receives a cell state and a hidden value as input. The hidden state is passed as input to the forget gate, the input gate, the cell state modifier, and the output gate.*

Fig. 8 shows a visual representation of a memory cell, where:

$$i^{(t)} = f(W^i[h^{(t-1)}, x^{(t)}] + b^i)$$

$$f^{(t)} = f\left(W^f\left[h^{(t-1)}, x^{(t)}\right] + b^f\right)$$

$$o^{(t)} = f(W^o[h^{(t-1)}, x^{(t)}] + b^o)$$

$$\bar{C} = tanh(W^C[h^{(t-1)}, x^{(t)}] + b^C)$$

$$C^{(t)} = f^{(t)}C^{(t-1)} + i^{(t)}\underline{C}^{(t)}$$

$$h^{(t)} = tanh(C^{(t)}) * o^{(t)}$$

Where $\boldsymbol{W}$ is a matrix of weights and $b$is bias.

A memory cell takes in the state of the previous memory cell, an input, and the hidden state of the previous cell (the same hidden state as in a recurrent neural network). The hidden state is passed to a sigmoid function (forget gate), to determine if the cell should forget its previous memories (results from previous time steps) or not (Hochreiter & Schmidhuber, 1997). It is then passed to another sigmoid function (input gate) to determine if the cell should update its memories or leave them how they are (Hochreiter & Schmidhuber, 1997). It is important to note that the forget gate comes before the input gate. This allows for the cell to clear its memory, but still add new memories from the current state. Next, it is passed to the cell state modifier. If the input gate is on, then the cell state modifier will produce some value to be stored in the cell state (Hochreiter & Schmidhuber, 1997). After the cell state modifier, the hidden state is passed to the output gate. If the output gate decides to output the cell's hidden state, then the hyperbolic tangent of the cell state is passed to the next memory cell as the hidden state input (Hochreiter & Schmidhuber, 1997). Because of the inclusion of the forget gate, cell state information is not always lost over time (Pascanu, Mikolov, & Bengio, 2013). This controls the rate at which

information is forgotten by the network, avoiding the vanishing gradient problem (Pascanu, Mikolov, & Bengio, 2013).

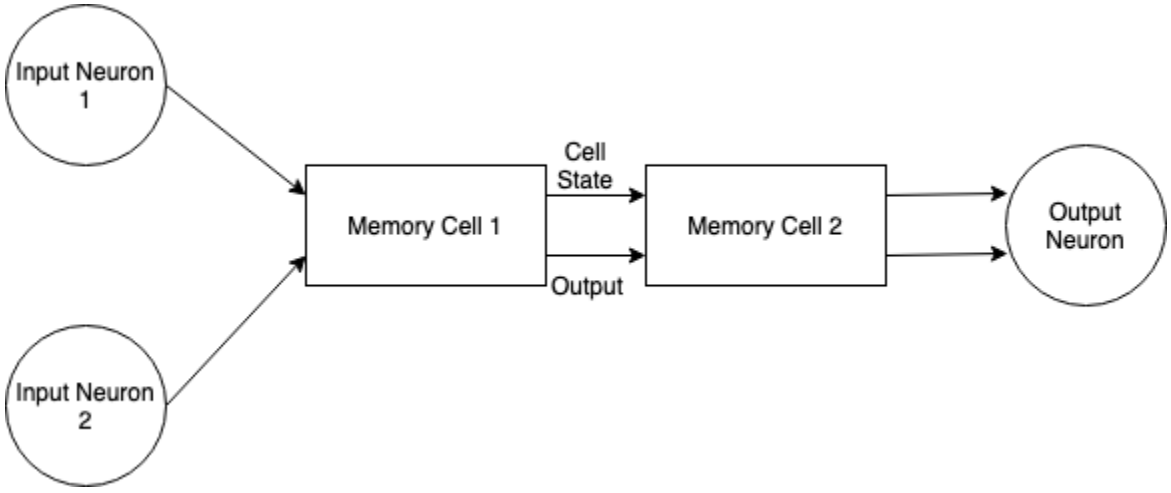

FIGURE 9. A simple LSTM network with 2 input neurons, 2 memory cells, and 1 output neuron. *Input flows from the input layer neurons into memory cell 1. From there the cell state and hidden value are passed to the next memory cell, then it is passed to the output neuron.*

In this study, a long short term memory neural network was implemented in the python programming language. The network consisted of five input nodes, a hidden long short term memory layer of 50 memory cells, a second long short term memory hidden layer with 100 nodes, and a single node in the output layer. The same data set of the Dow Jones was used for this algorithm as the linear regression algorithm. For input, the network takes in five independent variables, $x_1$through $x_5$to be used to predict a price y. Just as in the linear regression, the independent variables represent the five previous years before the year that is being predicted. The input data was split into two chunks, the data the network would use to train on, and the data

that the network would test its learned state on. The network was run three different times, once with the training data consisting of 50% of the data, once with 60% of the data, and once with 70% of the data. Before any data was passed into the neural network, all data was scaled down to a value between 0 and 1. This is standard to do to one's data when using a neural network, as they are computationally intense and scaling the values down vastly increases the execution time of the network. Predictions of the network were scaled back to their actual values.

After scaling, the network takes in the training data. During training, the network takes in independent variables 5 at a time, in order. It then attempts to produce a predicted price. The prediction it makes is compared to the actual value of the dependent variable from the training set and used to calculate a mean squared error, which is then back propagated through the network to adjust weights with the goal of setting weights in a manner that will produce more accurate predictions. The network ran through all of the training data 100 times over.

Once training is over, the testing set (all dependent variables not included in the training set) can be given to the neural network in order. At this point, since the network is no longer training, the weights will no longer adjust; whatever the weights become at the end of the training period are the permanent weights that will be used to make predictions on the test set. This produces an array of predicted prices, which can then be used to calculate the profits of the long short term memory neural network. To calculate the profits, the program compares the predicted price to the actual price from the year before the prediction, just like in the linear regression. If the predicted price was an increase, the program would buy a share. If the predicted price was a decrease, the program would sell a share.

Results

Human Experiment

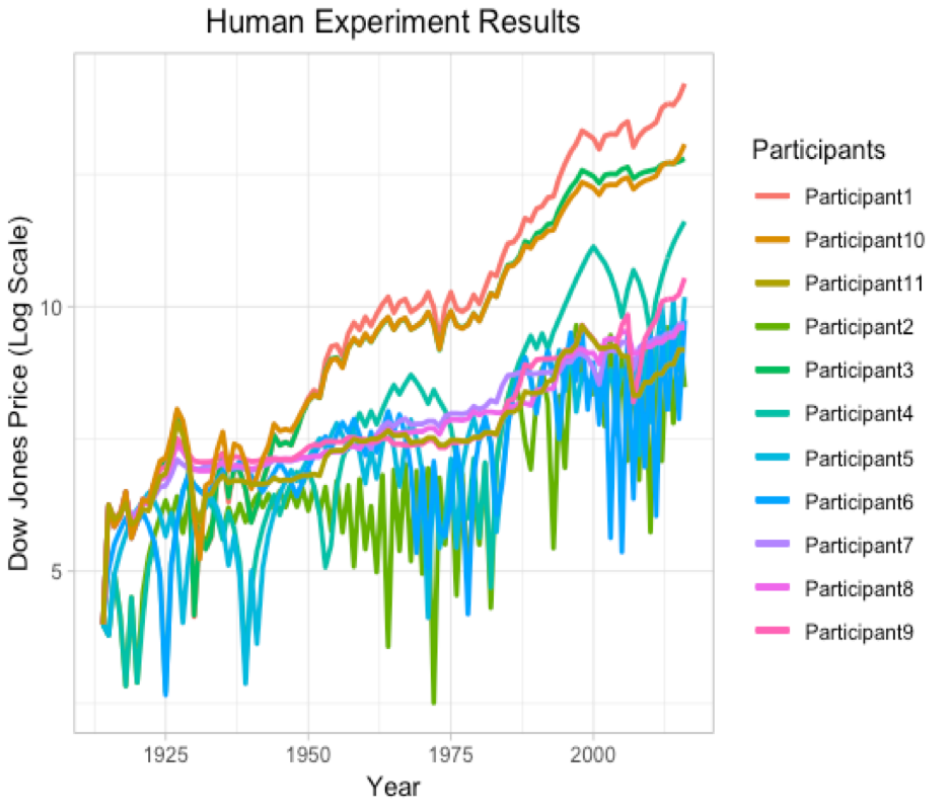

FIGURE 10. Profits of each human participant in the human portion of this study. *Most participants were in a similar range of profitability, however there were 3 outliers. Participant 1 was the most profitable, with participants 3 and 10 also outperformed the other participants by a wide margin.*

As shown in Fig. 10, the results of the various human participants were varied. Participants 1, 3, and 10 outperformed the other participants by quite a lot. This skews the average results of the participants to be more profitable than the majority of the participants were, so when looking to compare the profits of the participants to the profits of the other algorithms, it is best to look at the median profits of the participants. The lowest performing human participant ended with a profit of $4,782.92 USD, and the most successful human participant ended with a profit of $1,486,325.82 USD.

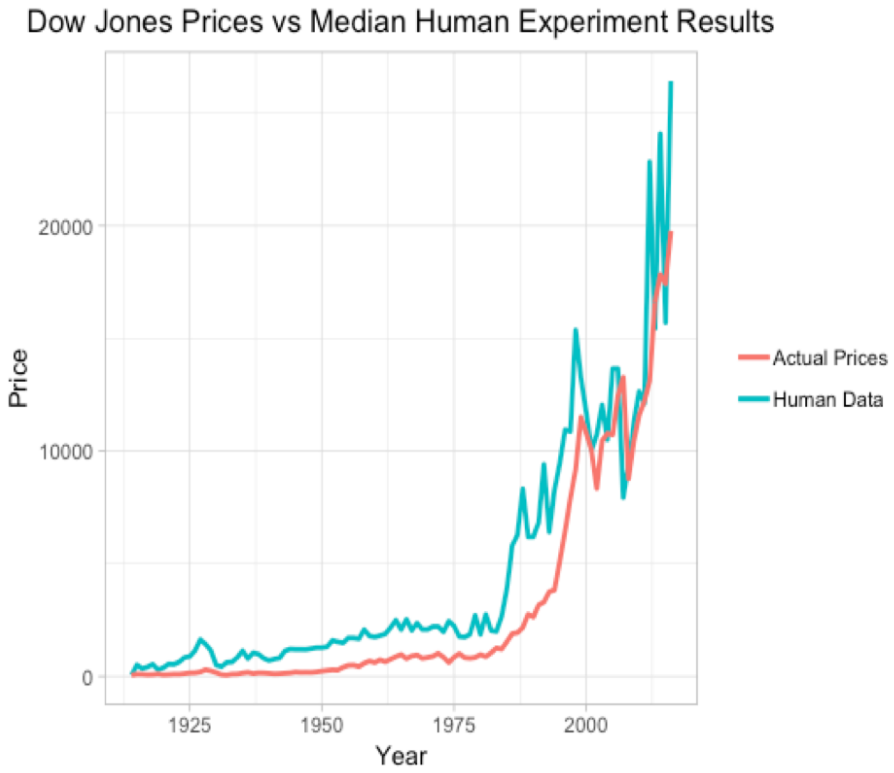

FIGURE 11. Median profits of the human participants compared to the actual prices of the Dow Jones over the same period of time. *The median human profits can be seen in teal and the red line show the actual price of the Dow Jones over the same period of time. Humans were able to outperform the price increases of the Dow Jones.*

Looking at the median prices of the human participants provides a better picture of how the humans actually performed. As shown in Fig. 11, the median results of humans were only slightly better than the actual performance of the Dow Jones over the same period of time, and the rise and falls of both were fairly similar. The median human profits ended with a profit of $26,411.72 USD.

Linear Regression

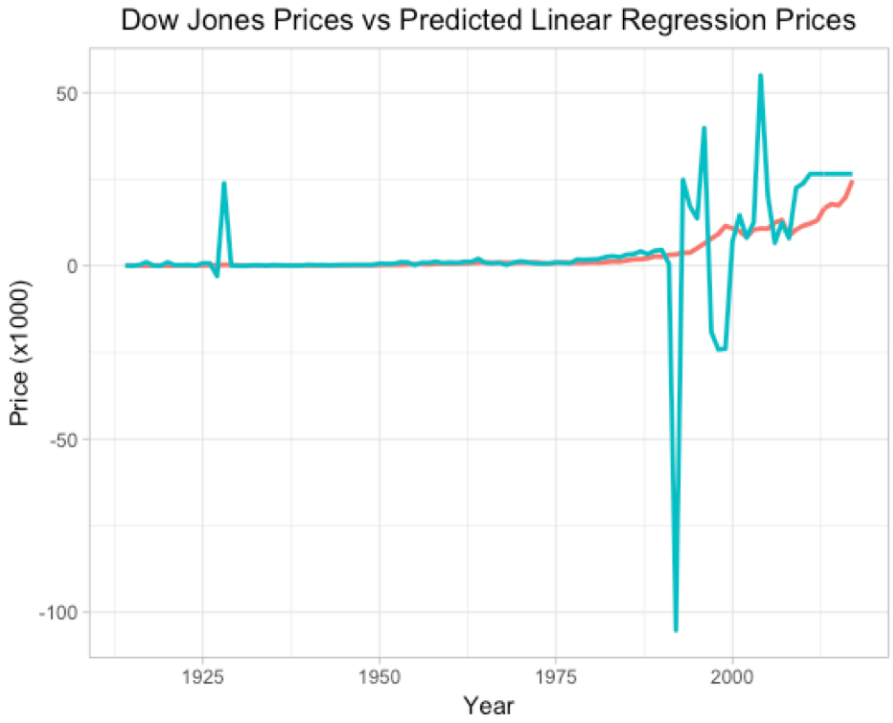

FIGURE 12. Actual price of the Dow Jones versus the prices predicted for the Dow Jones by Linear Regression over the same period of time. *The predicted prices are represented in teal, and the actual prices of the Dow Jones are shown in red. Linear regression was not effective at predicting prices when there was volatility in the market. When the market was stable, however, linear regression was able to closely predict the prices of the Dow Jones.*

Linear Regression was able to predict the prices of the Dow Jones reasonably well when there were no sharp rises or falls in the market; however, when there were, linear regression either vastly over or under predicted the price of the market. However, even with these inaccurate predictions mixed in, linear regression was still profitable.

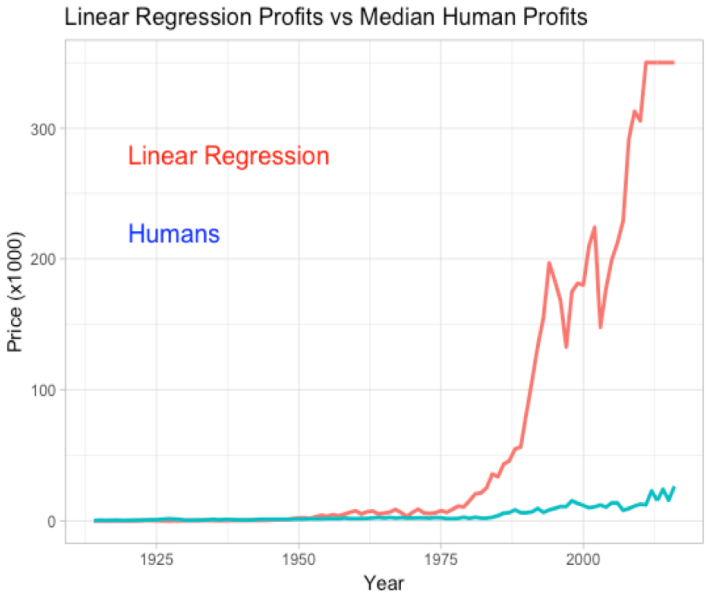

FIGURE 13. Profits of the linear regression program compared to the median profits of the human experiment. *Linear regression profits are shown in red and the median human profits are shown in teal. Linear regression was found to vastly outperform the median human results.*

As shown in Fig. 13, even with the over and under predictions, linear regression was still able to greatly outperform the median profits of the human participants. The linear regression ended with a total profit of $350,112.58 USD.

Reinforcement Learning

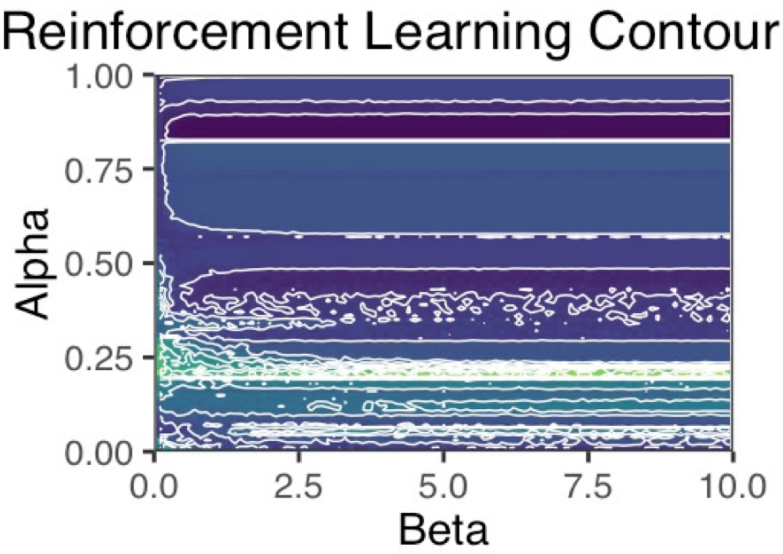

FIGURE 14. Contour plot showing the average profits of each alpha and beta combination. *The lighter the color, the higher the profits. The dark purple sections are the least profitable, and light-yellow sections are the most profitable. The most profitable combination was an alpha of 0.27 and a beta of 6.3.*

Fig. 14 shows the vast disparities between different combinations of alpha and beta. As shown, most alphas lower than 0.2 and higher than 0.3 were not very profitable. Most combinations of beta within this range of alpha, however, were all profitable.

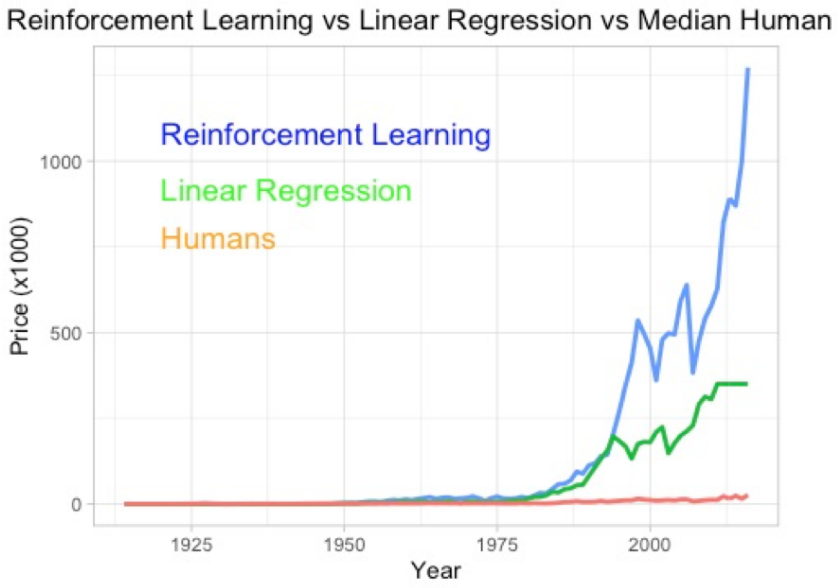

FIGURE 15. Profits of reinforcement learning (alpha = 0.27, beta = 6.3) compared to the linear regression profits and the median human profits. *Reinforcement learning was by far the most profitable of the three, followed by linear regression, with the median human profits being the least profitable.*

The best combination of alpha and beta, 0.27 and 6.3, were found to be far more profitable than both the linear regression program and the median human participants. Reinforcement learning with this combination of alpha and beta ended with a total profit of $1,272,594.52 USD.

Neural Network

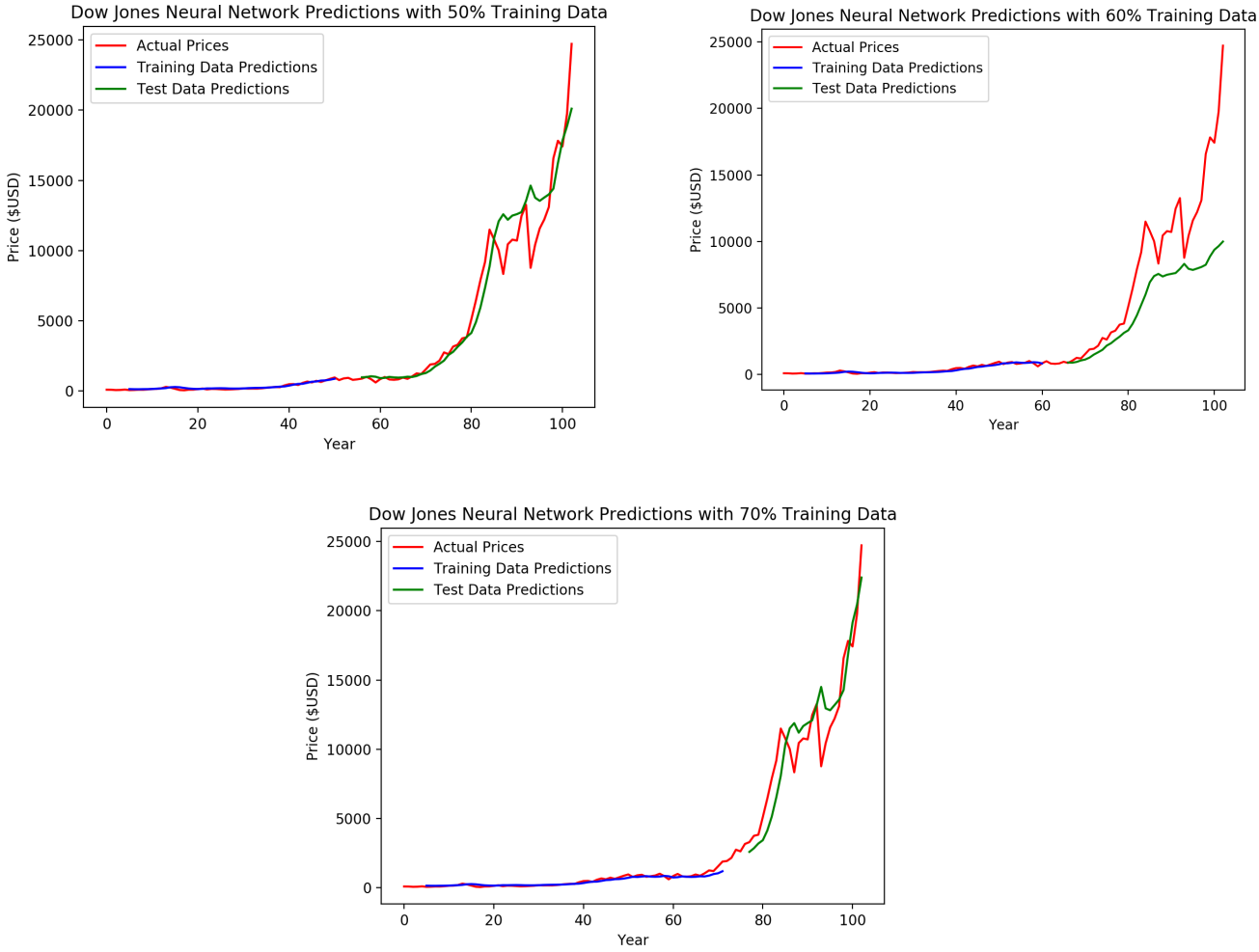

FIGURE 16. LSTM Neural Network predictions of testing and training data versus the actual prices of the Dow Jones with 50%, 60%, and 70% of all data as training data. *50% training data and 70% training data had the best predictions of the three, with 50% being the best of the two. The 60% training data set predictions were all under what the actual prices were, making it an ineffective model for profitability.*

Prediction quality of the neural networks was varied. Neural Network theory suggests that more training data produces better predictions (Nielsen, 2015), but as shown in Fig. 16, the

predictions created with 60% of the training data were less accurate than those with 50% of the data. Training with 50% of the data also produced more accurate testing data predictions than the 70% training data set for all but the last 10 years, however this can be attributed to the longer length of the testing prediction set due to less training data. Overall, both the 50% training data set, and the 70% training data set were able to closely predict the prices of the Dow Jones.

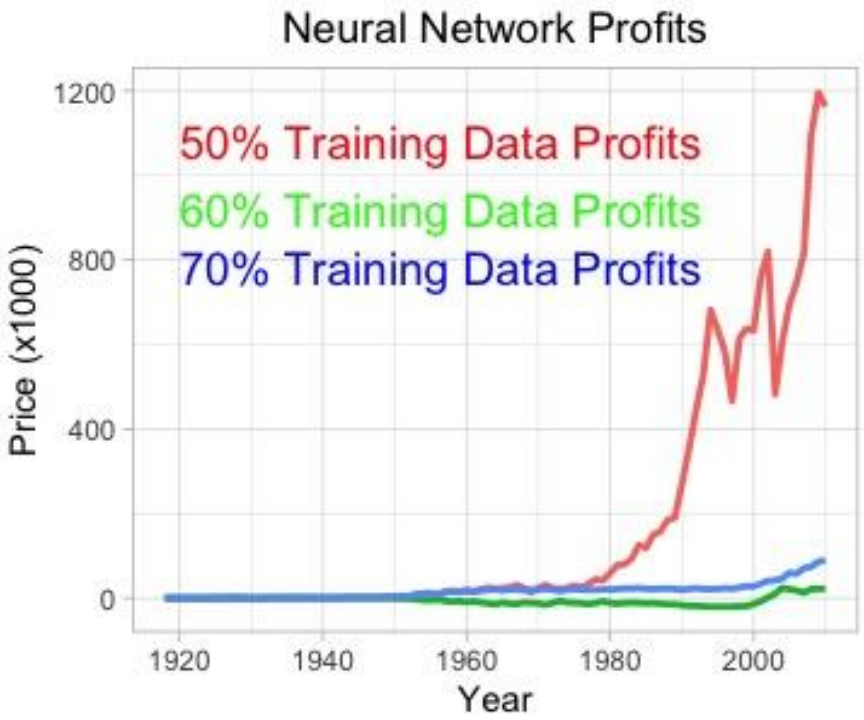

FIGURE 17. Comparison of the profits generated by the LSTM Neural Network with 50% training data, 60% training data, and 70% training data. *The 50% training data set is represented in red; the 60% training data set is represented in green, and the 70% training data set is represented in blue. 50% training data was able to make the largest profit of the three.*

While the 70% training data set did have the best predictions of prices, the 50% neural network was able to vastly outperform both other training data sets in terms of profit. Because the 70% training data neural network had a smaller test data size to make predictions on, it missed out on predicting the years 1979-1993. During those years, the Dow Jones experienced a 22.34% increase, which gave the 50% training data a large advantage in accumulating profit, due to having a longer time frame to make predictions on, thus a longer time to generate a profit. The 50% training data set was able to end with a total profit of $1,163,146.20 USD, while the 60% and 70% training data sets had profits of $21,883.02 USD and $88,555.03 USD respectively.

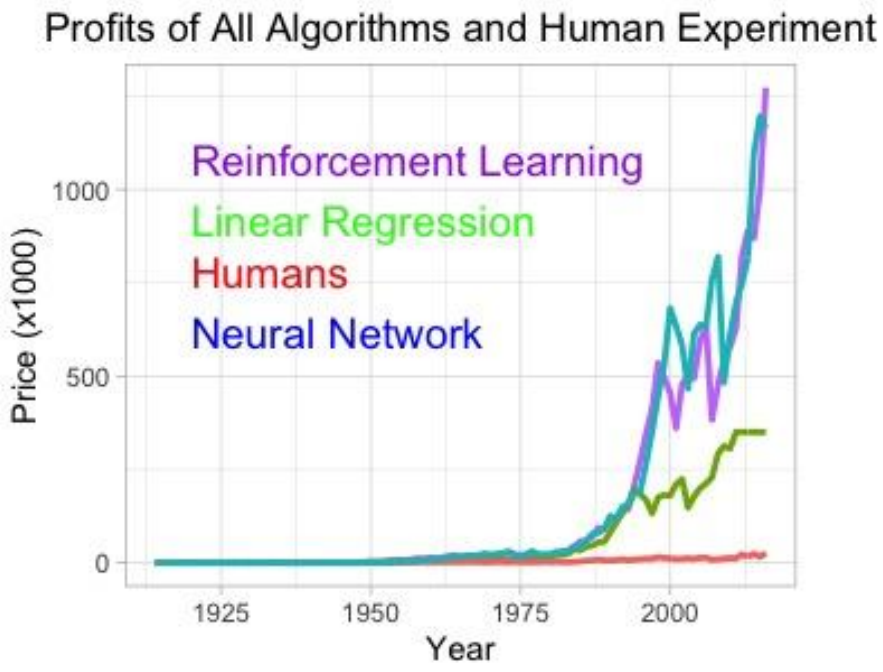

FIGURE 18. Comparison of the profits of all algorithms and the profits of the median results of the human experiment. *Reinforcement learning profits are represented in purple, neural network profits in blue, linear regression profits in green, and humans in red. Reinforcement learning was the most profitable model, closely followed by the neural network.*

The results of all algorithms and all human participants show that they were all able to turn a profit trading on the stock market, with none of them ending up with a negative balance. One thing to note about this result is that this study covered a large period of time, 103 years, during which the Dow Jones itself experienced 22,478.38% growth. The overall massive growth experienced by the market as a whole makes it near impossible for even the naivest trader or algorithm to not turn at least some profit over this period of time. This also makes evaluating the effectiveness of any of these algorithms as an effective tool for actual day to day trading on the stock market difficult. The researchers can, however, accurately compare each algorithm to each other and the human participants to see which of them is the best at learning over long periods of time.

When comparing all of the algorithms and the median human results, Reinforcement learning preformed the best. Reinforcement learning generated a total profit of $1,272,594.52, outperforming the second-best algorithm, the 50% training data neural network, by $109,448.32 (8.99%). If the highest-performing human participant is used for comparison instead of the median, however, this does not hold true. The highest-performing human participant was able to outperform reinforcement learning by $213,731.30 (15.49%), but none of the other 10 participants were able to outperform either reinforcement learning or the neural network.

Discussion

The results of this study suggest that while these algorithms are psychologically plausible and based in theories of learning in the brain, humans do not pick up this information as effectively as computer programs do. This is most likely caused by multiple reasons. Most importantly, the algorithms all had a specifically defined method they had to follow, while humans were able to

choose their actions freely. Computer programs can also make calculations faster than humans can, and they cannot make errors in their calculations, assuming they are programmed without errors. This increase in speed and lack of error can lead to picking up the best strategies faster than humans can.

This study contains multiple limitations. In the interest of giving each algorithm the same exact data the human experiment had, each algorithm only received 103 data points. Neural networks typically use at minimum 10,000 data points and typically use far more than that (Nielsen, 2015). In addition to this, neural networks on the stock market typically use many more metrics as independent variables besides just the prices of the last five years; the more independent variables that play a role in a stock price the better (Baba, & Kozaki, 1992; Chenoweth, & Obradović, 1996; Tu, 1996; Walczak, 1999; Fernández-Rodríguez, González-Martel, & Sosvilla-Rivero, 2000; Leigh, Purvis, & Ragusa, 2002; Ghiassi, Saidane, & Zimbra, 2005; Hamzaçebi, Akay, & Kutay, 2009; Ticknor, 2013; Kim, & Kim, 2019). By using such a small data set and a lack of many independent variables, the effectiveness of the neural network was limited. The lack of a large data size is also an issue for the reinforcement learning, as reinforcement learning needs ample time to properly converge to the mathematically best actions.

Future research on this topic should look into using larger data sets and more variables in order to more accurately evaluate the practical effectiveness of using these algorithms in the real world. The number of human participants used in this study was also fairly small due to difficulties finding university students willing to sit down and execute 103 trades in a row. Finding a larger population of humans to participate in a future study would be highly beneficial to obtaining clearer results for human participants. Finding people with experience in trading and

comparing them to the algorithms and to participants without trading experience would also add an interesting layer to this study.

References

Arbib, M. A. (2003). *The handbook of brain theory and neural networks*. Cambridge, MA: MIT Press.

Baba, N., & Kozaki, M. (1992). An intelligent forecasting system of stock price using neural networks. *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*. doi:10.1109/ijcnn.1992.287183

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks,5*(2), 157-166. doi:10.1109/72.279181

Bremer, M. (2012). Multiple Linear Regression. Retrieved from http://mezeylab.cb.bscb.cornell.edu/labmembers/documents/supplement%205%20-%20multiple%20regression.pdf

Brezak, D., Bacek, T., Majetic, D., Kasac, J., & Novakovic, B. (2012). A comparison of feed-forward and recurrent neural networks in time series forecasting. *2012 IEEE Conference on Computational Intelligence for Financial Engineering & Economics (CIFEr)*. doi:10.1109/cifer.2012.6327793

Chaboud, A., Hjalmarsson, E., Vega, C., & Chiquoine, B. (2011). Rise of the Machines: Algorithmic Trading in the Foreign Exchange Market. *SSRN Electronic Journal*. doi:10.2139/ssrn.1501135

Chenoweth, T., & Obradović, Z. (1996). A multi-component nonlinear prediction system for the S&P 500 index. *Neurocomputing,10*(3), 275-290. doi:10.1016/0925-2312(95)00109-3

Dash, R., & Dash, P. K. (2016). A hybrid stock trading framework integrating technical analysis with machine learning techniques. *The Journal of Finance and Data Science, 2*(1), 42-57. doi:10.1016/j.jfds.2016.03.002

Fernández-Rodríguez, F., González-Martel, C., & Sosvilla-Rivero, S. (2000). On the profitability of technical trading rules based on artificial neural networks:. *Economics Letters,69*(1), 89-94. doi:10.1016/s0165-1765(00)00270-6

Ghiassi, M., Saidane, H., & Zimbra, D. (2005). A dynamic artificial neural network model for forecasting time series events. *International Journal of Forecasting,21*(2), 341-362. doi:10.1016/j.ijforecast.2004.10.008

Glimcher, P. W. (2011). "Understanding dopamine and reinforcement learning: The dopamine reward prediction error hypothesis": Correction. *PNAS Proceedings of the National Academy of Sciences of the United States of America, 108*(42), 42.

Hamzaçebi, C., Akay, D., & Kutay, F. (2009). Comparison of direct and iterative artificial neural network forecast approaches in multi-periodic time series forecasting. *Expert Systems with Applications,36*(2), 3839-3844. doi:10.1016/j.eswa.2008.02.042

Hinton, G.E. (1992). How neural networks learn from experience. *Scientific American*, *236,* 145-151

Hochreiter, J., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation, 9(8)*, 1735-1780. doi:10.1162/neco.1997.9.8.1735

Karlik, B., & Olgac, A.V. (2010). Performance Analysis of Various Activation Functions in Generalized MLP Architectures of Neural Network

s. *International Journal of Artificial*

Intelligence and Expert Systems,1(4), 111-122

Khaldi, R., Chiheb, R., & Afia, A. E. (2018). Feedforward and Recurrent Neural Networks for Time Series Forecasting. *Proceedings of the International Conference on Learning and Optimization Algorithms: Theory and Applications - LOPAL 18*. doi:10.1145/3230905.3230946

Kim, T., & Kim, H. Y. (2019). Forecasting stock prices with a feature fusion LSTM-CNN model using different representations of the same data. *Plos One,14*(2). doi:10.1371/journal.pone.0212320

Lee, J. W., Park, J., O, J., Lee, J., & Hong, E. (2007). A Multiagent Approach to Q-Learning for Daily Stock Trading. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans,37*(6), 864-877. doi:10.1109/tsmca.2007.904825

Lee, J., Hong, E., & Park, J. (2004). A Q-learning based approach to design of intelligent stock trading agents. *2004 IEEE International Engineering Management Conference (IEEE Cat. No.04CH37574)*. doi:10.1109/iemc.2004.1408902

Lee, J. W., & O, J. (2002). A Multi-agent Q-learning Framework for Optimizing Stock Trading Systems. *Lecture Notes in Computer Science Database and Expert Systems Applications,*153-162. doi:10.1007/3-540-46146-9_16

Lee, J. W. (2001). Stock price prediction using reinforcement learning. *ISIE 2001. 2001 IEEE International Symposium on Industrial Electronics Proceedings (Cat. No.01TH8570)*. doi:10.1109/isie.2001.931880

Lehmann, E. L., & Casella, G. (2013). *Theory of point estimation*. New York: Springer.

Leigh, W., Purvis, R., & Ragusa, J. M. (2002). Forecasting the NYSE composite index with technical analysis, pattern recognizer, neural network, and genetic algorithm: A case study in romantic decision support. *Decision Support Systems,32*(4), 361-377. doi:10.1016/s0167-9236(01)00121-x

Liestbl, K., Andersen, P. K., & Andersen, U. (1994). Survival analysis and neural nets. *Statistics in Medicine,13*(12), 1189-1200. doi:10.1002/sim.4780131202

Mikolov, T., Karafiat, M., Burget, L., Cernocky, J.H., & Khudanpur, S. (2010) Recurrent Neural Network Based Language Model. *International Speech Communication Association*

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press.

Pascanu, R., Mikolov, T. & Bengio Y. (2013). On the difficulty of training recurrent neural networks. *ICML '13 Proceedings of the 30th International Conference on International Conference on Machine Learning 28*(3), 1310-1318.

Rescorla, R. A. & Wagner, A. R. (1972). A theory of Pavlovian conditioning: Variations on the effectiveness of reinforcement and non-reinforcement. In A. H. Black & W. F. Prokasy (ed.), *Classical conditioning II: Current research and theory* (pp. 64-99) . Appleton-Century-Crofts.

Schultz, W., Dayan, P., & Montague, P. R. (1997). A neural substrate of prediction and reward. *Science, 275*(5306), 1593-1599.

Sutton, R. S. (1998). *Reinforcement learning*. Boston: Kluwer Academic.

Ticknor, J. L. (2013). A Bayesian regularized artificial neural network for stock market forecasting. *Expert Systems with Applications,40*(14), 5501-5506. doi:10.1016/j.eswa.2013.04.013

Tu, J. V. (1996). Advantages and disadvantages of using artificial neural networks versus logistic regression for predicting medical outcomes. *Journal of Clinical Epidemiology,49*(11), 1225-1231. doi:10.1016/s0895-4356(96)00002-9

Walczak, S. (1999). Gaining Competitive Advantage for Trading in Emerging Capital Markets with Neural Networks. *Journal of Management Information Systems,16*(2), 177-192. doi:10.1080/07421222.1999.11518251

Watkins, C. J. C. H., Dayan, P. (1992). Q-learning. *Machine Learning,* 8(3-4):279–292.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards.* Ph.D. thesis, University of Cambridge.

Appendix A - Human Experiment Code

All source code as well as the data used in this study can be downloaded at

https://github.com/chudleyj/thesis

experiment.html

experiment.html 〉 <head>

```
<!DOCTYPE html>
<html style ="background-color:black; color:white;">
    <head>
        <title>My experiment</title>
        <script src="jspsych-6.0.5/jspsych.js"></script>
        <script src="jspsych-6.0.5/plugins/jspsych-html-keyboard-response.js"></script>
        <link href="jspsych-6.0.5/css/jspsych.css" rel="stylesheet" type="text/css"></link>

        <script src="jspsych-6.0.5/plugins/jspsych-fullscreen.js"></script>

        <link href="experiment.css" rel="stylesheet" type="text/css">
        <script src="functions.js"></script>
    </head>
    <body>
    </body>
</html>
```

```
let currTrial = 0

let shareBought = false

let balanceChange = 0

let balanceChangeStr = "null"

/*
  let choice keep tracks of which combination of buy, sell, and hold options
  are legal for a user during a certian trial
*/
let choice = 3

document.addEventListener('DOMContentLoaded', function ()
{
  const nTrials = prices.length-1

  fullscreen = {type: 'fullscreen', fullscreen_mode: true}

  timeline = []

  timeline.push(fullscreen)
  timeline.push(welcomeScreen)
  timeline.push(insctructionsScreen)
  timeline.push(accInfoIntroScreen)
  timeline.push(priceDateScreen)
  timeline.push(buySellHoldScreen)

  for(let i = currTrial; i < nTrials; i++)
  {
    timeline.push(accInfoScreen)
    timeline.push(priceDateScreen)
    timeline.push(buySellHoldScreen)
  }

  jsPsych.init({
      timeline: timeline,
      on_finish: function(data){jsPsych.data.get().localSave('csv', `data.csv`)}
  })

});


/*
  Key press values:
   Space: 32
   F: 70
   J: 74
*/
function updateBalance()
{
  const trialData = jsPsych.data.getLastTrialData()
```

```
    //Hold
    if(trialData.values()[0].key_press == 32){
      return balance
    }

    //Buy
    else if(trialData.values()[0].key_press == 70){
      bal = bal - prices[trialData.values()[0].trial_number-1]
      balance = bal + (prices[trialData.values()[0].trial_number] * sharesOwned)
      balanceChange = prices[trialData.values()[0].trial_number] -
       prices[trialData.values()[0].trial_number-1]
      balanceChangeStr = balanceChange.toLocaleString()
      if(balanceChange > 0)
        balanceChangeStr = "+" + balanceChangeStr
      return balance
    }

    //Sell
    else if(trialData.values()[0].key_press == 74){
      bal = bal + prices[trialData.values()[0].trial_number-1]
      balance = bal + (prices[trialData.values()[0].trial_number] * sharesOwned)
      balanceChange = prices[trialData.values()[0].trial_number] -
       prices[trialData.values()[0].trial_number-1]
      balanceChangeStr = balanceChange.toLocaleString()
      if(balanceChange > 0)
        balanceChangeStr = "+" + balanceChangeStr
      return balance
    }

  }

  function updateShares()
  {
    const trialData = jsPsych.data.getLastTrialData()

    //Hold
    if(trialData.values()[0].key_press == 32){
      return
    }

    //Buy
    else if(trialData.values()[0].key_press == 70){
      sharesOwned = sharesOwned + 1
      shareBought = true
    }

    //Sell
    else if(trialData.values()[0].key_press == 74){
      sharesOwned =  sharesOwned - 1
      shareBought = false
    }
  }
```

```
let accInfoIntroScreen =
{
  type: 'html-keyboard-response',
  response_ends_trial: true,
  choices: jsPsych.NO_KEYS,
  trial_duration: 4000,
  stimulus: function()
  {
    return `<p>Account Balance: $${balance}</p>
            <p>Shares Owned: ${sharesOwned}</p>`;
  }
};

let accInfoScreen =
{
  type: 'html-keyboard-response',
  response_ends_trial: true,
  choices: jsPsych.NO_KEYS,
  trial_duration: 4000,
  stimulus: function()
  {
      if(shareBought){
        if(balanceChange > 0){
          return `<div id="ParentDiv">
                    <div id="Balance" style="float:left;">
                      <p>Total Assets: <strong>$${balance.toLocaleString()}</strong>
                      <font color = "#00cc00"> <strong> ${balanceChangeStr}</
                       strong></font></p>
                    </div>
                    <br>
                    <div id = "Shares" style="float:left;">
                      <p>  Shares Owned: <strong>${sharesOwned}</strong>
                      <font color = "#00cc00"><strong>+1</strong></font></p>
                    </div>
                  </div>`;
        }
        else{
          return `<div id="ParentDiv">
                    <div id="Balance" style="float:left;">
                      <p>Total Assets: <strong>$${balance.toLocaleString()}</strong>
                      <font color = "#ff0000"> <strong> ${balanceChangeStr}</
                       strong></font></p>
                    </div>
                    <br>
                    <div id = "Shares" style="float:left;">
                      <p>  Shares Owned: <strong>${sharesOwned}</strong>
                      <font color = "#00cc00"><strong>+1</strong></font></p>
                    </div>
                  </div>`;
        }
      }
      else{
        if(balanceChange > 0){
          return `<div id="ParentDiv">
```

```
                    <div id="Balance" style="float:left;">
                      <p>Account Balance: <strong>$${balance.toLocaleString()}</
                       strong>
                      <font color = "#00cc00"> <strong> ${balanceChangeStr}</
                       strong></font></p>
                    </div>
                    <div id="Shares" style="float:left;">
                      <p>  Shares Owned: <strong>${sharesOwned}</strong>
                      <font color = "#ff0000">  <strong>-1</strong></font></p>
                    </div>
                  </div>`;
      }
        else{
          return `<div id="ParentDiv">
                    <div id="Balance" style="float:left;">
                      <p>Account Balance: <strong>$${balance.toLocaleString()}</
                       strong>
                      <font color = "#ff0000"> <strong> ${balanceChangeStr}</
                       strong></font></p>
                    </div>
                    <div id="Shares" style="float:left;">
                      <p>  Shares Owned: <strong>${sharesOwned}</strong>
                      <font color = "#ff0000">  <strong>-1</strong></font></p>
                    </div>
                  </div>`;
      }
    }
  }
};

let buySellHoldScreen =
{
  type: 'html-keyboard-response',
  response_ends_trial: true,
  choices: ["f", "j"],
  data: function(){return ({trial: 'buySellHold', trial_number: currTrial, date:
   dates[currTrial], price: prices[currTrial]})},
  stimulus: function()
  {
      return `<div id ="ParentDiv" style = "align-text:center; width:100%;">
                <p>Would you like to buy or sell a share for: $$
                 {prices[currTrial-1]}</p>
                <div id="Buy" style ="position: relative; display: inline-block;">
                  <p>Buy</p>
                  <p>( <strong>F</strong> )</p>
                </div>
                <div id="Sell" style ="position: relative; display: inline-block;">
                  <p>  Sell</p>
                  <p>  ( <strong>J</strong> )</p>
                </div>
              </div>`;
  },
  on_finish: function(trial_data)
  {
```

```
    updateShares()
    updateBalance()
    jsPsych.data.addDataToLastTrial
    ({
      choice: trial_data.key_press ==
       jsPsych.pluginAPI.convertKeyCharacterToKeyCode('F') ? "buy" : "sell",
      balance: balance,
      shares: sharesOwned
    })
  }
};

let insctructionsScreen =
{
  type: 'html-keyboard-response',
  response_ends_trial: true,
  choices:["space"],
  stimulus: `<p class='very-large center-content'>During this experiment you will be
   buying and selling shares on a simulated stock market.</p>
             <p>You will start with $10,000 and 5 shares.</p>
             <p>You will be shown dates and the price of the stock on that date. You
              will then either buy or sell a share on that date.</p>
             <p>Place your left index finger on the 'F' key and your right index
              finger on the 'J' key and leave them there for the entire
              experiment.</p>
             <p>Press the 'Space Key' to begin.</p>`,
  timing_post_trial: 0
}

let priceDateScreen =
{
  type: 'html-keyboard-response',
  response_ends_trial: true,
  choices:["space"],
  stimulus: function()
    {
      currTrial++;
      return `<br><br><br><br><br><br><br><br>
              <p>Date: ${dates[currTrial-1]}</p>
              <p>Price: $${prices[currTrial-1]}</p>
              <br><br><br><br><br><br><br><br><br><br><br><br>
              <p style="display: table-row; vertical-align: bottom; text-align:
               center">Press <strong>Space</strong> to continue</p>`;
    }
}

let welcomeScreen =
{
    type: 'html-keyboard-response',
    stimulus: `<p class='very-large center-content'>Welcome.</p>
               <p class='center-content'>Wait until you are told to begin, then hit
                any key for the insctructions</p>`,
    response_ends_trial: true,
    timing_post_trial: 0
```

Appendix B - Linear Regression Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm

class portfolio:
    def __init__(self, bank = 10000, stocksOwned = 0, profits = 0):
        self.bank = bank
        self.stocksOwned = stocksOwned
        self.assets = bank
        self.profits = 0

    def executeAction(self, action, price):
        if action: #Buy
            self.stocksOwned += 1
            self.bank -= price
            self.assets = self.bank + (price * self.stocksOwned)

        else: #Sell
            self.stocksOwned -= 1
            self.bank  += price
            self.assets = self.bank + (price * self.stocksOwned)

    def calcProfits(self):
        self.profits = self.assets-10000
        self.profits = round(self.profits,2)
        self.profits = str(self.profits)
        self.profits += "\n"

def chooseAction(yPoly, y): #Returns true if predicted value n is an increase over y
 @ n-1, else returns false
    return yPoly > y

def fileWrite100(predictedPrices, profits): #Write the data for the smaller data set
 to a file
    file = open("100predictions.txt", "a")

    for i in predictedPrices:
        file.write(str(i)+"\n")

    file.close()

    file = open("100profits.txt", "a")

    for j in profits:
        file.write(str(j))

    file.close()

def fileWriteFullDataSet(predictedPrices, profits): #Write the data for the full
 data set to a file
    file = open("predictions.txt", "a")

    for i in predictedPrices:
```

```
            file.write(str(i)+"\n")

        file.close()

        file = open("profits.txt", "a")

        for j in profits:
            file.write(str(j))

        file.close()

    def DataSetRegression(X, y, dataSize): #Preform a linear regression on the data
        profList = []
        pfolio = portfolio()
        predictedPrices = []

        X = sm.add_constant(X) # adding a constant
        for n in range (5, dataSize, 1):
            model = sm.OLS(y[n-5:n], X[n-5:n]).fit()
            prediction = model.predict(X[n+1: n+2]).values
            action = chooseAction(prediction, y[n])
            pfolio.executeAction(action, y[n])
            predictedPrices.append(prediction)
            pfolio.calcProfits()
            profList.append(pfolio.profits)
            print(y[n], y[n+1],predictedPrices[n-5], prediction, action, pfolio.assets,
             pfolio.profits)

        fileWrite100(predictedPrices, profList) if dataSize == 103 else
         fileWriteFullDataSet(predictedPrices,profList)
        print (pfolio.assets, pfolio.profits)
        #plotData(predictedPrices, y)



    def loadData(fileName):
        col_names = ['y', 'x1','x2','x3','x4','x5']
        df = pd.read_csv(fileName, header=None, names=col_names)
        y = df['y']
        X=df[['x1','x2','x3','x4','x5']]
        return X,y

    def plotData(predictedPrices, y):
        print (predictedPrices)
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.plot(predictedPrices, label ='Predicted Prices')
        #ax.plot(y, label = 'Actual Prices')
        plt.legend(loc=2)
        plt.show()

    X,y = loadData("DJI.csv")
    DataSetRegression(X,y,103)
    X2,y2 = loadData("DJI2.csv")


                                DataSetRegression(X2,y2,27808)
```

Appendix C - Reinforcement Learning Code

```
#include <iostream>
#include <fstream>
#include <vector>
#include <stdlib.h>
#include <random>
#include <set>
#include <iterator>
#include <functional>
#include <algorithm>
#include <initializer_list>
#include <math.h> //Euler's number
#include <cmath>
#include <cstdlib>
#include <random>
#include <stdlib.h>     /* atoi */

const int _DATA_SIZE = 104;
const std::string _FILE_PATH = "DJI.csv";

struct portfolioData
{
    double previousAssets = 0; //Gets over written right away
    double currentAssets = 10273.1;//10,000 + (5*54.62)
    double currentBank = 10000;
    int currentStocksOwned = 5;
};

struct RLdata
{
  const double alpha;
  const double beta;
  bool action; //0  = BUY, 1 = SELL
  double Qbuy = 0;
  double Qsell = 0;
  RLdata(const double &a, const double &b) : alpha(a), beta(b){}
};

std::vector<double> get_CSV_data();
double softMax(const RLdata &);
void chooseAction(RLdata &, const double &, portfolioData &, const
 std::vector<double>, const int);
double random_double();
void buy(portfolioData &, RLdata &, const std::vector<double> &, const int &);
void sell(portfolioData &, RLdata &, const std::vector<double>, const int);
void updateQvalues(RLdata &, const portfolioData &);

void ToFile(const double &, const std::vector<double> &, const
 std::vector<double> &,const std::vector<bool> &,const double &, const double &);
void ToFile2(const std::vector<double> &, std::string);
void ToFile3(const double &, const double &, const double &, const std::string &);
```

```
#include "RL.h"

int main(int argc, char **argv)
{
  std::cout << "Called" << std::endl;

  portfolioData portfolio; //Start with 'X' (See RL.h) dollars
  RLdata RL(atof(argv[1]), atof(argv[2]));

  std::vector<bool> actions; //write actions to here for file output.
  std::vector<double> QS; //write Qsell to here for file output.
  std::vector<double> QB; //write Qbuy to here for file output.
  std::vector<double>profVec;


  //Fill in prices from CSV file
  const std::vector<double> historicalPrices = get_CSV_data();
  double ProbabilityOfBuy = 0.0;

  for(int t = 0; t < historicalPrices.size()-1; t++)
  {//Pass t
    //Start the RL by choosing an action
    ProbabilityOfBuy = softMax(RL);

      // Choose action, and update assets a year later.
    chooseAction(RL,ProbabilityOfBuy,portfolio,historicalPrices,t);

    //Action was taken, reward has been given, now update Q
    updateQvalues(RL, portfolio);

    //Write values to vectors for file output
    actions.push_back(RL.action); //Write action to vec for output at end
    QS.push_back(RL.Qsell);
    QB.push_back(RL.Qbuy);
    profVec.push_back(portfolio.currentAssets - 10273.1);
    ProbabilityOfBuy = 0.0;
   }
  //RL is over, sell any left overs to calc total profits.
  std::cout << "Profit: " << portfolio.currentAssets - 10273.1;
 // ToFile((portfolio.currentAssets - 10273.1), QS, QB, actions, RL.alpha, RL.beta);
  ToFile2(profVec, argv[3]);
  ToFile3(portfolio.currentAssets-10273.1, RL.alpha, RL.beta, argv[3]);
}

std::vector<double> get_CSV_data()
{
    std::ifstream ip(_FILE_PATH);

    if(!ip.is_open())
        std::cout << "Error opening file." << std::endl;

    std::vector<double> vec;
    std::string temp;
    while(std::getline(ip, temp))
```

```
        vec.push_back(atof(temp.c_str()));

    ip.close();
    return vec;
}

double softMax(const RLdata &RL)
{
    //1/(1+e^(Q(action)-Q(otherAction)*Beta)
    return(1/(1+exp((RL.Qsell-RL.Qbuy)*RL.beta)));
}

void chooseAction(RLdata &RL, const double &Pbuy, portfolioData &env, const
 std::vector<double> price, const int count)
{
    auto p = random_double();
    RL.action = p < Pbuy;
    //std::cout <<"rand: " << p << " prob: " << Pbuy << " action: " <<  RL.action <<
     " QB: " << RL.Qbuy << " QS: " << RL.Qsell << std::endl;

    if(RL.action || env.currentStocksOwned < 0)
        buy(env, RL, price, count);
    else
        sell(env, RL, price, count);
}

double random_double()
{
    std::random_device rd;
    std::mt19937 e2(rd());
    std::uniform_real_distribution<> dist(0, 1);
    return dist(e2);
}

void buy(portfolioData &env, RLdata &RL, const std::vector<double> &price, const int
 &count)
{
    env.previousAssets = env.currentAssets;
    env.currentStocksOwned++;
    env.currentBank = env.currentBank - price[count];
    env.currentAssets = env.currentBank + (env.currentStocksOwned * price[count+1]);
    //std::cout <<" Buy prev " <<  env.previousAssets << " curr " <<
     env.currentAssets << " stock " << env.currentStocksOwned << " bank "<<
     env.currentBank
    //<< " price " <<price[count] << " price +1 " << price[count+1] << std::endl;
}

void sell(portfolioData &env, RLdata &RL, const std::vector<double> price, const int
 count)
{
    env.previousAssets = env.currentAssets;
    env.currentStocksOwned--;
    env.currentBank = env.currentBank + price[count];
    env.currentAssets = env.currentBank + (env.currentStocksOwned * price[count+1]);
```

```
   // std::cout <<"Sell prev " <<  env.previousAssets << " curr " <<
    env.currentAssets << " stock " << env.currentStocksOwned << " bank "<<
    env.currentBank
   // << " price " <<price[count] << " price +1 " << price[count+1] << std::endl;
}

void updateQvalues(RLdata &RL, const portfolioData &env)
{
    double r = env.currentAssets - env.previousAssets;
   // std::cout<<"r: " << r << std::endl;
    if(RL.action)
        RL.Qbuy = (1 - RL.alpha) * RL.Qbuy + (RL.alpha*r);
    else
        RL.Qsell = (1 - RL.alpha) * RL.Qsell + (RL.alpha*r);
}

void ToFile(const double &profits, const std::vector<double> &Qsells,
            const std::vector<double> & Qbuys,const std::vector<bool> & actions,
            const double &alpha, const double &beta)
{
    std::ofstream file;

    file.open ("RL_Profits.txt", std::ios_base::app);
        file << std::fixed << profits << std::endl;
    file.close();

    file.open ("RL_Qsells.txt", std::ios_base::app);
    for(int i = 0; i < Qsells.size(); i++)
        file << std::fixed << Qsells[i] << std::endl;
    file.close();

    file.open ("RL_Qbuys.txt", std::ios_base::app);
    for(int i = 0; i < Qbuys.size(); i++)
        file << std::fixed << Qbuys[i] << std::endl;
    file.close();

    file.open ("RL_Actions.txt", std::ios_base::app);
    for(int i = 0; i < actions.size(); i++)
        file << actions[i] << std::endl;
    file.close();

    file.open ("RL_AlphaBeta.txt", std::ios_base::app);
    file << alpha << " " << beta << " " << std::endl;
    file.close();
}

void ToFile2(const std::vector<double> & profits, std::string fileNum)
{
    std::ofstream file;
    std::string fileName = "RLprofits";
    fileName += fileNum;
    fileName += ".txt";
    file.open ("./data/0.21_6.7_data.txt", std::ios_base::app);
    for(int i = 0; i < profits.size(); i++)



        file << std::fixed << profits[i] << std::endl;
    file << "NEW SET" << std::endl;
    file.close();
}

void ToFile3(const double &profits, const double &alpha, const double &beta, const
 std::string &fileNum)
{
    std::ofstream file;
    file.open("./data/data2.txt", std::ios_base::app);
    file << std::fixed << alpha << " " << beta << " " << profits << std::endl;
    file.close();
}
```

```
#include <iostream>
#include <unistd.h>
#include <stdlib.h>
#include <string>


int main()
{
   /* for(double a = 0.01; a < 1; a += 0.01){
        for(double b = 0.1; b < 10; b += 0.1){
            for(int i = 0; i < 100; i++){
                std::string tempa = std::to_string(a);
                std::string tempb = std::to_string(b);
                std::string filenum = std::to_string(a) + " " + std::to_string(b) +
                 ".txt";
                const char* const args[] = {"./RL", tempa.c_str(),
                 tempb.c_str(),filenum.c_str(), nullptr};
                std::cout << std::endl << "Calling: " << a << " " << b << std::endl;
                if (fork() == 0)
                    execvp(args[0], const_cast<char*const*>(args));
                else
                    wait(NULL);
                }
            }
        }*/
    std::string a = "0.21";
    std::string b = "6.7";
    for (int i = 1; i <= 30; i++){
        int count = i;
        std::string temp = std::to_string(count);
        const char* const args[] = {"./RL", a.c_str(), b.c_str(), temp.c_str(),
         nullptr};
        std::cout << std::endl << "Calling: " << a << " " << b << std::endl;
        if (fork() == 0)
            execvp(args[0], const_cast<char*const*>(args));
        else
            wait(NULL);
    }
    return 0;
    /*
    std::string a = "1";
    for(double b = 0.1; b < 10; b+= 0.1){
        for(int i = 0; i < 100; i++){
            std::string tempb = std::to_string(b);
            std::string tempc = "test";
            const char* const args[] = {"./RL", a.c_str(),
             tempb.c_str(),tempc.c_str(), nullptr};
            std::cout << std::endl << "Calling: " << a << " " << b << std::endl;
            if (fork()==0)
                execvp(args[0], const_cast<char*const*>(args));
            else
                wait(NULL);
        }
    }*/
```

Appendix D - Neural Network Code

```
import math
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.layers import Activation

class portfolio:
    def __init__(self, bank = 10000, stocksOwned = 0, profits = 0):
        self.bank = bank
        self.stocksOwned = stocksOwned
        self.assets = bank
        self.profits = 0

    def executeAction(self, action, price):
        if action: #Buy
            self.stocksOwned += 1
            self.bank -= price
            self.assets = self.bank + (price * self.stocksOwned)

        else: #Sell
            self.stocksOwned -= 1
            self.bank  += price
            self.assets = self.bank + (price * self.stocksOwned)

    def calcProfits(self):
        self.profits = self.assets-10000
        self.profits = np.round(self.profits,2)
        self.profits = str(self.profits)
        self.profits += "\n"

    def chooseAction(self, predicted, y): #Returns true if predicted value n is an
     increase over y @ n-1, else returns false
        return predicted > y

def generateDataSet():
    df = pd.read_csv('DJI.csv', usecols=[0])
    df = df.values
    return (df.astype('float32'))

def generateTrainTestData(df, splitSize): # split into train and test sets
    train_size = int(len(df) * splitSize)
    test_size = len(df) - train_size
    return (dataset_scaled[0:train_size,:],
     dataset_scaled[train_size:len(dataset),:])

def create_dataset(dataset, look_back):
    dataX, dataY = [], []
```

```
        for i in range(look_back, len(dataset)):
            dataX.append(dataset[i-look_back:i,0])
            dataY.append(dataset[i,0])
        return np.array(dataX), np.array(dataY)

    def configureLSTMnetwork(numLayers, numUnits, dropoutVal=None,activationType=None):

         model = Sequential()
         for i in range(0, numLayers-1):
             model.add(LSTM(
                           units = numUnits[i],
                           return_sequences = True,
                           input_shape = (trainX.shape[1], 1)))
             if dropoutVal is not None:
                 model.add(Dropout(dropoutVal))

         model.add(LSTM(
                       units = numUnits[-1]))

         if dropoutVal is not None:
             model.add(Dropout(dropoutVal))

         model.add(Dense(units = 1))

         if activationType is not None:
             model.add(Activation(activationType))
         else:
             model.add(Activation('linear'))

         return model

    def runLSTMnetwork(model, trainX, trainY, numEpochs, numBatch,lossType = None,
     optimizerType = None, validationSize = None):
        if lossType is not None:
            if optimizerType is not None:
                model.compile(loss = lossType, optimizer = optimizerType)
            else:
                model.compile(loss = lossType, optimizer = 'rmsprop')

        else:
            if optimizerType is not None:
                model.compile(loss = 'mean_squared_error', optimizer = optimizerType)
            else:
                model.compile(loss = 'mean_squared_error', optimizer = 'rmsprop')

            if validationSize is not None:
                model.fit(
                          trainX,
                          trainY,
                          epochs = numEpochs,
                          batch_size = numBatch,
                          verbose = 1,
                          validation_split = validationSize)
            else:
```

```
            model.fit(
                      trainX,
                      trainY,
                      epochs = numEpochs,
                      batch_size = numBatch,
                      verbose = 1)

        return model

def writeProfitsToFile(profitsList):
    file = open("LSTMprofits80%.txt", "w")
    for i in profitsList:
        file.write(str(i)+"\n")
    file.close()

def tradingSimulation(trainingPredictions, testPredictictions, actualPrices):
    pfolio = portfolio()
    proflist = []
    for i in range(len(trainingPredictions)):
        action = pfolio.chooseAction(trainingPredictions[i], actualPrices[i+4])
        pfolio.executeAction(action, actualPrices[i+4])
        pfolio.calcProfits()
        proflist.append(pfolio.profits)

    for i in range(len(testPredictictions)):
        action = pfolio.chooseAction(testPredictictions[i],
         actualPrices[50+i+look_back-1])
        pfolio.executeAction(action, actualPrices[55+i-1])
        pfolio.calcProfits()
        proflist.append(pfolio.profits)

    writeProfitsToFile(proflist)

def graphPredictions(dataset, trainPredict, testPredict):
    # shift train predictions for plotting
    trainPredictPlot = np.empty_like(dataset)
    trainPredictPlot[:, :] = np.nan
    trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict

    # shift test predictions for plotting
    testPredictPlot = np.empty_like(dataset)
    testPredictPlot[:, :] = np.nan
    testPredictPlot[len(trainPredict)+(look_back*2):len(dataset)+1, :] = testPredict

    # plot baseline and predictions
    plt.plot(dataset, color = 'r', label = 'Actual Prices')
    plt.plot(trainPredictPlot, 'b', label = 'Training Data Predictions')
    plt.plot(testPredictPlot, 'g', label = 'Test Data Predictions')
    plt.title('Dow Jones Neural Network Predictions with 80% Training Data')
    plt.xlabel('Year')
    plt.ylabel('Price ($USD)')
    plt.legend()
    plt.show()
```

```
if __name__ == "__main__":
    random.seed(17)

    dataset = generateDataSet()

    sc = MinMaxScaler(feature_range = (0,1))
    dataset_scaled = sc.fit_transform(dataset)

    train, test = generateTrainTestData(dataset_scaled, 0.8) #2nd parameter = % of
     data in training set

    look_back = 5

    trainX, trainY = create_dataset(train, look_back)
    testX, testY = create_dataset(test, look_back)

    trainX = np.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
    testX = np.reshape(testX, (testX.shape[0], testX.shape[1], 1))

    units = [100,50]
    model = configureLSTMnetwork(2,units,0.2) #'X' layers, list of units in each
     layer, 'X' % dropout

    model = runLSTMnetwork(model, trainX, trainY, 100, 1) #'X' epochs, 'X' batch
     size

    trainScore = model.evaluate(trainX, trainY, verbose = 0)
    print('Train Score: %.2f MSE (%.2f RMSE)' % (trainScore, math.sqrt(trainScore)))
    testScore = model.evaluate(testX, testY, verbose=0)
    print('Test Score: %.2f MSE (%.2f RMSE)' % (testScore, math.sqrt(testScore)))

    trainPredict = model.predict(trainX)
    testPredict = model.predict(testX)

    trainPredict = sc.inverse_transform(trainPredict)
    testPredict = sc.inverse_transform(testPredict)

    tradingSimulation(trainPredict,testPredict,dataset)

    graphPredictions(dataset, trainPredict, testPredict)
```