A Survey of Deep Learning Architectures for Algorithmic Cryptocurrency Trading

By Christopher Clark, M.S. Statistics, University of Colorado Denver, May 2022

Advisor: Dr. Erin Austin

Co-advisors: Dr. Troy Butler and Dr. Yaning Liu

Table of Contents

1. Abstract

Independent of one's belief in cryptocurrency as an alternative to fiat currency, it is difficult to deny the enthusiasm of investors, both institutional and retail. Several properties of this novel asset set it apart and demand advanced methods of analysis and prediction. One area is in algorithmic trading, wherein a program extracts data, analyzes it, decides on an action, and executes that action on behalf of the investor. For this, many statistical techniques have been brought to bear on predicting how the market will behave in the immediate future for the decision-making process in the present. In this manuscript, we explore several statistical and artificial-intelligence methods applied to predicting price changes of the highest-market-valued cryptocurrency, Bitcoin. Our training data consists of a roughly three-and-a-half-year period of hourly data from mid-2018 to mid-2021 and our training data is the last six months of 2021. Starting with creating a matrix of common market analysis techniques, known as Technical Indicators, and applying Principal Component Analysis to reduce our features, we then create trading agents with the following classical statistical, Artificial Neural Network, and Reinforcement Learning models: Exponential Smoothing, Seasonal AutoRegressive Integrated Moving Average with eXogenous variables, eXtreme Gradient Boosting, Multi-Layer Perceptron, Convolutional Neural Network, Long-Short Term Memory Recurrent Neural Network, and two variations of Deep Reinforcement Learning. Additionally, we compare their performance with a random agent, three agents trading from three technical indicators themselves, and Buy-and-Hold. We find that the best overall algorithm for optimizing a portfolio consisting solely of Bitcoin is a Convolutional Neural Network-Based Twin Delayed Deep Deterministic Policy Gradient. This algorithm combines two neural networks, an *actor* capable of making decisions in a continuous action space and a *critic* capable of critiquing those actions in a continuous state space, allowing for more nuanced investing decisions in an uncertain market environment.

2. Introduction

Cryptocurrency is a novel, non-traditional currency existing on the backbone of an equally new technology, the blockchain. From its inception by one Satoshi Nakamoto, a mysterious figure who gave the world a structure to build a decentralized currency [1], one not relying on banks or governments for its legitimacy, to the present, cryptocurrency has exploded in value and in controversy. Being decentralized, it has garnered criticism for its almost tailor-made applicability for illegal trade and tax evasion. At the same time, proponents envision a currency that is not subject to the whims of a government and that is more stable than those entities that control centralized currency. In developing countries, cryptocurrency is being used to circumvent corruption in the traditional banking systems and by political dissidents to fund their campaigns without interference. Positive or negative, it is undeniable that cryptocurrency presents unique opportunities, and the blockchain technology on which it is founded is rapidly being adopted by companies who seek to create permanent records accessible by everyone involved. *Bitcoin* (BTC), the highest-market-valued coin and the focus of this project, has seen explosive growth that attracts those looking to get rich quick and drawn the ire of those who see its volatility as a hinderance to its widespread adoption.

Traditional investors have commented both in favor and against the currency. Warren Buffet, perhaps the best investor in history, was quoted [2] as saying "Bitcoin its ingenious and blockchain is important but Bitcoin has no unique value at all it doesn't produce anything. You can stare at it all day and no little bitcoins come out or anything like that. It's a delusion basically." Should we listen to the *Oracle of Omaha*, as he is affectionately known? Or perhaps another famous investor, Mark Cuban, and the world's richest person, Elon Musk, two proponents of cryptocurrency, might change a skeptic's mind?

Looking out into the world to see how governments are handling the emergent phenomena is equally confusing. El Salvador has adopted Bitcoin as legal tender recently [3], and China's several attacks on the virtual currencies provide fascinating, contrary case-studies [4]. In the USA and abroad, banks, such as Morgan Stanley, and well-known investment companies, have begun to invest in cryptocurrency, while others continue to weigh the pros and cons [5].

Pivoting from cryptocurrency to more traditional markets, investing has changed dramatically over the years. The advent of computers and financial-modeling software has transformed the market. According to the *Staff Report on Algorithmic Trading in U.S. Capital Markets* for 2020 [6], "In 2019, national securities exchanges together executed approximately 78% of trades, …" and regarding these national securities exchanges, "Trading and communication at national securities exchanges are now almost entirely automated. Order entry, message acknowledgement, matching algorithms, trade confirmations, and market data systems all operate at microsecond or nanosecond timescales." [6] Most trading is not by humans, but by human-developed algorithms, a set of instructions that receives information from the market, decides on an action, and executes that action. These algorithms seek to identify entry points into the market, or *Buy* signals, and when to exit a trade, or *Sell* signals. As each financial entity seeks an advantage in the market, increasingly complex methods of analysis and prediction move from their original domains into the world of quantitative finance. This paper seeks to apply cutting

edge *Artificial Intelligence* (AI) algorithms, known as *Artificial Neural Networks* (ANN) for their historical motivation and structure akin to the biological brain, to the problem of analyzing cryptocurrency markets in search of profitable entry and exit signals for algorithmic trading.

ANNs, like the brain, have computational *neurons* which can receive information from many other neurons and transmit their information, allowing for complex patterns in data to be discovered. As such, they have found their way into everyday activities such as text prediction, identifying faces within a picture, machine translation, and a host of other uses. They are being applied, quite creatively, to the financial data space, as well. As computation and access to data expand, deeper, more complex, and therefore, more capable of learning complexity, ANNs known as *Deep Neural Networks* (DNN, or *Deep Learning*), should provide even better trading signal identification than their simpler counterparts.

To analyze the cryptocurrency markets, we use traditional *Technical Analysis Indicators* (TA), which are often used to signal entry and exit points into a trade, shifts in the market, and other patterns one might take advantage of. From these TA, we perform *Principal Component Analysis* (PCA), a kind of *dimensionality reduction* algorithm to simplify our data to train our DNNs faster and remove redundancy. The deep learning architectures we use in this study are *Multi-Layer Perceptron* (MLP) networks, *Convolutional Neural Networks* (CNN), *Long Short-Term Memory Recurrent Neural Networks* (LSTM), and two variations of *Reinforcement Learning* (RL) known as *Deep Q-Networks* (DQN) and *Deep Deterministic Policy Gradient* (DDPG).

The MLP, CNN, and LSTM are used for classification, wherein we will attempt to classify an hour as *Buy* or *Sell*. The intention is that an agent will then act on whether it should buy into the market or exit at that hour. The DQN and DDPG are agents whose goal is to maximize their investment, though the DQN will be classifying and acting on those classifications and the DDPG will be classifying *and* determining how much it should invest or sell at each point.

Then, we compare trading strategies based on each of these neural network models with two more common regression time-series models, *Seasonal AutoRegressive Integrated Moving Average with eXogenous variables* (SARIMAX), and *Exponential Smoothing* (ETS). For these two, our predicted values are considered only as positive or negative, indicating Buy or Sell, respectively. We also compare these algorithms' trading with a popular classification algorithm *eXtreme Gradient Boosting* (XGBoost), more naïve trading strategies using several individual indicators themselves, random-action agents, and the simple strategy known as Buy-and-Hold. Our goal is to identify the usefulness of advanced artificial intelligence modeling in outperforming the Bitcoin market, or, in other words, outperforming Buy-and-Hold.

3. Background and Literature Review

Suppose someone wished to train a DNN to determine if a picture is a dog or cat. For this task, they would turn to a CNN. These are commonly used for image analysis, in real time (for example, self-driving cars) or otherwise, such as identifying participants in a photo on social media and helping doctors diagnose diseases based on medical imaging. Their unique structure allows for filters to *convolve* (hence their name) around an image and identify increasingly complex abstractions in pixel data and use those to aid in classification. However, instead of cats and dogs, we consider the question of whether they can be trained for Buy and Sell signals? For this, we have the example of Omer Berat Sezera, et al. In their paper *Algorithmic Financial Trading with Deep Convolutional Neural Networks: Time Series to Image Conversion Approach* [1], they transform financial data into images on which they train their CNNs for classification. This is one such approach adopted in this paper.

A problem with CNNs, though, is that they cannot easily make sense of sequential data. They are fed the data without reference to time or order. For this reason, they are not good with translation, for example, where the beginning of a sentence might have a long-lasting effect on the conjugation of verbs at the end. A better ANN for this task is an LSTM. An LSTM can look at an input sequentially and learn how the beginning, middle, and end influence each other through various gates controlling the sequential flow of information through the network. Given that financial data is sequential, this is a reasonable approach. In the paper *A CNN-LSTM-Based Model to Forecast Stock Prices,* Jiazheng Li et al. [2], create a model first using CNNs to identify features most salient to forecast a stock's value and then pass them through an LSTM to take advantage of the sequential analysis such ANNs provide. They then compare their results with other ANNs, including a CNN and an LSTM by themselves. We will not be applying the CNN-LSTM architecture, but instead will use an LSTM for classification.

Another avenue we consider is casting financial markets as *games*, and asking if we could train an ANN to *play* the game well? Taking our cue from the paper *Deep Reinforcement Learning for Automated Stock Trading: An Ensemble Strategy* by Hongyang Yang et al. [3], we create an environment based on the market and train an agent to *play the market* to the best of its abilities. For this, we explore several RL architectures, including some used in this paper that demonstrate promise.

Looking for more information about the large and rapidly developing field of RL and how it might be applied to trading, we have a detailed overview in *Deep Reinforcement Learning in Quantitative Algorithmic Trading: A Review* by Tidor-Vlad Pricope [4]. In it, he reviews the main divisions in RL and their individual strengths and weaknesses in trading and the results other research groups have had in applying them. From this paper, we decide to use DQN and DDPG as our RL algorithms. DQN was chosen for its use of ANNs as the decision-making entity as our agent plays the market and learns, and DDPG was chosen for the same reasons but with the added benefit that instead of completely buying into the market or leaving in its entirety, a DDPG agent can invest or remove any amount of its investment at a given step.

Finally, we can combine ANN and RL architectures as Zhichao Jia et al. [5] demonstrate in their paper *LSTM-DDPG for Trading with Variable Positions*.

In each of the above papers, the authors find that adopting ANNs leads to better performance than their respective baselines in traditional markets. However, as mentioned earlier, cryptocurrency markets are notoriously volatile and are less studied than their well-developed counterparts. Will we find the same positive results, that ANNs can provide a trading advantage for Bitcoin investors, or will we find our agents trading randomly and unable to beat Buy-and-Hold? We seek to answer this question.

4. Primers

Due to the large number of diverse algorithms in this manuscript, we include here three *Primers* meant to help remind the reader of the basics of each category of algorithms. This will prevent us from having to delve into significant detail when we discuss their specific implementations or modifications. This section can be skipped if the reader is familiar with *Classical Statistical Methods and eXtreme Gradient Boosting* (Section A), *Artificial Neural Networks* (Section B), and *Reinforcement Learning* (Section C).

A. Primer on Classical Statistical Methods and eXtreme Gradient Boosting

i. Exponential Smoothing:

- Simple Exponential Smoothing:

In *Simple Exponential Smoothing*, the estimate of the $t+1^{th}$ value of a time series is modeled as a weighted average of the most recent values that came before it [1]:

$$\hat{y}_{t+1|t} = \alpha y_t + \alpha(1-\alpha)y_{t-1} + \cdots$$

Equivalently, this can be thought of as a combination of the most recent value, y_t, and the previous forecast, $\hat{y}_{t|t-1}$[1]:

$$\hat{y}_{t+1|t} = \alpha y_t + \alpha(1-\alpha)\hat{y}_{t|t-1}$$

Finally, to aid us in writing further exponential smoothing models, we will write this in *Component Form* to see that the equation has two pieces, the *Forecast,* and the *Level Smoothing* [1]:

$$Forecast: \hat{y}_{t+h|t} = l_t$$

$$Level\ Smoothing: l_t = \alpha y_t + (1-\alpha)l_{t-1}$$

Here, the l_t term is the estimate of the level of the series at time t and α is the smoothing parameter for the level.

- Trended Method:

If the time series displays a trend, we will add another component, a *Trend Smoothing* one. This model is called the *Holt's Linear Trend* [2]:

$$Forecast: \hat{y}_{t+h|t} = l_t + hb_t$$

$$Level\ Smoothing: l_t = \alpha y_t + (1-\alpha)(l_{t-1} + b_{t-1})$$

$$Trend\ Smoothing: b_t = \beta^*(l_t - l_{t-1}) + (1-\beta^*)b_{t-1}$$

Here, the l_t term is the estimate of the level of the series at time t, α is the smoothing parameter for the level, b_t is the estimate of the trend, or slope, of the series at time t, and β^* is the trend smoothing parameter.

However, it is often the case that a trended time series is expected to approach a constant. If that is the case, we can introduce a *dampening* constant $\phi, 0 < \phi < 1$, which serves to force the series to approach a constant as time goes on [2].

$$Forecast: \hat{y}_{t+h|t} = l_t + (\phi + \phi^2 + \cdots + \phi^h)b_t$$

$$Level\ Smoothing: l_t = \alpha y_t + (1-\alpha)(l_{t-1} + \phi b_{t-1})$$

$$Trend\ Smoothing: b_t = \beta^*(l_t - l_{t-1}) + (1-\beta^*)\phi b_{t-1}$$

- Seasonality:

If our time series has a seasonal component, we will add a *Seasonal Smoothing* one to our model. This has one of two forms, *additive* or *multiplicative*. We show two examples, the additive one [3]:

$$Forecast: \hat{y}_{t+h|t} = l_t + hb_t + s_{t+h-m(k+1)}$$

$$Level\ Smoothing: l_t = \alpha(y_t - s_{t-m}) + (1-\alpha)(l_{t-1} + b_{t-1})$$

$$Trend\ Smoothing: b_t = \beta^*(l_t - l_{t-1}) + (1-\beta^*)b_{t-1}$$

$$Seasonal\ Smoothing: s_t = \gamma(y_t - l_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m}$$

and the multiplicative one [3]:

$$Forecast: \hat{y}_{t+h|t} = (l_t + hb_t)s_{t+h-m(k+1)}$$

$$Level\ Smoothing: l_t = \alpha\left(\frac{y_t}{s_{t-m}}\right) + (1-\alpha)(l_{t-1} + b_{t-1})$$

$$Trend\ Smoothing: b_t = \beta^*\,(l_t - l_{t-1}) + (1-\beta^*)b_{t-1}$$

$$Seasonal\ Smoothing: s_t = \gamma\left(\frac{y_t}{l_{t-1} + b_{t-1}}\right) + (1-\gamma)s_{t-m}$$

In these equations m is the seasonality, $k = int\left(\frac{h-1}{m}\right)$, and $0 \le \gamma \le 1-\alpha$.

For brevity, we stop here, as we have shown how one builds increasingly complex *Exponential Smoothing* (ETS) models. However, we are still missing a significant piece, the errors, ϵ_t.

Just as with the multiplicative component, we can consider the errors in an additive or multiplicative way. We can also introduce a dampening parameter into either of the two above models. From this, the number of possible models grows quite large, so we introduce the following notation [4]:

$$ETS(\,E, T, S)$$

The E is the *error* term, the T is the *trend* term, and the S is the *seasonality* term. The errors can be additive or multiplicative, so, $Errors = \{A, M\}$. The trend can be none (absent), additive, or

additive and dampened, so, $Trend = \{N, A, A_d\}$. The season can be none, additive, or multiplicative, so, $S = \{N, A, M\}$. Note, we are not including multiplicative trend models, as they tend to do poorly. Simple arithmetic shows that, with these simple parameters, we have a taxonomy of 18 models. We provide a table for them and close this section of the primer [4].

Table 8.7: State space equations for each of the models in the ETS framework.

ADDITIVE ERROR MODELS

Trend	Seasonal N	Seasonal A	Seasonal M
N	$y_t = \ell_{t-1} + \varepsilon_t$ $\ell_t = \ell_{t-1} + \alpha\varepsilon_t$	$y_t = \ell_{t-1} + s_{t-m} + \varepsilon_t$ $\ell_t = \ell_{t-1} + \alpha\varepsilon_t$ $s_t = s_{t-m} + \gamma\varepsilon_t$	$y_t = \ell_{t-1}s_{t-m} + \varepsilon_t$ $\ell_t = \ell_{t-1} + \alpha\varepsilon_t/s_{t-m}$ $s_t = s_{t-m} + \gamma\varepsilon_t/\ell_{t-1}$
A	$y_t = \ell_{t-1} + b_{t-1} + \varepsilon_t$ $\ell_t = \ell_{t-1} + b_{t-1} + \alpha\varepsilon_t$ $b_t = b_{t-1} + \beta\varepsilon_t$	$y_t = \ell_{t-1} + b_{t-1} + s_{t-m} + \varepsilon_t$ $\ell_t = \ell_{t-1} + b_{t-1} + \alpha\varepsilon_t$ $b_t = b_{t-1} + \beta\varepsilon_t$ $s_t = s_{t-m} + \gamma\varepsilon_t$	$y_t = (\ell_{t-1} + b_{t-1})s_{t-m} + \varepsilon_t$ $\ell_t = \ell_{t-1} + b_{t-1} + \alpha\varepsilon_t/s_{t-m}$ $b_t = b_{t-1} + \beta\varepsilon_t/s_{t-m}$ $s_t = s_{t-m} + \gamma\varepsilon_t/(\ell_{t-1} + b_{t-1})$
A_d	$y_t = \ell_{t-1} + \phi b_{t-1} + \varepsilon_t$ $\ell_t = \ell_{t-1} + \phi b_{t-1} + \alpha\varepsilon_t$ $b_t = \phi b_{t-1} + \beta\varepsilon_t$	$y_t = \ell_{t-1} + \phi b_{t-1} + s_{t-m} + \varepsilon_t$ $\ell_t = \ell_{t-1} + \phi b_{t-1} + \alpha\varepsilon_t$ $b_t = \phi b_{t-1} + \beta\varepsilon_t$ $s_t = s_{t-m} + \gamma\varepsilon_t$	$y_t = (\ell_{t-1} + \phi b_{t-1})s_{t-m} + \varepsilon_t$ $\ell_t = \ell_{t-1} + \phi b_{t-1} + \alpha\varepsilon_t/s_{t-m}$ $b_t = \phi b_{t-1} + \beta\varepsilon_t/s_{t-m}$ $s_t = s_{t-m} + \gamma\varepsilon_t/(\ell_{t-1} + \phi b_{t-1})$

MULTIPLICATIVE ERROR MODELS

Trend	Seasonal N	Seasonal A	Seasonal M
N	$y_t = \ell_{t-1}(1 + \varepsilon_t)$ $\ell_t = \ell_{t-1}(1 + \alpha\varepsilon_t)$	$y_t = (\ell_{t-1} + s_{t-m})(1 + \varepsilon_t)$ $\ell_t = \ell_{t-1} + \alpha(\ell_{t-1} + s_{t-m})\varepsilon_t$ $s_t = s_{t-m} + \gamma(\ell_{t-1} + s_{t-m})\varepsilon_t$	$y_t = \ell_{t-1}s_{t-m}(1 + \varepsilon_t)$ $\ell_t = \ell_{t-1}(1 + \alpha\varepsilon_t)$ $s_t = s_{t-m}(1 + \gamma\varepsilon_t)$
A	$y_t = (\ell_{t-1} + b_{t-1})(1 + \varepsilon_t)$ $\ell_t = (\ell_{t-1} + b_{t-1})(1 + \alpha\varepsilon_t)$ $b_t = b_{t-1} + \beta(\ell_{t-1} + b_{t-1})\varepsilon_t$	$y_t = (\ell_{t-1} + b_{t-1} + s_{t-m})(1 + \varepsilon_t)$ $\ell_t = \ell_{t-1} + b_{t-1} + \alpha(\ell_{t-1} + b_{t-1} + s_{t-m})\varepsilon_t$ $b_t = b_{t-1} + \beta(\ell_{t-1} + b_{t-1} + s_{t-m})\varepsilon_t$ $s_t = s_{t-m} + \gamma(\ell_{t-1} + b_{t-1} + s_{t-m})\varepsilon_t$	$y_t = (\ell_{t-1} + b_{t-1})s_{t-m}(1 + \varepsilon_t)$ $\ell_t = (\ell_{t-1} + b_{t-1})(1 + \alpha\varepsilon_t)$ $b_t = b_{t-1} + \beta(\ell_{t-1} + b_{t-1})\varepsilon_t$ $s_t = s_{t-m}(1 + \gamma\varepsilon_t)$
A_d	$y_t = (\ell_{t-1} + \phi b_{t-1})(1 + \varepsilon_t)$ $\ell_t = (\ell_{t-1} + \phi b_{t-1})(1 + \alpha\varepsilon_t)$ $b_t = \phi b_{t-1} + \beta(\ell_{t-1} + \phi b_{t-1})\varepsilon_t$	$y_t = (\ell_{t-1} + \phi b_{t-1} + s_{t-m})(1 + \varepsilon_t)$ $\ell_t = \ell_{t-1} + \phi b_{t-1} + \alpha(\ell_{t-1} + \phi b_{t-1} + s_{t-m})\varepsilon_t$ $b_t = \phi b_{t-1} + \beta(\ell_{t-1} + \phi b_{t-1} + s_{t-m})\varepsilon_t$ $s_t = s_{t-m} + \gamma(\ell_{t-1} + \phi b_{t-1} + s_{t-m})\varepsilon_t$	$y_t = (\ell_{t-1} + \phi b_{t-1})s_{t-m}(1 + \varepsilon_t)$ $\ell_t = (\ell_{t-1} + \phi b_{t-1})(1 + \alpha\varepsilon_t)$ $b_t = \phi b_{t-1} + \beta(\ell_{t-1} + \phi b_{t-1})\varepsilon_t$ $s_t = s_{t-m}(1 + \gamma\varepsilon_t)$

ii. Seasonal AutoRegressive Integrated Moving Average with eXogenous variables**:**

- AutoRegressive (AR) Models:

In an autoregressive model, the variable of interest y at time t, with $y_t \in \mathbb{R}$, is forecast as a linear combination of past values of the variable. The general structure has the following form [5, 6]:

$$y_t = \sum_{i=1}^{p} \phi_i y_{t-i} + c + \epsilon_t$$

In the above equation, p is the order of the model, or the number of *lagged values* of y_t, c is the *intercept*, and ϵ_t is the *error* at time t.

We can also introduce another common notation, L, the *Lag Operator* [7]:

$$L^n y_t = y_{t-n}$$

From the lag operator, we can define the *Lag Polynomial for AR Processes*, $\phi(L)^p$, also written as $\phi_p(L)$. In this notation, we can rewrite the first equation (ignoring the intercept) as [8, 9]:

$$\phi(L)^p y_t = \left(1 - \sum_{i=1}^{p} \phi_i L^i\right) y_t = \epsilon_t$$

We can rearrange this to get the first AR equation:

$$y_t = \epsilon_t + \sum_{i=1}^{p} \phi_i L^i \, y_t$$

- Moving Average (MA) Models:

In a moving average model, the variable of interest at time t is forecast as a linear combination of past forecast errors, ϵ_t. The general structure has the following form [6, 10]:

$$y_t = \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + c + \epsilon_t$$

In the above equation, q is the order of the model, or the number of past errors being used, c is the intercept, and ϵ_t is the error at time t.

With the *Lag Polynomial for MA Processes*, $\theta(L)^q$, notation, our MA model becomes [8]:

$$y_t = \left(1 + \sum_{i=1}^{q} \theta_i L^i\right) \epsilon_t = \theta(L)^q \epsilon_t$$

- ARMA Model:

We can then combine them to create an *Autoregressive Moving Average* (ARMA) model [6, 8]:

$$y_t = \sum_{i=1}^{p} \phi_i y_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + c + \epsilon_t$$

Or, using our other notation, dropping the constant, and rearranging a bit, we get:

$$\phi(L)^p y_t = \theta(L)^q \epsilon_t$$

- Differencing*:*

In general, it is easier to forecast from a s*tationary* series, one whose statistical properties do not depend on the time at which the series is observed. We make a series stationary by performing *Differencing*, or taking the difference between two consecutive values of the series [11]:

$$y'_t = y_t - y_{t-1}$$

Using lag notation, we write this as:

$$\Delta y_t = (1 - L) y_t = y_t - y_{t-1} = y'_t$$

We use several different stationarity tests, such as the *Augmented Dickey-Fuller Test*, to check for stationarity and perform the necessary number of differencings to achieve stationarity. The number of differencings is known as parameter d. We also write the relationship as follows [6, 7, 11]:

$$\Delta^{\mathrm{d}} y_t = (1 - \mathrm{L})^d y_t = d_t$$

- ARIMA:

Now, we combine each of these terms to create an *AutoRegressive Integrated Moving Average* (ARIMA) model [6]:

$$d_t = \sum_{i=1}^{p} \phi_i d_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + c + \epsilon_t$$

Using our lag operator notation and dropping the constant, this is [9]:

$$\phi(L)^p \Delta^d y_t = \theta(L)^q \epsilon_t$$

This ARIMA model is often written in shorthand as $ARIMA(p, d, q)$.

- ARIMAX:

However, this model only used errors and past values of the series to forecast future values. If we wish to include more variables, $\boldsymbol{x} = (x_1, \dots, x_r)$, named *exogenous* variables, we now have an *AutoRegressive Integrated Moving Average with eXogenous variables* (ARIMAX) model. To make this change, we need only add a linear combination of the exogenous variables to our original ARIMA model [4]:

$$d_t = \sum_{i=1}^{p} \phi_i d_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \sum_{i=1}^{r} \beta_i x_t^i + c + \epsilon_t$$

Using our lag operator notation and dropping the constant, this is [9]:

$$\phi(L)^p \Delta^d y_t = \theta(L)^q \epsilon_t + \sum_{i=1}^{r} \beta_i x_t^i$$

- Seasonality and SARIMAX:

Often, our data has a *Seasonal* component, or a cycle that repeats every certain number of time periods. Perhaps our data shows significant changes throughout the month, or week, or day. In that case, we need to consider the period of the season and integrate it into our model. Now, our variable of interest might depend on not just several past instances or errors, but on values from a past seasonal period. Fortunately, we treat the seasonal component with an ARIMA model, as well. Though, we now use the parameters $(P, D, Q)_m$, where P is the *Seasonal Autoregressive* parameter, D is the *Seasonal Differencing* parameter, Q is the *Seasonal Moving Average* parameter, and m is the *Seasonality*, or the number of time periods in a season. For example, on daily data that has a weekly seasonal component, $m = 7$.

We augment our $ARIMAX(p, d, q)$ model to become a $SARIMAX(p, d, q)(P, D, Q)_m$ model with the form [6]:

$$d_t = \sum_{i=1}^{p} \phi_i d_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \sum_{i=1}^{r} \beta_i x_t^i + \sum_{i=1}^{P} \tilde{\phi}_i d_{t-im} + \sum_{i=1}^{Q} \tilde{\theta}_i \epsilon_{t-im} + c + \epsilon_t$$

As above, we rearrange this and drop the constant to arrive at a more common form [9, 12]:

$$\phi(L)^p \, \tilde{\phi} \, (L^m)^P \Delta^d \Delta_{\mathrm{m}}^D \, y_t = \theta(L)^q \, \tilde{\theta} (L^m)^Q \epsilon_t + \sum_{i=1}^{r} \beta_i x_t^i$$

We note here that the Δ_m^{D} is, like our two lag polynomials, formulated in terms of L^m. So, $\Delta_m^{\mathrm{D}} = (1 - L^m)^D$.

There is one final piece missing, the *Trend Polynomial*, $A(t)$. With it, our final formulation becomes [9, 12]:

$$\phi(L)^p \, \tilde{\phi} \, (L^m)^P \Delta^d \Delta_{\mathrm{m}}^D \, y_t = A(t) + \theta(L)^q \, \tilde{\theta}(L^m)^Q \epsilon_t + \sum_{i=1}^{r} \beta_i x_i$$

We note that, though this formulation is, in our opinion, simpler to understand, another common way in which is it is written is [12]:

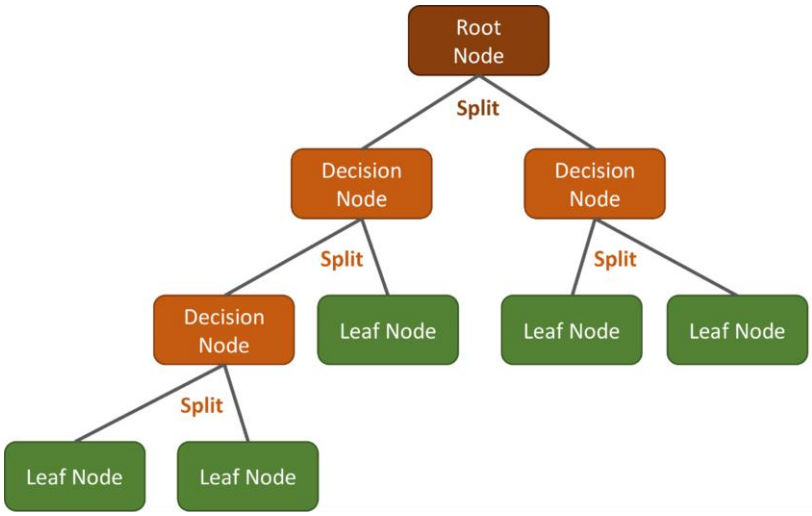
$$y_t = \sum_{i=1}^{r} \beta_i x_i + u_t$$

$$\phi(L)^p \, \tilde{\phi} \, (L^m)^P \Delta^d \Delta_{\mathrm{m}}^D \, u_t = A(t) + \theta(L)^q \, \tilde{\theta}(L^m)^Q \epsilon_t$$

iii. eXtreme Gradient Boosting

eXtreme Gradient Boosting (XGBoost) is an algorithm that has proven both quick and accurate, garnering many wins in online competitions, such as Kaggle [13]. It is both an algorithm in the traditional sense, as well as a specific implementation [14]. We provide a brief mathematical overview of how the algorithm works and direct the reader to the developers' resources for computational specifics [15].

To begin, we introduce the concept of a *Classification and Regression Tree* (CART). They are simple and intuitive tools to make a classification or regression based on iterative splitting of a decision through features of the data. A simple picture helps to elucidate why they have the name *tree* [16]:



Now, though we could create a single CART and attempt our task with that, it is more beneficial to combine several of them, giving us an *ensemble* of trees. Consider a set of m data instances $\boldsymbol{x}_i$ having n features collected into X. Each $\boldsymbol{x}_i$ has and a corresponding label y_i collected into Y. Then, an ensemble model of trees has the following form [14, 17]:

$$\hat{y}_i = \sum_{k=1}^{K} f_k(\boldsymbol{x}_i)$$

Let us explore a bit more the structure of these CART functions f. Each has a decision algorithm that takes an instance of data $\boldsymbol{x}$ and assigns it to a leaf. Call this $q(\boldsymbol{x})$. Then, at each leaf, it recieves a weight w. This is the value of $f(\boldsymbol{x})$. Thus, we think of $f(\boldsymbol{x})$ as $f_{(q,\boldsymbol{w})}(\boldsymbol{x})$. Here, the vector $\boldsymbol{w}$ is a vector of length T, the number of leaves, and the decision algorithm $q(\boldsymbol{x})$ recieves $\boldsymbol{x}$ and assigns it to a leaf with weight w_j. Formally [17]:

$$f(\boldsymbol{x}) = w_{q(\boldsymbol{x})}$$

$$q: \mathbb{R}^n \to \{1, 2, \ldots T\}, w \in \mathbb{R}^T$$

Let $\mathcal{F}$ be the set of all CARTS. So, $\mathcal{F} = \{f: f(\boldsymbol{x}) = w_{q(\boldsymbol{x})}\}$ [17].

Now, we build an objective function, J, that we seek to minimize. We include a *Loss Function*, $\mathcal{L}$, and a *Regularization Function*, Ω. [17]

We note that this is a slight notational deviation. In the original paper, l represented the loss function. We will also replace the use of $\mathcal{L}$ used in the original paper as the objective function for J. This notation will carry throughout our *Primers*, so we wish to remain consistent.

To properly define J, we first define the parameter set Ξ to be a set of CART functions used to create our ensemble tree model. Then, $J = J(\Xi)$.

Formally:

$$\Xi = \left\{ f: \hat{y} = \sum_{k} f_k(\boldsymbol{x}) \right\}, f \in \mathcal{F}$$

Thus, our objective function is [14, 17]:

$$J(\Xi) = \sum_{i=1}^{m} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

We are now trying to find these tree functions f to minimize $J(\Xi)$.

Looking at each $\hat{y}_i$, we write out how it is obtained iteratively as [14]:

$$\hat{y}_i^{(0)} = 0$$

$$\hat{y}_i^{(1)} = f_1(\boldsymbol{x}_i) = \hat{y}_i^{(0)} + f_1(\boldsymbol{x}_i)$$

$$\hat{y}_i^{(2)} = f_1(\boldsymbol{x}_i) + f_2(\boldsymbol{x}_i) = \hat{y}_i^{(1)} + f_2(\boldsymbol{x}_i)$$

...

$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(\boldsymbol{x}_i) = \hat{y}_i^{(t-1)} + f_t(\boldsymbol{x}_i)$$

We see that we can minimize $J(\Xi)$ *at each step t*, meaning we are now seeking to minimize the following [14]:

$$J(\Xi)^{(t)} = \sum_{i=1}^{m} \mathcal{L}(y_i, \hat{y}_i^{(t)}) + \sum_{k=1}^{t} \Omega(f_k)$$

We rewrite this as [14]:

$$J(\Xi)^{(t)} = \sum_{i=1}^{m} \mathcal{L}\left(y_i, \hat{y}_i^{(t-1)} + f_t(\boldsymbol{x}_i)\right) + \Omega(f_t) + c$$

Here, c is a constant and the result of the previous additions of $\Omega(f_k)$ up to $t-1$, or:

$$c = \sum_{k=1}^{t-1} f_k(\boldsymbol{x}_i)$$

We note here that our J function at step (t) is only a function of f_t, as everything else has been calculated at previous steps. So, $J(\Xi)^{(t)} = J(f_t)$. This will become important soon, as we begin looking into the parameters of f_t itself.

Now, we take a *second order Taylor approximation* on the loss function $\mathcal{L}$ and get [14, 17]:

$$\tilde{J}(f_t) \approx \sum_{i=1}^{m} \left[\mathcal{L}\left(y_i, \hat{y}_i^{(t-1)}\right) + g_i f_t(\boldsymbol{x}_i) + \frac{1}{2}\, h_i f_t^2(\boldsymbol{x}_i) \right] + \Omega(f_t) + c$$

$$g_i = \frac{\partial \mathcal{L}\left(y_i, \hat{y}_i^{(t-1)}\right)}{\partial \hat{y}_i^{(t-1)}}$$

$$h_i = \frac{\partial^2 \mathcal{L}\left(y_i, \hat{y}_i^{(t-1)}\right)}{\partial \hat{y}_i^{(t-1)^2}}$$

We remove everything constant to get something easier to work with, $\tilde{J}(f_t)$ [14, 17]:

$$\tilde{J}(f_t) = \sum_{i=1}^{m} \left[g_i f_t(\boldsymbol{x}_i) + \frac{1}{2}\, h_i f_t^2(\boldsymbol{x}_i) \right] + \Omega(f_t)$$

Now, we deal with our Ω term, which we write as [14, 17]:

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

We note here that this regularization is $L2$ regularization as presented in the original paper and in the official developer website, but there is a parameter for α, or the $L1$ penalization. As the original paper and developer website use the above formulation, so will we here [18].

Rewriting our objective function in terms of $\boldsymbol{w}$ and q, the parameters of f_t itself, and reintroducing the (t) superscript to remind us we are on iteration t, we get [14, 17]:

$$\tilde{J}(\boldsymbol{w}, q)^{(t)} = \sum_{j=1}^{T} \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma \mathrm{T}$$

In this above formulation, $I_j = \{i : q(\boldsymbol{x}_i) = j\}$, or the set of indices of data points assigned to the jth leaf by some decision function, which we have yet to discover, $q(\boldsymbol{x})$ [14].

Calling $\sum_{i \in I_j} g_i = G_j$ and $\sum_{i \in I_j} h_i = H_j$, we get [14, 17]:

$$\tilde{J}(\boldsymbol{w}, q)^{(t)} = \sum_{j=1}^{T} \left[G_j w_j + \frac{1}{2} \left(H_j + \lambda \right) w_j^2 \right] + \gamma \mathrm{T}$$

We now fix a function q and ask for the best weights for that. We get [14, 17]:

$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

We make another note here of a hyperparameter η available to be set. This is the *Learning Rate* and would multiply the above weight as another form of regularization. Though, as we did with $L1$ regularization above, we maintain the original formulation for this primer [18].

Plugging this into our objective function, we get [14, 17]:

$$\tilde{J}(q)^{(t)} = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T$$

This last functions measures how good a tree structure we have in q.

At this point, we would like to enumerate over all tree structures, but that is prohibitively difficult. Instead, we build each tree level by level, checking to see if each addition or split is beneficial. To do this, we will use the following equation [14, 17]:

$$Gain = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma$$

Above, the *L* and *R* refer to the *right* and *left* leaf generated at the split/new branch. If this gain is smaller than γ, that branch is not beneficial to use. Thus, we can prune our tree.

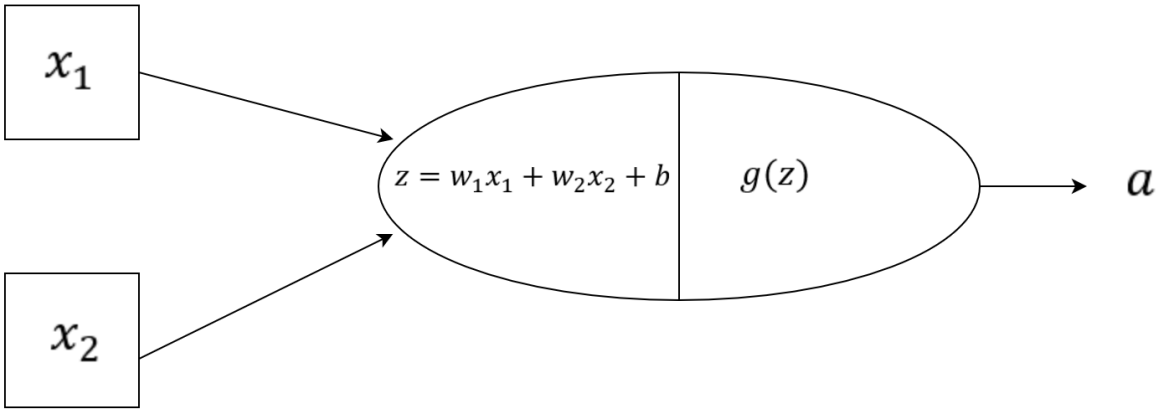
Now that we have the mathematical underpinning of XGBoost, we turn to the documentation for how it is implemented and the computational tricks it uses to achieve the speeds it does [15].

B. Primer on Artificial Neural Networks

Artificial Neural Networks (ANN) take their name and their inspiration from the human brain, wherein a single neuron is connected to, and therefore, receives and transmits, information to and from other neurons. While modern ANNs have evolved beyond this initial understanding, simpler ANNs, such as the Multi-Layer Perceptron used in this paper, are closer in spirit to the original formulation.

i. Perceptron and Simple Artificial Neural Networks

In an ANN, we begin with a single neuron, the *perceptron*, which acts, in its simplest form, as linear regression. They are commonly shown with the following pictorial:



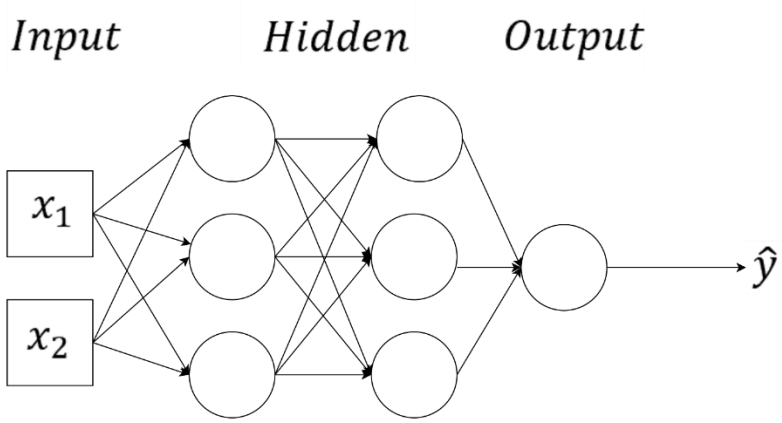
In this diagram, our perceptron recieves from the *Input Layer* $\boldsymbol{x} = (x_1, x_2)$, multiplies $\boldsymbol{x}$ by $\boldsymbol{w}^T = (w_1, w_2)^T$ to get $w_1x_1 + w_2x_2$, then adds a final term b to get $z = w_1x_1 + w_2x_2 + b$. Now, if the final step, $g(z)$ is the identity function, then we have a simple linear equation and our goal is to solve for $\boldsymbol{w}$, the *weights*, and b, the *bias*, to get an output a that we would like. This is linear regression. However, the power of perceptrons is that we can vary the $g(z)$ piece, the *activation function*, for various needs. A short table is provided with some common activations functions and why they might be used [1]:

$\tanh(z)$	$\frac{e^z - e^{-z}}{e^z + e^{-z}}$	This function has a range of $(-1,1)$, and so is used for probabilities.
$\sigma(z)$	$\frac{1}{1 + e^{-z}}$	This function has a range of $(0,1)$, and so is used for probabilities, as well.
$ReLU(z)$	$f(x) = \begin{cases} z, & z > 0 \\ 0, & otherwise \end{cases}$	Acts as linear regression, but with only positive values.
$I(z)$	z	Linear Regression

These activation functions allow an ANN to capture non-linearities in the data.

Often, multiple perceptrons (which we will refer to as *neurons* from now on) are used, in which case, we will have a layer of neurons all receiving the $\boldsymbol{x}$ input, applying their weights, biases, and activation functions to it, then feeding their results into a final *Output Layer*. Depending on the type of problem, the output layer will have one or more neurons and the activation function will be tailored to the task. If it is a *classification*, out final output $\hat{y}$ will be a list of probabilities $[p_1, \dots, p_q]$, with $p_i \in [0,1]$ being the probability of that instance belonging to class i out of q total classes. If our task is *regression*, we wish to provide a continuous value to the input, so $\hat{y} \in \mathbb{R}$.

From here, we begin adding more *Hidden Layers* with more neurons, each with their own biases, weights, and activation functions, to capture more difficult and non-linear relationships between the inputs and the targets, thus moving from a simple ANN to a multi-layered *Deep Neural Network* (DNN).



ii. Deep Neural Networks

- Structure Overview [2]

Formally, we symbolize the structure of an *L*-layered DNN as follows:

Our data has the form $X = (n_x, m)$, where n_x is the number of features of the $\boldsymbol{x}$ instances and m is the number of training examples, and $Y = (1, m)$, a vector of the *actual* values or classifications associated with each $\boldsymbol{x}$ in X. Given that we have the actual values of Y, this is a *Supervised Learning* problem, wherein we hope to train our DNN from known values in order to generalize onto new data.

To keep track of our layer and data instance, we use $*^{[l]}$ to denote the l^{th} layer and $*^{(i)}$ for the i^{th} data instance. Each layer has $n^{[l]}$ neurons. We use vectorized notation to discuss all the weights, biases, and operations taking place in each layer.

Define $A^{[l]}$ as the output of layer l over all instances i. This means that $A^{[0]} = X$. Let $W^{[l]}$ be the matrix of all the weights of the l^{th} layer and $b^{[l]}$ the matrix of all the biases of the l^{th} layer. $W^{[l]}$ is an $\left(n^{[l]}, n^{[l-1]}\right)$ matrix, which is also thought of as $n^{[l]}$ vectors of length $n^{[l-1]}$. Note that the number of columns of this matrix is the number of neurons from the previous layer, as the

information layer l will receive is the outputs of each of the $(l-1)^{th}$ layer's neurons, and $b^{[l]}$ is an $(n^{[l]}, 1)$ matrix. Then, our DNN looks like:

$$A^{[0]} = X$$

$$A^{[1]} = g\left(W^{[1]}X + b^{[1]}\right)$$

$$A^{[2]} = g\left(W^{[2]}A^{[1]} + b^{[2]}\right)$$

...

$$\hat{Y} = A^{[L]} = g\left(W^{[L]}A^{[l-1]} + b^{[L]}\right)$$

Here, $\hat{Y}$ are the *predictions* our DNN has given us for each of our instances.

In more succinct form, we define the output of layer l as [3]:

$$f^{[l]}_{(W^{[l]},b^{[l]})} = g(W^{[l]}f^{[l-1]}_{(W^{[l-1]},b^{[l-1]})} + b^{[l]})$$

and our DNN is written as [3]:

$$\hat{Y} \coloneqq F_{(W,b)}(X) = f^{[L]}_{(W^{[L]},b^{[L]})} \circ \ldots \circ f^{[1]}_{(W^{[1]},b^{[1]})}(X)$$

where the $\circ$ is *composition* operation.

- Feedforward Step

We call this process of feeding X into $F_{(W,b)}$ the *Feedforward Step* and results in our first attempt at predictions, $\hat{Y}$. From this, we identify the trainable *parameters*, which are [2]:

$$\boldsymbol{\Theta} = \left(W^{[1]}, b^{[1]}, \ldots W^{[L]}, b^{[L]}\right) = (W, b)$$

The goal is to update $\boldsymbol{\Theta}$ to provide increasingly accurate predictions.
For this, we need the *Cost Function* [2]:

$$J(\boldsymbol{\Theta}) = J\left(W^{[1]}, b^{[1]}, \ldots W^{[L]}, b^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(Y^{(i)}, \hat{Y}^{(i)}\right)$$

Here, $\mathcal{L}\left(Y^{(i)}, \hat{Y}^{(i)}\right)$ is the *Loss Function* and depends on the application, i.e., how we determine the performance of our model. Essentially, we are averaging over how close our models' predictions, $\hat{Y}$, are to the known answers, Y. Now, the more accurate our model, the smaller this value $J(\boldsymbol{\Theta})$ will be. The goal is thus to update our parameters to *minimize* $J(\boldsymbol{\Theta})$.

To do this, we run through the process of *Backpropagation* using *Gradient Descent.*

- Backpropagation and Gradient Descent

We begin with a feedforward step, passing our X into our DNN for the k^{th} time, giving us $\hat{Y}^k$, our output from pass k. We then wish to update our weights and biases for all layers based on this information.

We determine the iterative change from the k^{th} to the $(k+1)^{th}$ iteration/pass-through in the weights and biases with the following equation [2, 4]:

$$\forall w \in \boldsymbol{w}_i^{[l]k}, w^{k+1} = w^k - \frac{\alpha_k}{s}\left(\sum_{p\in\{1,\dots,s\}} \frac{\partial \mathcal{L}_{(W,b)^k}\left(Y^{(p)}, \hat{Y}\left(X^{(p)}\right)\right)}{\partial w^k}\right)$$

$$b_i^{[l](k+1)} = b_i^{[l]k} - \frac{\alpha_k}{s}\left(\sum_{p\in\{1,\dots,s\}} \frac{\partial \mathcal{L}_{(W,b)^k}\left(Y^{(p)}, \hat{Y}\left(X^{(p)}\right)\right)}{\partial b_i^{[l]k}}\right)$$

In the above equations, $\boldsymbol{w}_i^{[l]k}$ is the vector of weights associated with the i^{th} neuron of the l^{th} layer during the k^{th} feedforward iteration, and $b_i^{[l]k}$ is the bias of the same neuron in the same iteration. We add a term, s, which is the *Batch Size*, or the number of instances to compute before updating the parameters in an iteration. If $s = 1$, we have simple *Stochastic Gradient Descent*. If it is more than 1, we have *Batch Gradient Descent*.

Writing out the algorithm, we begin by taking s instances of data X, labeled $X^{(p)}$ in keeping with our convention of using parenthesis to indicate a data instance, calculate the loss for each, determine the contribution each parameter (weights and bias) contributes to the loss for that data instance, sum over those contributions, and then divide by s to get the average loss contribution by that parameter for that batch. To avoid overshooting the minimum, we include a decay term, the *Learning Rate*, α^k, which will typically decrease as the number of iterations increases. From here, we update each weight vector and bias of the neuron by taking their previous values and subtracting the average contribution to the loss for the batch of the weights and bias of that neuron (which we just calculated) multiplied by our learning rate.

Thus, we wander around the $\mathcal{L}_{(W,b)^k}$ function until we find a local minimum, which, from above, will mean that our $\hat{Y}$ is as accurate as can be.

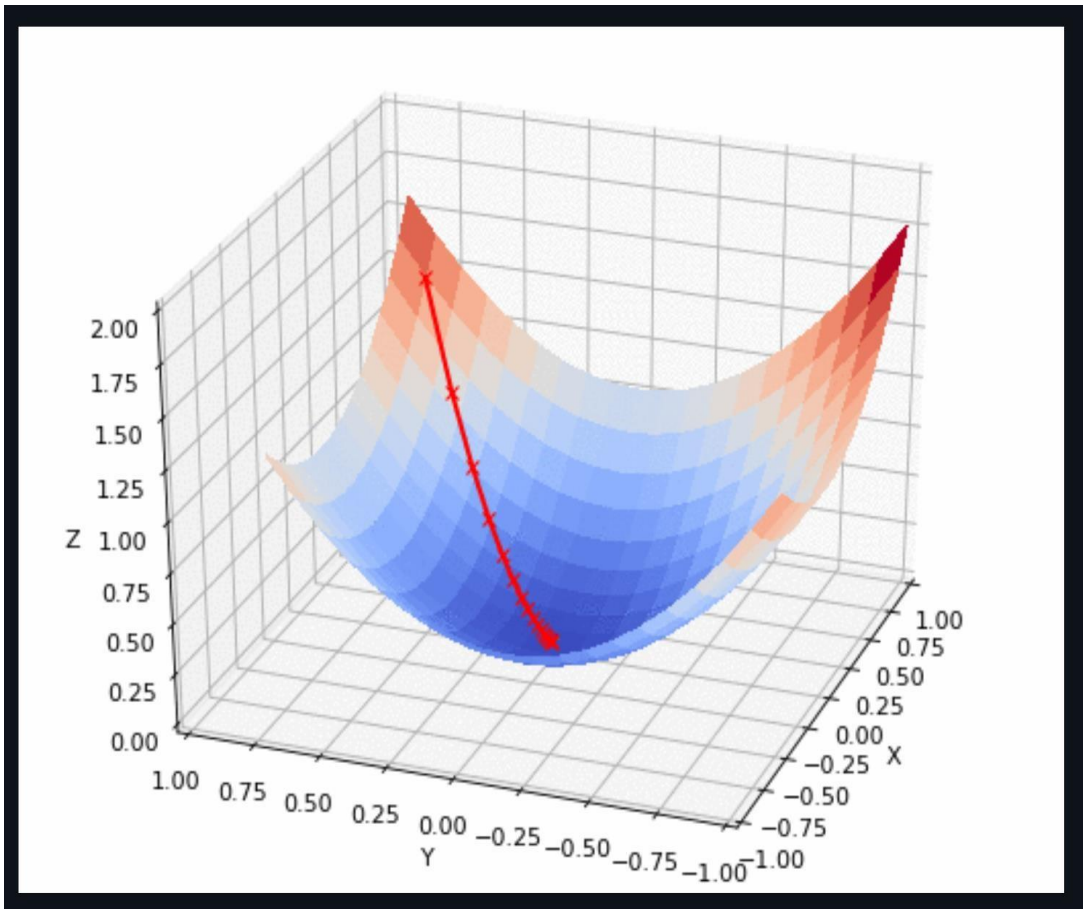
Though we have chosen to write the update equations in the above manner for complete transparency about what is happening inside them, a simpler formulation will aid us when we discuss optimizations to our update equations [4]:

$$g^k = \frac{1}{s} \sum_{p \in \{1,\ldots,s\}} \nabla \mathcal{L}_{(W,b)^k} \left(Y^{(p)}, \hat{Y}\left(X^{(p)}\right) \right)$$

Then, our update equation can be written more compactly as [4]:

$$(W,b)^{k+1} = (W,b)^k - \alpha_k g^k$$

Visually, we have the following, where the contour function is our $\mathcal{L}_{(W,b)}$ [5]:



In this image, we have the loss function, and we are attempting to find the bottom, the deep-blue region. Our steps are the red x marks, and we see that we choose to step *down* (hence the negative in our update equations) in the direction of steepest slope. Also, we notice that the steps become smaller, a result of our α^k, to prevent us from accidentally beginning to climb up the other side.

C. Primer on Reinforcement Learning

i. Foundations

Reinforcement Learning (RL) lies between *supervised* and *unsupervised* learning. In supervised learning, the algorithm has the correct answers (labels or values associated with each data instance) and seeks to learn patterns in the data to reproduce those answers. In unsupervised learning, the algorithm recieves unlabeled data and seeks to, depending on the specific structure of the algorithm, determine its own way of differentiating the data. In RL, the algorithm, known as an *agent*, is placed in an environment with a goal and a reward/penalty associated with its actions. It learns how to maneuver in its environment by seeking rewards and avoiding penalties. So, we do not tell it how to move/interact with the environment (as we might not actually know how to do that ourselves), but we do judge its behavior and give it the ability to learn from its mistakes. In this way, after spending enough time in its environment, the hope is that it has learned how to achieve the goal we set out for it.

For this primer, let s be a state, i.e., the realization of the environment at a given time t. Let a be an action the agent can take in state s, and let $R \in \mathbb{R}$ be a reward associated with that action. We use the following general notation for a *Reward Function*, or the reward received at time $t+1$ [1,2]:

$$R_{t+1} = R_{t+1}(S_t, A_t, S_{t+1})$$

We note that in general, the mathematical set-up of rewards has the reward arriving at $t+1$, as the agent finds itself in state $S_t = s$ at time t, takes an action $A_t = a$, then the environment responds by changing to state $S_{t+1} = s'$ and giving a reward. So, we talk about the step $t-1$ to t or t to $t+1$, as both appear in the relevant literature. It is also written as: r_{t+1} or $r(s_t, a_t)$. In this last version, it is understood that at the state and action are at time t but the reward will come at time $t+1$.

- Value and Policy Functions:

The goal is to develop a proper *Policy Function*, a set of rules which tell the agent how to act in each state. It is written [3]:

$$A_t = \pi_t(S_t)$$

To evaluate our policy function, we need to know the value of a state under said policy. This is the *Value Function* [3]:

$$V^{\pi}(S_t)$$

- Environment:

For many tasks, it is not necessary to know exactly *how* the environment ended up where it is, but only to have an idea of where it will be next. This simplifies the information we need to feed to our agent. In situations where we *do* need to know what the environment was like several time steps into the past, we develop our model of the environment to subsume past time-steps' information into the current time step. By doing so, we model our environment with *Markovian*

dynamics, where the probability of transitioning into a next state depends on only a few states in the past. This looks like [4]:

$$p(s_t|s_{0:t-1}) = p(s_t|s_{t-K})$$

When $K = 1$, we have simply that the probability of the next state is determined entirely by the previous one, giving us a *Markov chain* [4].

- Markov Decision Processes

A *Markov Decision Process* (MDP) is defined by [5]:

$$T = \{t_0, \dots, t_n\}$$

$$\{\mathcal{S}, \mathcal{A}(s), p(s'|s,a), \mathcal{R}, \gamma\}$$

In this formulation, $\mathcal{S}$ is the set of all possible states, $\mathcal{A}(s)$ is the set of all actions realizable for a given state, $p(s'|s,a)$ is the probability of transitioning to state s' given s and a, and $\mathcal{R}$ is the set of all possible rewards.

We can also write our total, cumulative reward earned as [6]:

$$R_{cumulative} = \sum_{t=0} \gamma^t R_{t+1}(S_t, A_t S_{t+1},) = R_1(s_0, a_0, s_1) + \gamma R_2(s_1, a_1, s_2) + \gamma^2 R_3(s_2, a_2, s_3) + \cdots$$

We introduce the γ factor for two reasons. First, it allows us to calibrate how the agent should prioritize rewards. If the agent needs to seek better rewards sooner, then γ will be small. A value of $\gamma = 0$ means our agent *only* looks to the next time step's reward. A value of $\gamma = 1$ would weight each time step's reward equally. The second reason to introduce γ is in case our time horizon is infinite. In that case, we set $\gamma < 1$ to ensure that our sum converges [6].

Now, at time t, we can ask for the cumulative reward from then on, given by the *Return Function* [7]:

$$G_t = \sum_{i=0}^{T-t-1} \gamma^i R_{t+i+1}(S_{t+i}, a_{t+i}, S_{t+i+1})$$

Now, to find the expected value of a given state at time t, $S_t = s$, we take the expectation of the cumulative rewards gained from starting in that state and following policy π. This is the *State-Value Function* [8]:

$$V_t^\pi(s) = \mathbb{E}_t^\pi \left[\sum_{i=0}^{T-t-1} \gamma^i R_{t+i+1}(S_{t+i}, A_{t+i}, S_{t+i+1}) \,|S_t = s \right] = \mathbb{E}_t^\pi [G_t | S_t = s\,]$$

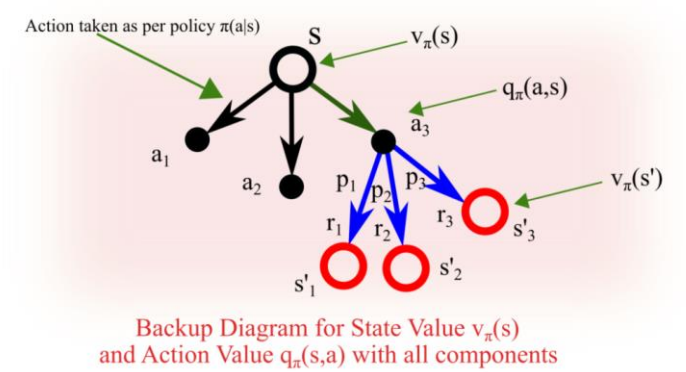
Similarly, we can specify a value of simultaneously being in state s and taking an action a as a first action while following policy π for all following actions, as opposed to always following a policy. This is the *Action-Value Function* [7]:

$$Q_t^\pi(s,a) = \mathbb{E}_t^\pi\left[\sum_{i=0}^{T-t-1} \gamma^i R_{t+i+1}(S_{t+i}, A_{t+i}, S_{t+i+1}) \,|S_t = t, A_t = a\right] = \mathbb{E}_t^\pi[G_t|S_t = s\,, A_t = a]$$

We see that it is essentially the same as the state-value function, but we have an extra specified first action a, after which we follow our policy π. This is captured by the following relationship: [7]

$$V_t^\pi(s) = \mathbb{E}^\pi[Q_t^\pi(s,a)] = \sum_{a\in\mathcal{A}} \pi(a|s) Q_t^\pi(s|a)$$

A nice visual that showcases how we begin in a state s, take an action a based on policy $\pi(s)$, receive a reward r, and transition to a new state s' is called a *Backup Diagram* [9]:



Backup Diagram for State Value $v_\pi(s)$ and Action Value $q_\pi(s,a)$ with all components

- Bellman Equations:

We notice that when $i = 0$ in either of the action-value or state-value functions, we have only the expected value of the first reward and then the expected value of the other time steps. Similarly, with G_t, we can write it as $G_t = R_{t+1} + \gamma G_{t+1}$. This allows us to write the two equations as depending on all future evaluations iteratively [10, 11]:

$$V_t^\pi(s) = \mathbb{E}_t^\pi[R_{t+1}(s,a,s')] + \gamma\mathbb{E}_t^\pi[V_{t+1}^\pi(s')] = \mathbb{E}_t^\pi[R_{t+1} + \gamma G_{t+1}|S_t = s]$$

$$Q_t^\pi(s,a) = \mathbb{E}_t^\pi[R_{t+1}(s,a,s')] + \gamma\mathbb{E}_t^\pi[V_{t+1}^\pi(s')] = \mathbb{E}_t^\pi[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a]$$

We notice that in our action-value function, the state-value shows up, as well. We fix this soon. So, our evaluation of state s depends on our evaluation of s', and so on, a concept known as *bootstrapping*.

To find the *optimal* equations, the *Bellman Optimality Equations*, we look for the best policy [12, 13]:

$$V_t^*(s) \coloneqq V_t^{\pi_*}(s) = \max_\pi V_t^\pi(s), \qquad \forall s \in S$$

$$Q_t^*(s,a) \coloneqq Q_t^{\pi_*}(s,a) = \max_\pi Q_t^\pi(s,a), \qquad \forall s \in S$$

Since the only difference between the action-value and state-value functions is that we have a first action in the action-value formulation, if we pick the best action for our first action, then we have the best value. This leads to the relationship [12]:

$$V_t^*(s) = \max_a Q_t^*(s,a)$$

From this, we rewrite the Bellman optimal action-value function as [12]:

$$Q_t^*(s,a) = \mathbb{E}_t^*\left[R_{t+1}(s,a,s') + \gamma \max_a Q_{t+1}^*(s',a')\right]$$

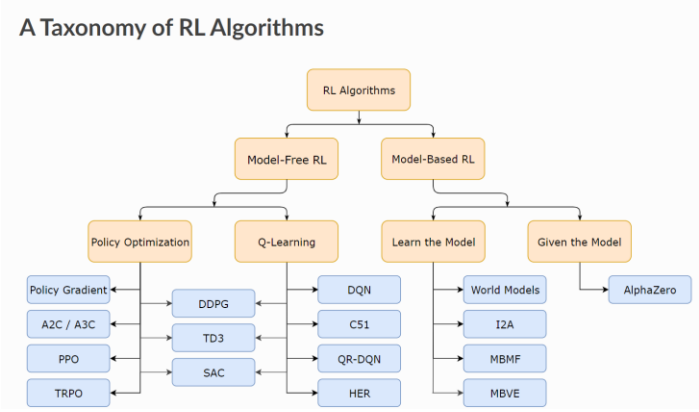
We also rewrite the optimal state-value function as [14]:

$$V_t^*(s) = \max_a \mathbb{E}_t^*[R_{t+1}(s,a,s') + \gamma V_{t+1}^*(s')]$$

Note, if we want to consider these as *time-independent*, we simply remove the t index and think of our system as evolving over states instead of linearly through time.

Reinforcement Learning is thus primarily focused on finding or approximating the optimal policy, π_*, to maximize our reward (in the form of maximizing our Q or V functions). To do this, we broadly divide our strategies into *Dynamic Programming*, *Monte Carlo Methods*, or *Temporal Difference Learning*, depending on how much information we have about the environment, or a *model* [15].

A nice map follows [16]:



ii. Model-Based Algorithms

- Dynamic Programming

Dynamic Programming (DP) refers to a set of algorithms used to compute optimal policies given a *perfect* model of the environment as a finite *Markov Decision Process* (MDP). Recall the form of an MDP:

$$T = \{t_0, \dots, t_n\}$$

$$\{\mathcal{S}, \mathcal{A}(s), p(s'|s,a), \mathcal{R}, \gamma\}$$

In DP, we first begin with *Policy Evaluation*, or evaluating how well each policy does, given our actions and states. Given the finite nature of the MDP, this can, in principle, be done. In the literature, this is often shown as a tabular method, where a value under a policy is assigned to each state-action pair. Thus, we run through V^{π_0}, V^{π_1} ... and see the consequences for V for each policy [17].

From here, we ask if, though we know the value of an action for a given state, can we switch policies as we move through the states, instead of relying on one policy for all states? We are thus asking if we can choose a in state s such that $a = \pi'(s) \neq \pi(s)$. This is *Policy Improvement*. To check this, we look to see if the following action-value and state-value inequality is true [18]:

$$Q^{\pi}\big(s, \pi'(s)\big) \geq V^{\pi}(s)$$

If this is so, then it is better to switch to policy $\pi'(s)$ once we are in state s. This leads us to an important point. We ask this at *each* state, giving us the following equation [19]:

$$\pi'(s) := \max_a Q^{\pi}(s,a)$$

Essentially, at *each step, we are looking for the best policy in that state*. This is termed *greedy*, because we are not interested in subsequent steps, only the most current one. However, it can be shown that by choosing greedily at each step, we will arrive at the best policy possible (unless our original was optimal).

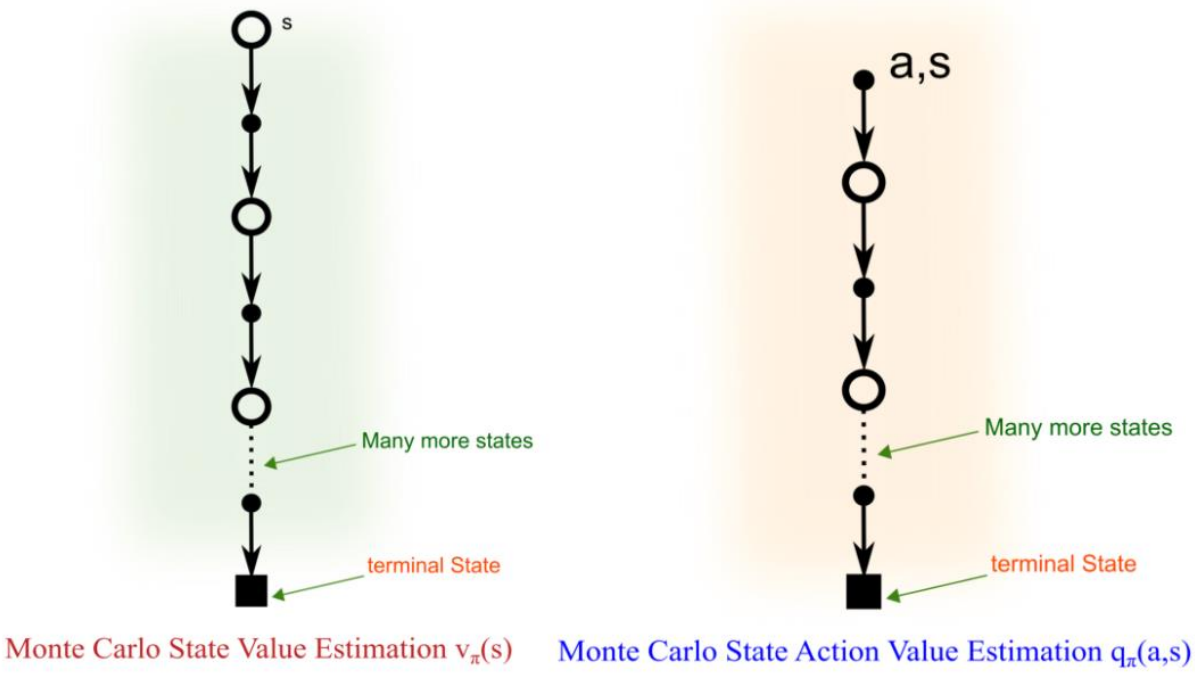
Finally, we combine both to arrive at *Policy Iteration*, wherein at each step, we evaluate that policy π with V^{π}, then improve it to a better policy π', then evaluate that with $V^{\pi'}$, and so on [20].

iii. Model-Free Methods

- Monte Carlo Methods

To begin, a significant departure from DP and *Monte Carlo* (MC) methods of solving for the optimal policy is that in MC, we *do not assume we perfectly know the environment*. We are missing the $p(s'|s,a)$ from our original MDP formulation. MC methods only require *experience*, or samples of sequences of states, actions, and rewards, often from simulations (hence the moniker *Monte Carlo*). Another difference is that in using MC methods, we talk about *episodes*, or finite sequences of states which must terminate, and updates occur after each episode based on

the *average* as opposed to *expected* returns. The types of evaluations and updating described in the DP sections have their counterparts here, so we will not delve into detail about them [21]. Two backup diagrams for simple MC methods of state-value and action-value sampling follow: [9]



Monte Carlo State Value Estimation $v_\pi(s)$ Monte Carlo State Action Value Estimation $q_\pi(a,s)$

However, there is one important aspect of MC methods that *does* need to be introduced, as it is important for the models used in this paper. Recall the policy improvement algorithm introduced in the DP section. To choose a better policy, we act greedily, choosing the best action in that state under a given policy. However, if we do not have all the state-action-value pairs, we might miss a good action. Another issue is that to generate a sample, the agent follows a policy, but there is no guarantee that all state-action pairs will be visited to evaluate them. We need a method for our agent to *explore*, to visit state-action pairs that it isn't likely to, as they might end up being better in the long run. For this, we introduce *ϵ-greedy policy* [22]:

$$a = \pi_{\epsilon-greedy}(s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & for\ a = \max_a Q(s,a) \\ \frac{\epsilon}{|\mathcal{A}|} & otherwise \end{cases}$$

With an *ϵ-greedy policy*, the agent takes the action with the best Q-value with probability $1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|}$ and any other action with probability $\frac{\epsilon}{|\mathcal{A}|}$. As the agent learns over time, the value of ϵ can be reduced to reduce the amount of exploring the agent does.

- Temporal Difference Learning:

Temporal Difference (TD) learning can be seen as integrating both earlier methods, able to learn from experience without a well-defined model of the environment through sampling, as in MC methods, as well as being able to update estimates using bootstrapping, as in DP. TD is a large and active area of research, with many more methods than can fit in this paper, and so we move quickly to the model outline that we will use in this paper [15].

We begin with the *Update Equation for TD* [23]:

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

This formulation is called $TD(0)$. We can also define the *Difference* or *Error* as [24]:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(s)$$

Then, our update equation above becomes:

$$V(S_t) = V(S_t) + \alpha\delta_t$$

Now, we need to mention two kinds of TD algorithms, *On-policy* and *Off-policy*. On-policy algorithms assume that the policy being used to produce a dataset (while exploring, for example), is also the optimal policy, and the task is therefore to learn the optimal policy function from the generated data. However, off-policy does not assume this, but merely seeks to learn the optimal policy when data is collected from a different (possibly random) policy [25].

We introduce an on-policy algorithm, *SARSA*, with the objective of contrasting it to the algorithm we will use from now on [26]:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

We see that this is the same equation as our TD update equation, but now focusing on actions, as we want to maximize the return by choosing the right actions. In this model, we need $S_t, A_t, R_{t+1}, S_{t+1}$, and A_{t+1}. Hence the name SARSA. With this model, the agent explores with an ϵ-greedy policy, producing an action a, and then uses it *again* to produce the next action, a' [27].

Now, we move to the last part of this primer, *Q-Learning* [27, 28]:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{A_{t+1}} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

We see the only difference between SARSA and Q-Learning is the inclusion of the max operator when choosing the action to take at the next state. However, this means that, though our agent might explore, if that exploration is not the optimal action to take in that state, the agent will choose something different to learn as the policy. From this, the agent can learn an optimal policy from a non-optimal data generation or exploratory policy through maximizing Q-values.

5. Technical Indicators, Data Processing, and Principal Component Analysis

For this section, we will describe how we analyze each hour's worth of Bitcoin data and how we consider how an agent should act in that hour to maximize a portfolio consisting of Bitcoin.

A. Technical Indicators

When stock markets are represented in media, they often look quite confusing. One might see several timeframe's worth of *Candlestick Charts*, or blocks with whiskers representing the *High Price*, *Open Price*, *Close Price*, and *Low Price* for that time slice [1].

For hourly Open, Close, High, Low, and Volume Bitcoin data, we used *Crypto Data Download*, a repository of several exchanges' cryptocurrency data. Due to the availability of several coins' data in USD and in daily, hourly, and minute slices, we chose *Bitstamp* as the exchange we took data from [2].

Sometimes, one might see something a bit more complicated:

What are all the lines and graphs on the bottom and top of the image? Those are *Technical Analysis Indicators* (TA), or heuristic/pattern-based analyses of the asset based on price and volume. The general idea is to identify how the asset is behaving with the goal of predicting where it will go [3].

Technical indicators can be classified into several different categories, based on what specifically they seek to quantify and/or expose for the investor. Daily FX gives a brief overview of four categories: *Trend Indicators* seek to identify trends in the asset's price, *Oscillator/Momentum Indicators* show how price momentum is developing (with the idea that an asset generally continues in the direction of momentum for a while, much like physical objects), *Volatility Indicators* measure how long the upswings and downswings of an asset are, and *Support/Resistance Indicators* indicate prices that form "barriers" on the asset, or upper and lower bounds the price is unlikely to break or has not passed in some time with the idea that if the price *does*, then that is a strong trading signal [4].

In this project, we use *TA-Lib*, a python package that has dozens of built-in functions to calculate indicators. The TA-Lib package has slightly different categories than mentioned above, but they are the same indictors [5].

The most obvious question now might be: "Which is the best?" Another, related question, is: "On what *timeframe* should we look?" Most indicators can be fine-tuned to look over a shorter or longer timeframe, with the choice of what to focus on depending on the asset and the trader's goals. For example, the *Moving Average* can be calculated with any number of past hours (in this case) the analyst wants. So, we can choose to give our algorithms several different iterations of the moving average. However, this makes the number of possible analyses by technical indicators prohibitively large and makes answering either of the above questions difficult. This project hopes to circumvent this difficulty in favor of a broader approach. For any given asset on any given timeframe, it is often the case that traders will use a combination of several different indicators, aggregating the information in hopes that their aggregate will lead to better decisions than any individually. This project does something similar.

For our given asset, Bitcoin, we have chosen at least one indicator from each category and, where feasible, are analyzing the hour with respect to each indicator over several timeframes. We chose indicators based on their popularity and ease of understanding, though we admit that the whole space (or really, even a large part of it) is not explored. This could form the basis for further analysis.

The chosen technical indicators are: *Bollinger Bands, Aroon, Aroon Oscillator, Relative Strength Index, Normalized Average True Range, Exponential Moving Average, Weighted Moving Average, Stochastic Oscillator, Stochastic Fast, Moving Average Convergence Divergence, Medium Price, Accumulation/Distribution, Hilbert Transform-Dominant Cycle Period,* and *Hilbert Transform Phase of the Dominant Cycle*. For each of these where the timeframe can be specified, we chose a large range to hopefully capture short-term and long-term information. With these, we add the *High, Low, Close,* and *Volume* for that hour. For brevity, we point the curious reader to Investopedia for a detailed description of each. In total, this means that for each hour of Bitcoin data, we have 101 features.

A single row (call it the i^{th} row) or hour of our technical indicator data matrix X_{TA} now looks like:

$X_{TA_i} = [$*BB_upper 10, BB_middle 10, BB_lower 10, Aroon_down 10, …*

Med Price, Chaikin A/D Line, HT Dom Phase, HT Dom Period, High, Low, Close, Volume]

A table of the indictors, their classification, and the timeframes on which they are calculated follows:

Overlap Studies	**Timeframes in hours**
Bollinger Bands	10, 12, 14, 16, 18, 20
Exponential Moving Average	20, 25, 30, 35, 40
Weighted Moving Average	20, 25, 30, 35, 40
Momentum Indicators	
Aroon	10, 12, 14, 16, 18, 20
Aroon Oscillator	10, 12, 14, 16, 18, 20
Moving Average Convergence Divergence	6, 9, 12, 15, 18, where the Fast Period is these values, the Slow Period is twice these values, and the Signal Period is $floor(1.5 * value)$
Relative Strength Index	10, 12, 14, 16, 18, 20
Stochastic Oscillator	1, 3, 5, 7, 9 and 3, 5, 7, 9, 11 (There are two components to this one)
Stochastic Fast	1, 3, 5, 7, 9 and 3, 5, 7, 9, 11 (There are two components to this one)
Volume Indicators	
Chaikin A/D Line	N/A
Volatility Indictors	
Normalize Average True Range	10, 12, 14, 16, 18, 20
Price Transformations	
Median Price	10, 12, 14, 16, 18, 20
Cycle Indicators	
Hilbert Transform- Dominant Cycle Period	N/A
Hilbert Transform- Dominant Cycle Phase	N/A

B. Principal Component Analysis

Now that we have our data matrix of technical indicators and our labels, we ask if *all* the indicators are necessary? We expect there to be significant correlation between some indicators or the same indicator on different time frames. To discover such correlations, we use *Principal Component Analysis* (PCA).

To do this, we use *Singular Value Decomposition* to decompose the matrix X_{TA} of p variables (our technical indicators, high, low, close, and volume) into:

$$X_{TA} = USV^T$$

Here, S is a diagonal matrix of the square root of the eigenvalues of $X_{TA}{X_{TA}}^T$ or ${X_{TA}}^T X_{TA}$. We call these $\sqrt{\lambda_i}$ and order them in descending order. V is a matrix whose columns give the coefficients for a linear combination of the original variables to create p *uncorrelated* variables. U is a matrix whose columns are the values of the uncorrelated random variables created by linear combinations defined by the columns of V divided by $\sqrt{\lambda_i}$. We then truncate our original $X_{TA_k} = US_kV^T$. Here, we choose the first k eigenvalues to reduce the complexity of our X_{TA}, as S_k now only has diagonal entries up to k, and 0 elsewhere.

This X_{TA_k} will be the best $rank\ k$ approximation to the original X_{TA}.

We then take the first k columns of V, and multiply them by X_{TA} to get:

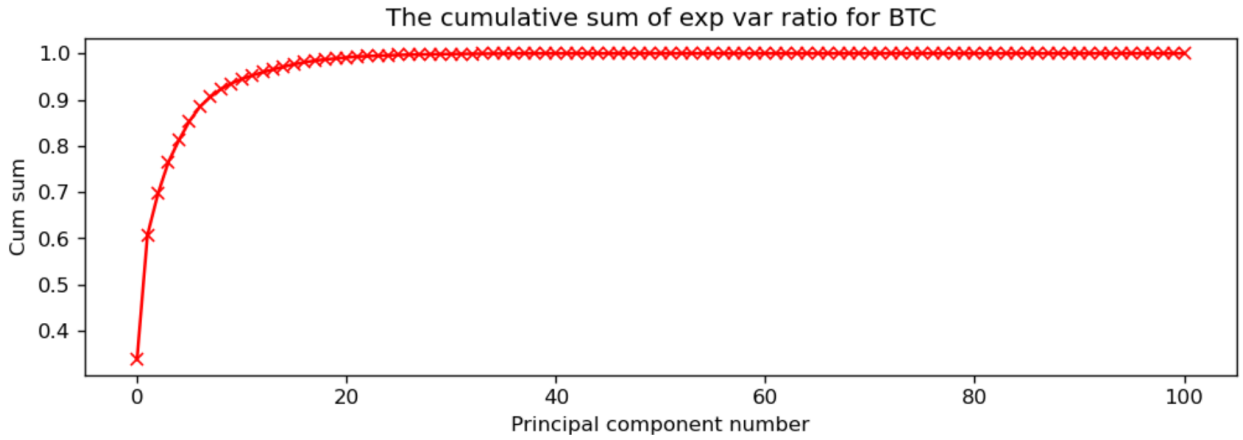
$$T = X_{TA}V_k$$

This T, which we refer to as X_{PCA}, is $n \times k$, preserving the number of data instances but with only k features each.

We quantify the percent of variation of the original matrix captured by our k *Principal Components* with the following equation:

$$s_k = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{p} \lambda_i}$$

A graph of this looks like the following:



We see that by the first 20 principal components, we have captured almost all the variance, and we gain very little by having more. It is easier to train our models with only a few uncorrelated variables, rather than 101 correlated ones. To avoid *look-ahead bias* in training, we perform PCA on our training set, X_{train} to get V_k, and then apply V_k to our validation and testing sets like we did to get $X_{PCA_{val}}$ and $X_{PCA_{test}}$ above.

C. Labeling Scheme

Now that we have our features for each time slice i, how do we evaluate that time slice? We do it in two related ways, one for classification algorithms and one for regression. For the decision-making agent we wish to create, we are simply interested in deciding what action to take based on whether we believe the price will increase or decrease by the next hour.

For classification, we look *one* hour into the future to see how the asset behaved between them. We then label the slice 1 if the price rose and 0 if it fell or remained the same. These correspond to the actions *Buy* and *Sell*. If at time $t + 1$ the price was higher than at time t, then the correct action to take is to buy at time t, so hour t will receive a label of 1, for example. To label hour $t + 1$, we will look at hour $t + 2$, and so on.

For regression, we do mostly the same, giving each hour the value of the *Percent Change* from that hour and the one immediately after.

Thus, any hour t that sees a price growth for $t + 1$ is labeled 1 for classification and receive the numerical value of the percent change, which will be positive, for regression, and vice versa if the price fell.

Though some exchanges have a fee per transaction, which might limit the number of trades we make or the threshold for price change needed to enter or exit a trade, several online and popular ones (such as Robinhood, for example), do not. So, there is no penalty for trading often. As such, we are only interested in whether the price will rise *at all* or whether it will fall *at all* one hour into the future. With fees, a small change might not be enough to act upon, as one might pay more in making the trade than they would earn from it. However, with fee-less trading, any percent gained is profit.

6. Algorithm Training and Performance

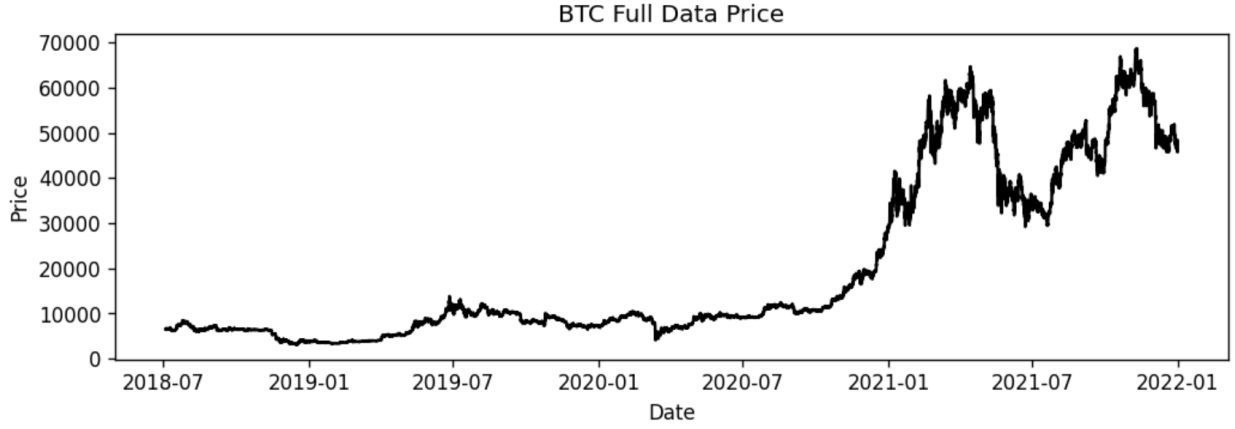
A. Classical Statistical Methods and eXtreme Gradient Boosting

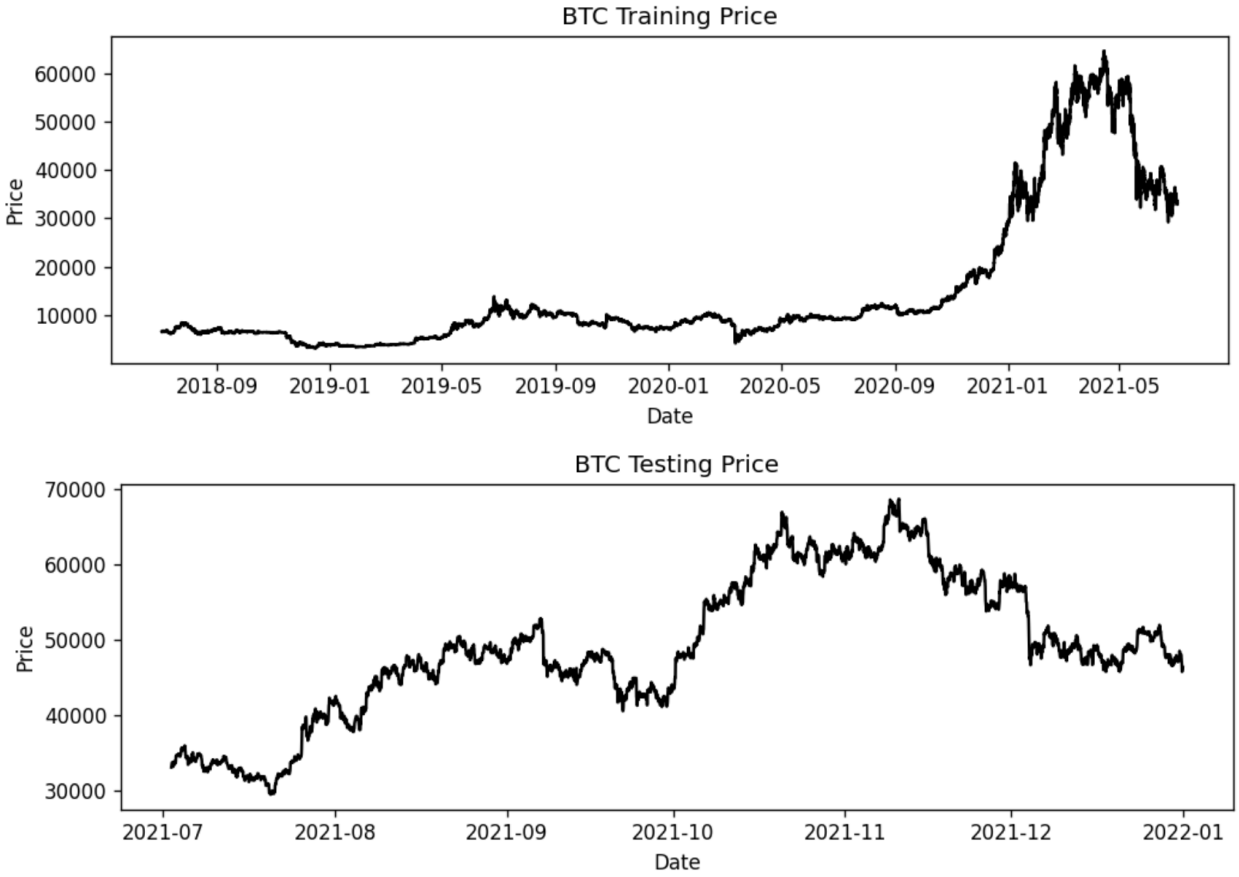
i. Exponential Smoothing

To begin, will do some exploration of our data. We plot BTC's *closing* price (henceforth to be simply called *price*) during all the data we have, then only the portion we use for training, and finally, the testing portion, which we have chosen to be about six months, or 4380 hours. The full data set runs from July 1st of 2018 at 12:00 AM to December 31st of 2021 at 11:00 PM.

However, because we need many hours to create certain TA in the feature matrix (for example, to calculate the EMA for 12 hours would require 12 hours of data for the first usable EMA value), after transforming our raw price and volume data into our 101-rowed X_{TA}, our dates are from July 3rd of 2018 at 3:00 PM to December 31st of 2021 at 10:00 PM. We lost several days in the beginning at their data was subsumed into the TA calculations, and the final hour was lost when we calculated the percent change and shifted it *back* one. This way, each row, or hour, has 101 TA, price, and volume features, the label, and the *future percent change*, for a total of 103 elements. Of course, upon implementing our PCA, we always drop the label and the percent change.

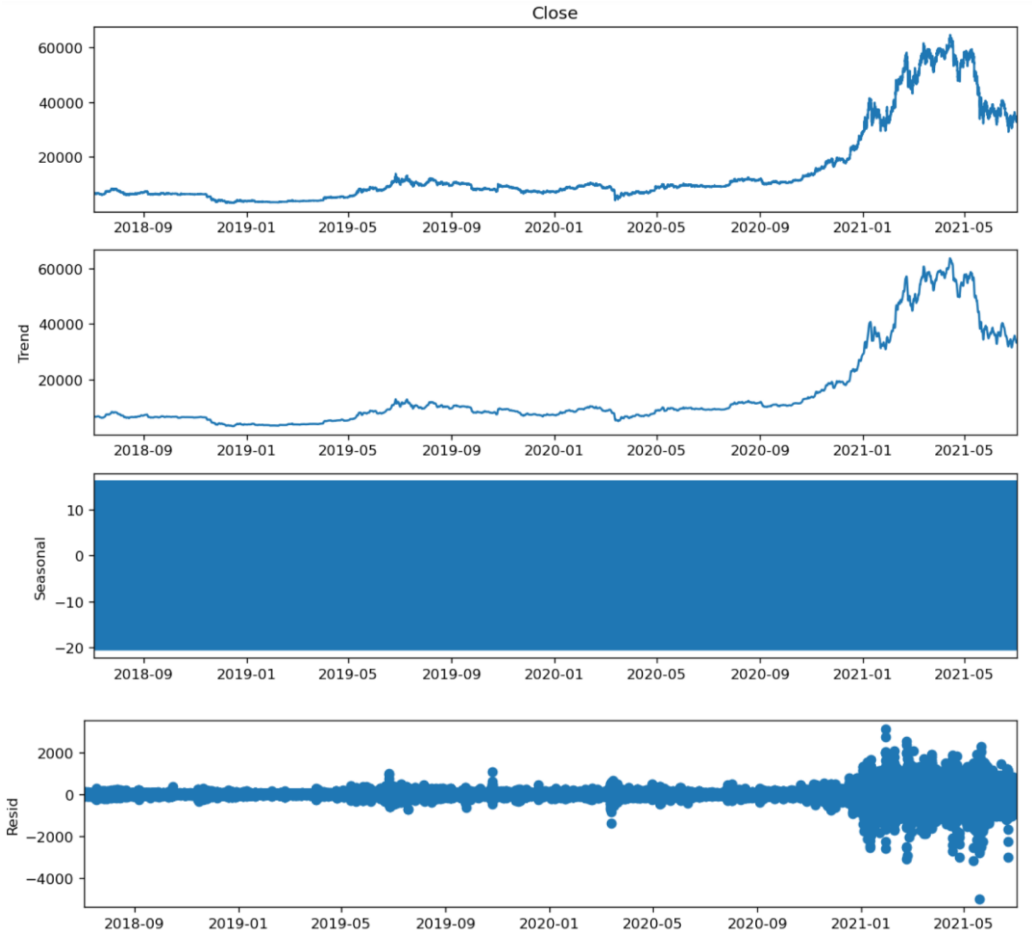
Checking when we reach 4380 hours moving backward, it is on July 2nd of 2021 at 11:00 AM. Thus, our training data for the next several algorithms is from July 3rd of 2018 at 3:00 PM to July 2nd of 2021 at 10:00 AM and our testing data is from July 2nd of 2021 at 11:00 AM to December 31st of 2021 at 10:00 PM. Here are three plots of the full (post X_{TA} matrix creation) data set's price, then the training data, then the testing.



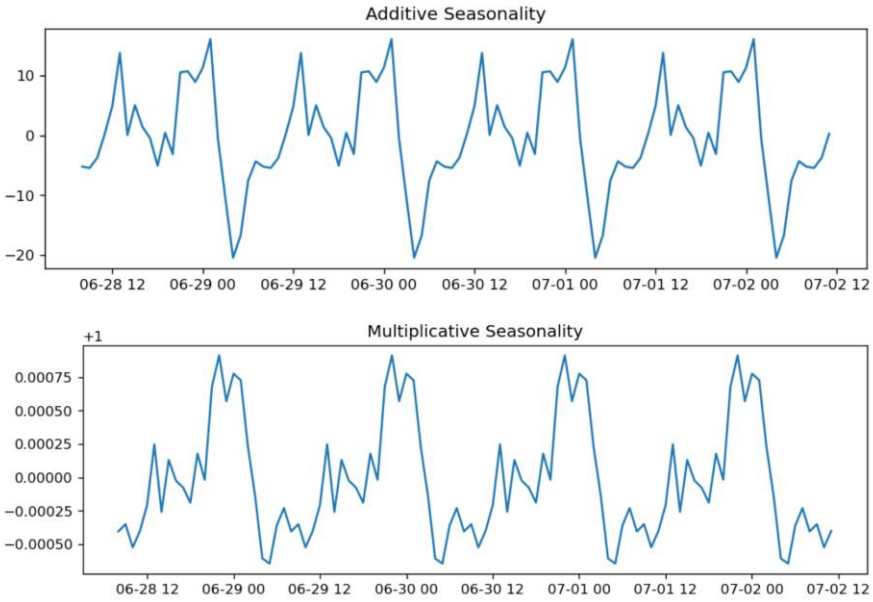


From our data, we see an overall extreme climb in price, which is interesting, as well as several large increases and drops starting after the beginning of 2021. Fortunately, as we are focusing on the last six months of 2021, we see that both our training and testing data has periods of growth and drops. Hopefully, this variation in both data sets allows the algorithms to learn how to behave in both scenarios. For all three sets, the full, training, and testing, the distribution of Buy and Sell labels is, to two decimal places, 51% and 49% respectively. This avoids class imbalances.

We decompose the training data further to determine if *Exponential Smoothing* (ETS) models are appropriate. To do so, we decompose the training data into trend, seasonality, and residuals for both multiplicative and additive seasonality, but, as these decompositions are very similar, we only put one of them here, the additive version.



We see a clear trend, but the seasonal plot is difficult to see. We show both the additive and multiplcative seasonal plot for only the first 100 hours.



We clearly see a 24-hour seasonality in both, and we choose that for the following model building.

For the classical statistical models, we run a standard parameter search through pre-made functions in the *Statsmodels* python package, which returns to us parameters and the *Sum of Squared Errors* (SSE), which we use to determine the best model, and is defined as [1]:

$$SSE = \sum_{i=1}^{n} (X_i - \bar{X})^2$$

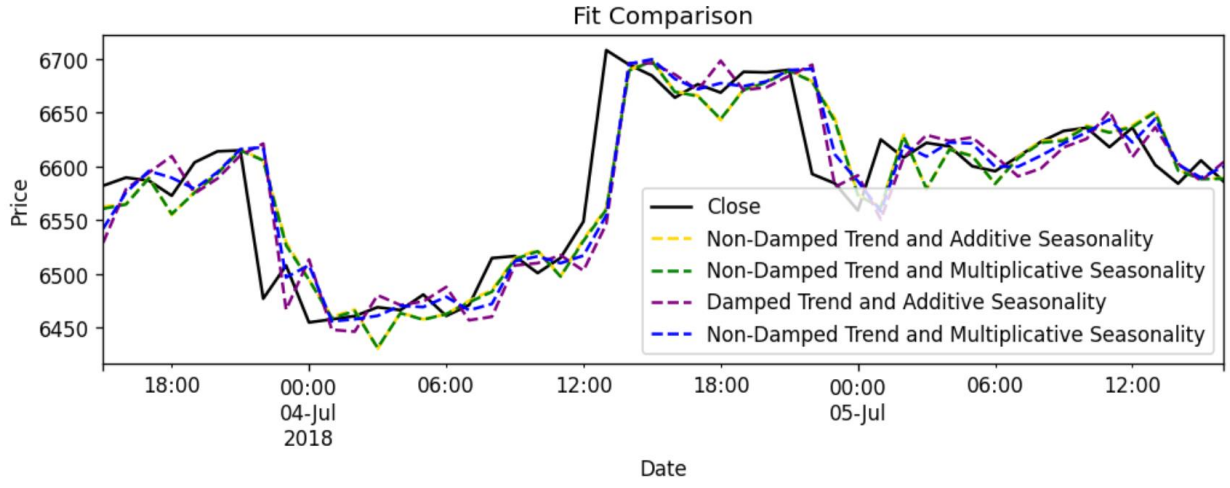
We run four different models on the BTC price data *alone* (no TA or extraneous variables), fitting for trend, which can be damped or not, as well as seasonality, which can be multiplicative or additive, using the 24-hour cycles we found above.

We get the following results:

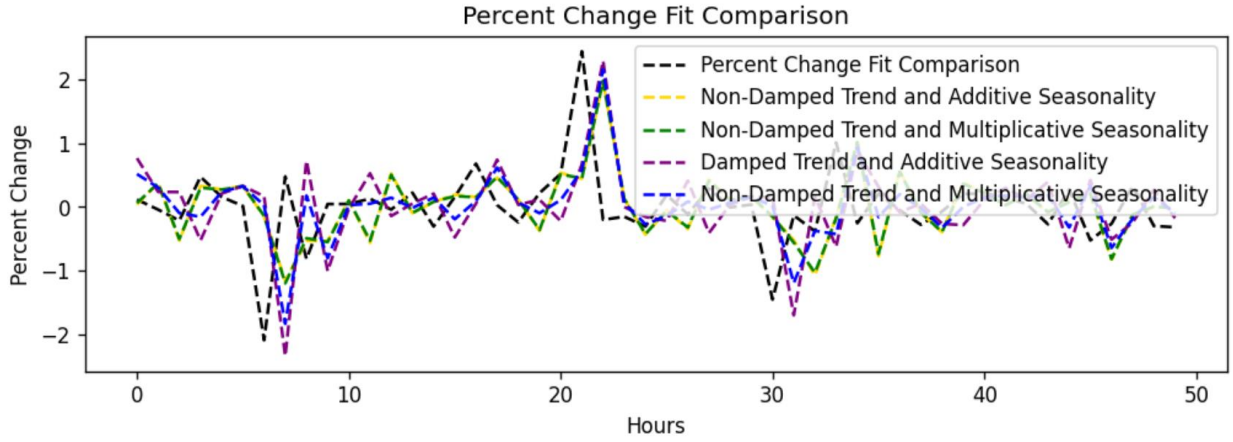
	Additive	Multiplicative	Additive Dam	Multiplica Dam
α	9.242857e-01	9.242857e-01	9.242857e-01	9.242858e-01
β	1.001181e-04	1.001395e-04	1.000193e-04	1.000202e-04
ϕ	NaN	NaN	9.900000e-01	9.900000e-01
γ	1.081633e-02	1.081633e-02	1.081633e-02	1.081632e-02
l_0	1.903727e+00	1.903727e+00	1.903728e+00	1.903727e+00
b_0	-7.629997e-07	-1.924910e-06	2.634638e-07	3.769545e-06
SSE	1.132708e+09	1.132095e+09	1.123685e+09	1.122825e+09

From comparing the SSE of each model, we see that the Multiplicative Seasonality with Dampening is the best option, as it has the lowest SSE.

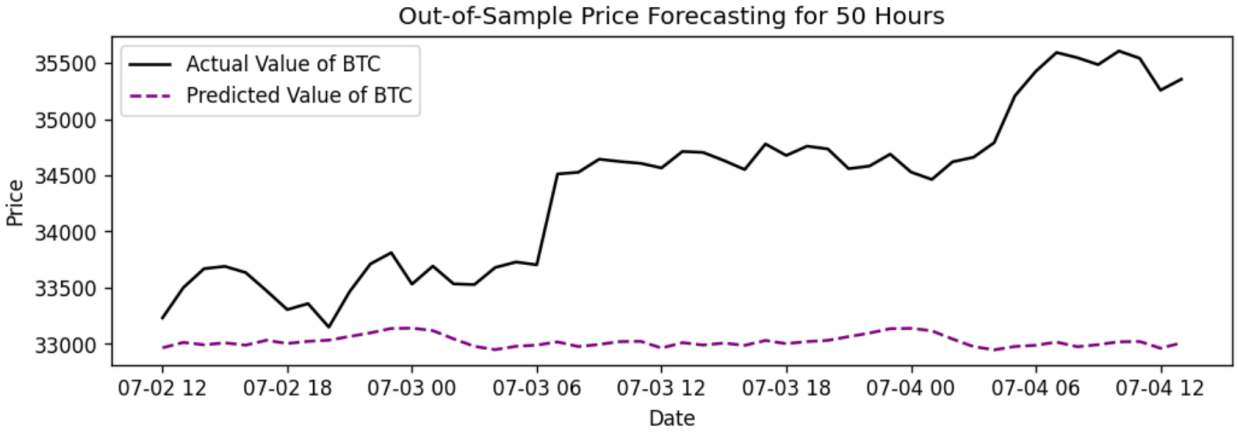
We compare the various fittings graphically. Here, we show only the first 50 hours. We see very little difference between them (which is reflected in the very similar SSE scores).

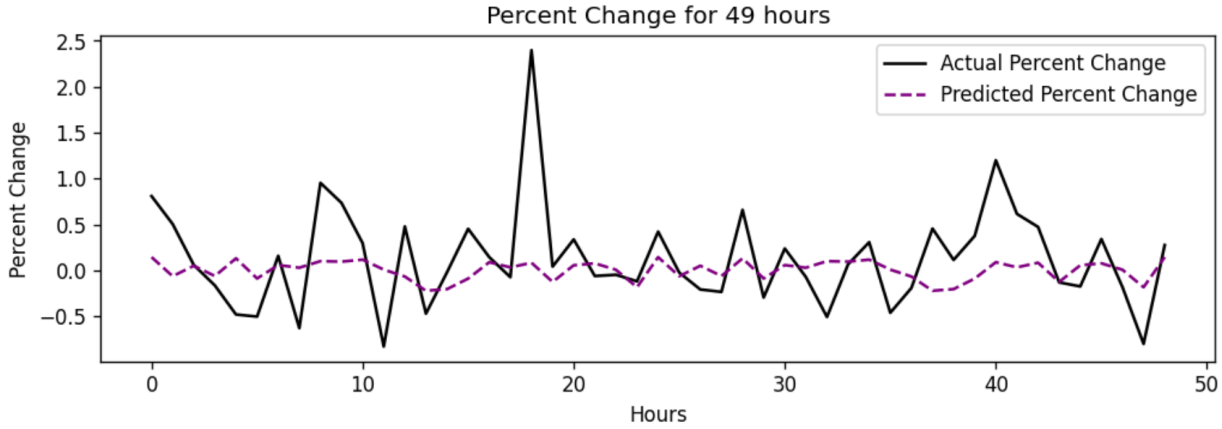


Now, we see how well our models captured the changing behavior. Again, we show only the first 50 hours. This is created by taking the percent change between successive hours and graphing them.



Now, we forecast 50 steps into the future and compare the price and price change (which, due to use having to shift backward one to get the percent changes to line up, will only be 49 values).



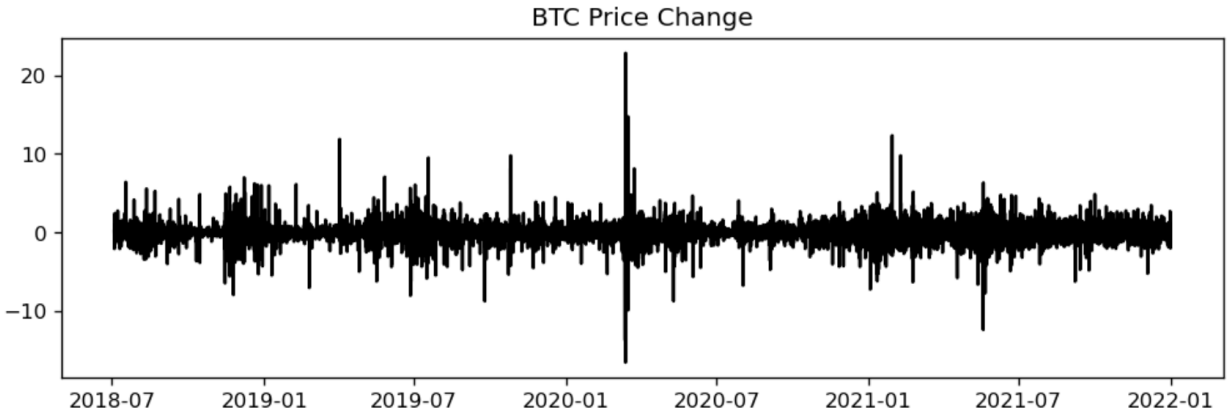


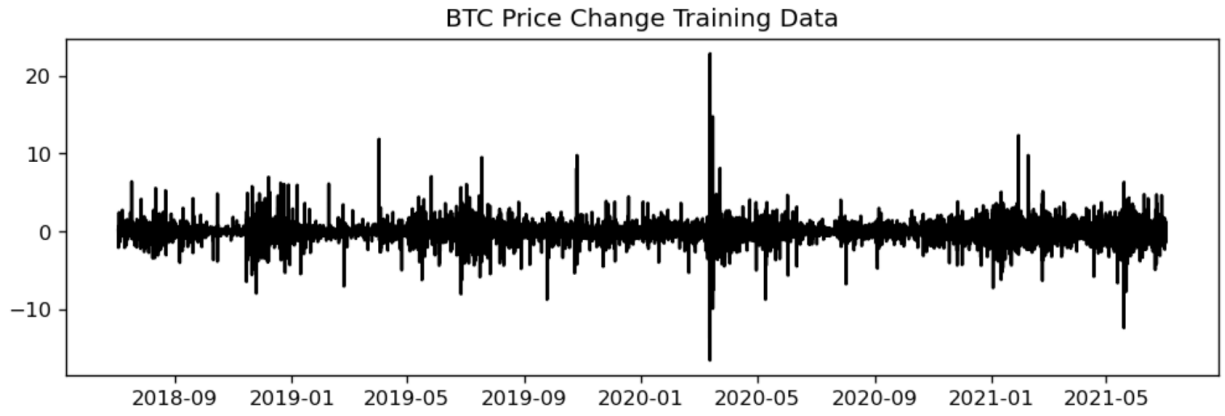
We see that our ETS model fails to capture any of the volatility, which is not a surprise, though perhaps it has found something interesting in the way the price changes. Even though the magnitudes of the changes are inaccurate, is the *direction* modeled well? Running an accuracy

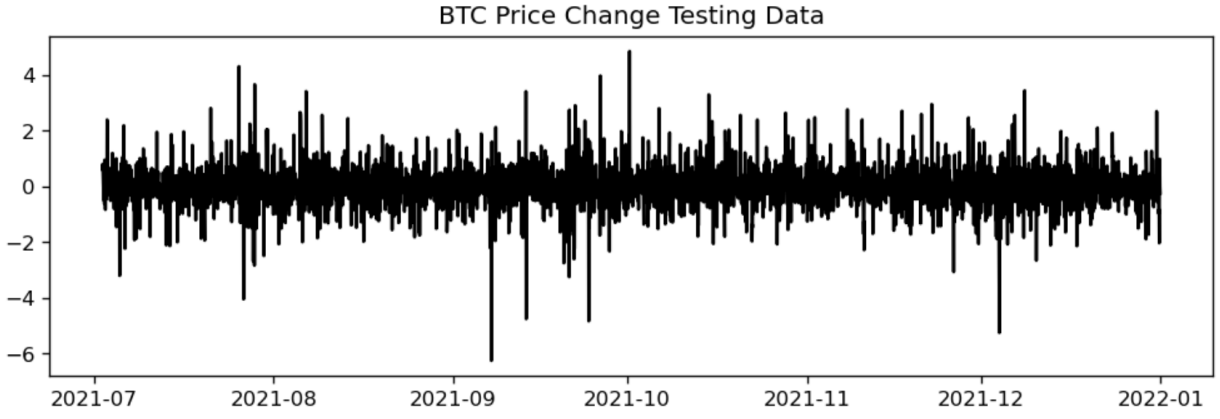
check by comparing the sign of the percent changes predicted and known, we get 50.92%. It appears not.

ii. Seasonal AutoRegressive Integrated Moving Average with eXogenous variables

For our *Seasonal AutoRegressive Integrated Moving Average with eXogenous variables* (SARIMAX), we will be attempting to model the price change directly, as we could not do so for ETS. We need to run the same checks for trend and seasonality for this time series, as well. We look at the price change.





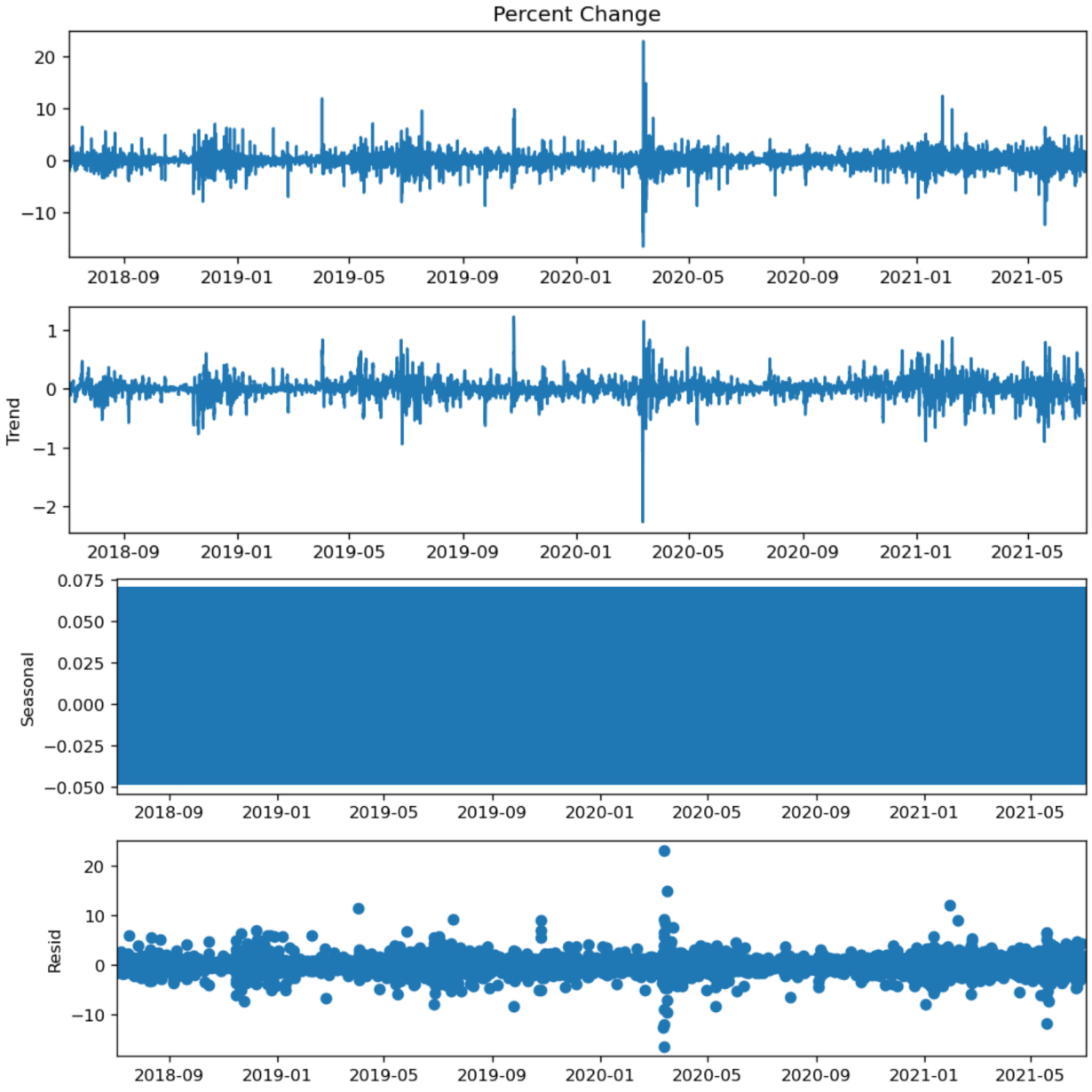


Now, we check for stationarity using an *Augmented Dickey-Fuller Test*. When we do, we get the following:

```
ADF Statistic for BTC Percent Change is -64.75818353224876
p-value for BTC Closing Values is 0.0
```

So, we treat the data as stationary, meaning that the *Integrated*, or *d*, portion of SARIMAX will be set to 0.

Checking for trend and seasonality, we note that because our percent changes can be negative, we only run an additive seasonal decomposition.



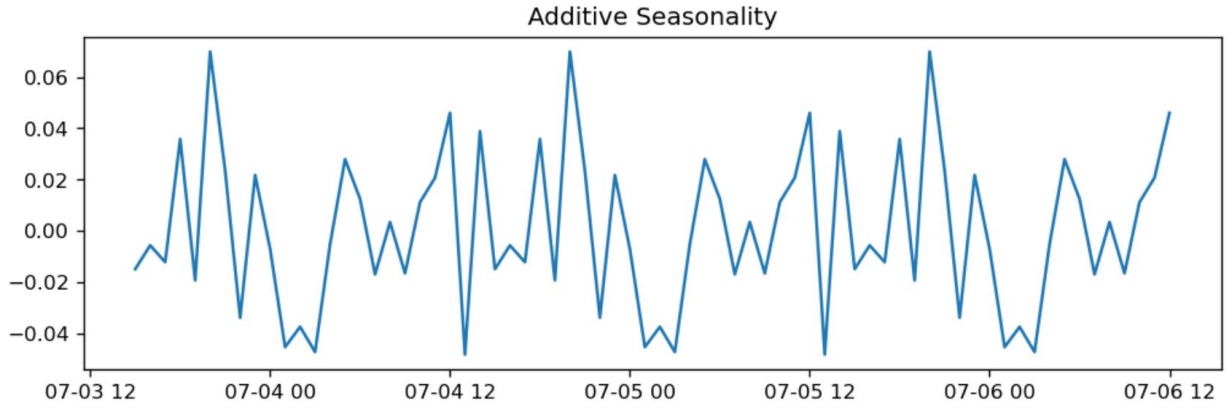
We see there is no trend, so the $A(t)$ term in our full SARIMAX equation,

$$y_t = \sum_{i=1}^{r} \beta_i x_i + u_t$$

$$\phi(L)^p \, \tilde{\phi} \, (L^m)^P \Delta^d \Delta_{\mathrm{m}}^D \, u_t = A(t) + \theta(L)^q \, \tilde{\theta}(L^m)^Q \epsilon_t$$

will be 0.

From here, we need to zoom in on the seasonal portion of the decomposition, plotting only the first 70 hours, to see it better.



We again see clear 24-hour seasonality, so the m term is 24.

We now run an *Auto-ARIMA* function from Statsmodels with *stationarity* set to 0 (indicating $d = 0$) with the exogenous variables being the TA and the endogenous (target) variable being the percent change and we find the best model is ARIMA(2, 0, 2)(1, 0, 1, 24). Now, we train a model, giving us the following:

SARIMAX Results

Dep. Variable:	Percent Change	**No. Observations:**	26276
Model:	SARIMAX(2, 0, 2)x(1, 0, [1], 24)	**Log Likelihood**	-32692.243
Date:	Thu, 24 Mar 2022	**AIC**	65438.487
Time:	16:05:51	**BIC**	65659.250
Sample:	07-03-2018	**HQIC**	65509.771
	- 07-02-2021		
Covariance Type:	opg		

The coefficients and P-values:

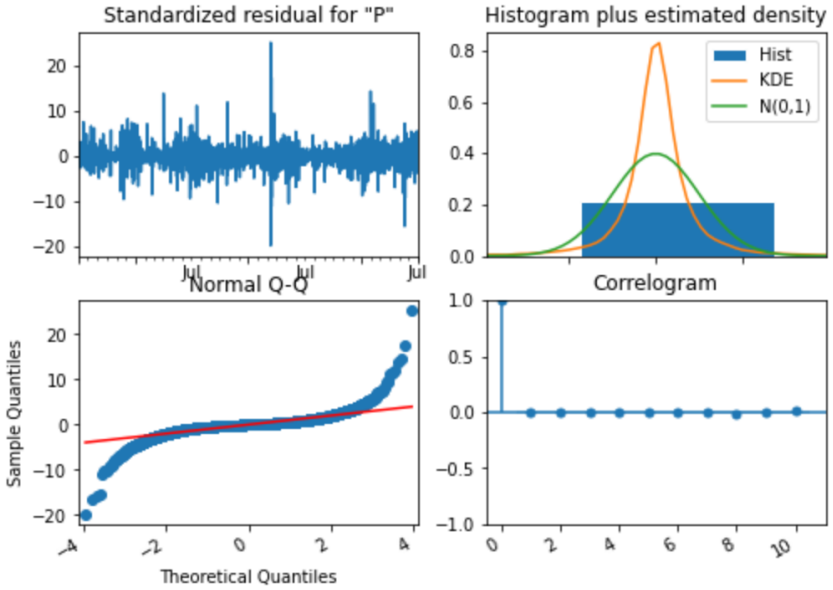
	coef	std err	z	P>|z|	[0.025	0.975]
x1	0.0003	0.001	0.412	0.680	-0.001	0.002
x2	-0.0088	0.001	-8.902	0.000	-0.011	-0.007
x3	0.0077	0.001	5.242	0.000	0.005	0.011
x4	-0.0017	0.002	-0.999	0.318	-0.005	0.002
x5	0.0045	0.001	4.350	0.000	0.002	0.007
x6	0.0051	0.002	2.130	0.033	0.000	0.010
x7	0.0021	0.003	0.813	0.416	-0.003	0.007
x8	-0.0095	0.003	-2.919	0.004	-0.016	-0.003
x9	0.0084	0.004	2.050	0.040	0.000	0.016
x10	-0.0171	0.004	-4.248	0.000	-0.025	-0.009
x11	-0.0050	0.004	-1.313	0.189	-0.012	0.002
x12	-0.0152	0.006	-2.663	0.008	-0.026	-0.004
x13	0.0050	0.005	0.974	0.330	-0.005	0.015
x14	0.0105	0.005	2.125	0.034	0.001	0.020
x15	0.0189	0.006	3.008	0.003	0.007	0.031
x16	0.0095	0.006	1.682	0.093	-0.002	0.021
x17	-0.0065	0.009	-0.754	0.451	-0.023	0.010
x18	-0.0102	0.008	-1.201	0.230	-0.027	0.006
x19	-0.0021	0.011	-0.186	0.852	-0.024	0.020
x20	0.0341	0.011	3.136	0.002	0.013	0.055
ar.L1	-0.1157	0.060	-1.913	0.056	-0.234	0.003
ar.L2	0.5180	0.049	10.485	0.000	0.421	0.615
ma.L1	0.0498	0.060	0.832	0.405	-0.068	0.167
ma.L2	-0.5874	0.052	-11.192	0.000	-0.690	-0.485
ar.S.L24	0.1653	0.081	2.049	0.040	0.007	0.323
ma.S.L24	-0.2175	0.080	-2.716	0.007	-0.374	-0.061
sigma2	0.7054	0.002	470.026	0.000	0.702	0.708

The last bit of the summary:

Ljung-Box (L1) (Q):	0.16	**Jarque-Bera (JB):**	2661703.57
Prob(Q):	0.69	**Prob(JB):**	0.00
Heteroskedasticity (H):	1.45	**Skew:**	0.11
Prob(H) (two-sided):	0.00	**Kurtosis:**	52.31

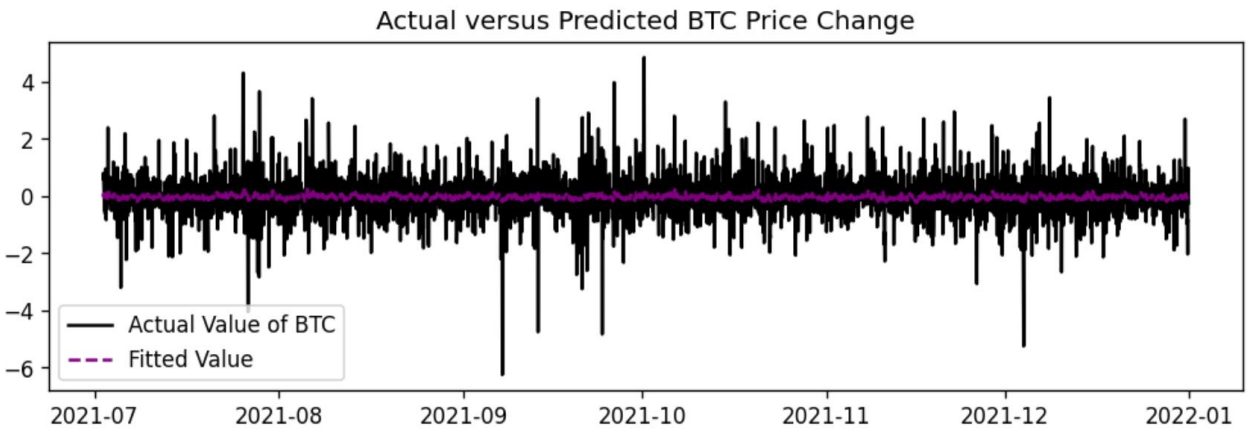
Though this is quite a bit of information, we decided to include it to demonstrate a few interesting things. Looking at the P-values for several of the exogenous regressors, such as column x_{19}, we see that they are *not* statistically significant. Perhaps we could use this information to simplify our model. We also see this with one of the moving average coefficients. Further testing might yield a simpler but more powerful model, though we do not explore that in this project.

Now, looking at the diagnostics:

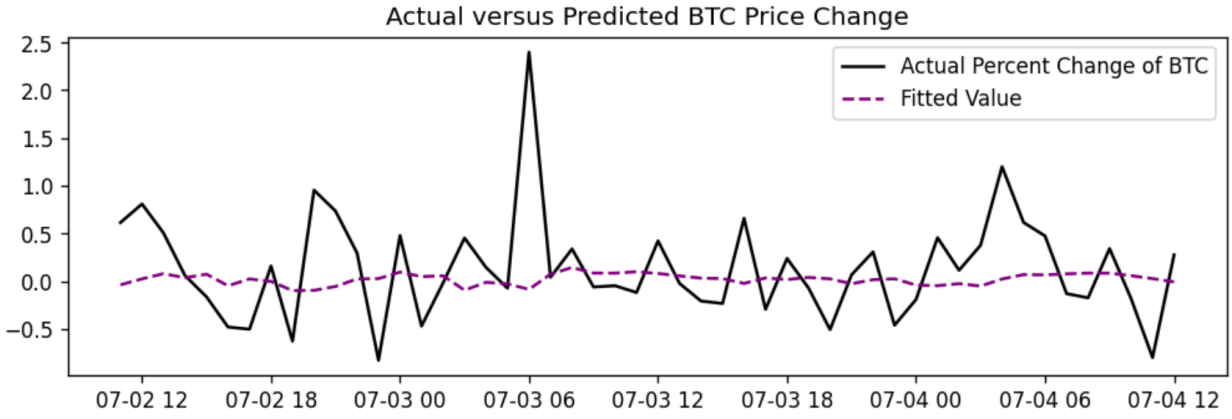


We note here that *technically*, the Auto-ARIMA function found an $ARIMA(2,0,2)(1,0,1,24)$ with intercept model. However, after training for an intercept and without, the intercept model had poorer performance, so we continue with the non-intercept model.

Though these diagnostics are not terrible, we do see that in the *Normal Q-Q* plot, the edges are spiraling away from the red line, indicating that this type of model might not be best for this data. We will now compare the fit on the full data.



Unfortunately, given the large amount of data, this graphic is difficult to see. However, we can reasonably say that the SARIMAX model cannot capture the extreme drops and increases in price. Focusing in only on the first 50 hours, we get the following graph:



From this, we see that the model is predicting increases and decreases in price, but as determined above, cannot capture the full volatility. Creating an accuracy metric based on whether the model predicts the correct sign of the change, we find it is only 47.37% accurate.

We note here that we have forecast *the entire testing period*. Normally, it is preferred to forecast only a few time steps into the future or even a *single* timestep into the future. Then, we could retrain with the training set *plus* the true values/information from that hour, forecast into the next hour with an up-to-date agent, and so on. This would hopefully allow us to retain the accuracy that might be lost with long-term forecasting. However, the usage adopted in this project is still interesting to see how leaving a SARIMAX trading agent to its own devices for some time performs, even if we believe there is a better method.

iii. eXtreme Gradient Boosting

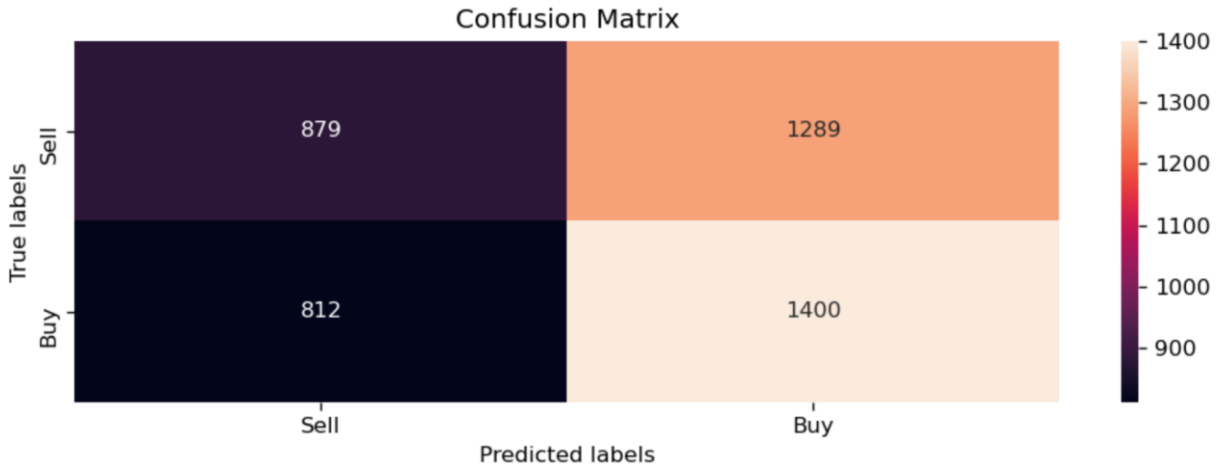
Recall from the *Primer* that *eXtreme Gradient Boosting* (XGBoost) is a gradient-boosted decision tree algorithm with a specific implementation designed to be quick. We mention here the parameters ranged over in the grid search but refer the reader to the documentation for the full list, as it is extensive.

Hyperparameters Searched Over for XGboost	Type/Number
Max Depth	3, 4, 5, 7
Learning Rate η	$1 \times 10^{-1}, 1 \times 10^{-2}, 1 \times 10^{-3}, 1 \times 10^{-4}, 1 \times 10^{-5}$
γ	0, 0.25, 1
λ	0, 1, 5, 10

After training, we find that the best combination among our hyperparameters is: $\gamma = 1, \eta = 0.01, \lambda = 10$, and a max depth of 5. Fortunately, we can output the full parameter list from python:

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=1, gpu_id=-1,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.01, max_delta_step=0, max_depth=5,
              min_child_weight=1, missing=nan, monotone_constraints='()',
              n_estimators=100, n_jobs=8, num_parallel_tree=1, random_state=0,
              reg_alpha=0, reg_lambda=10, scale_pos_weight=1, subsample=1,
              tree_method='exact', validate_parameters=1, verbosity=0)
```

The *Confusion Matrix*, a visualization of how the model correctly classifies the instances it was given, will help us determine the performance. The diagonals of the matrix are the correctly classified instances, so we hope those have a higher number than their off-diagonal counterparts.

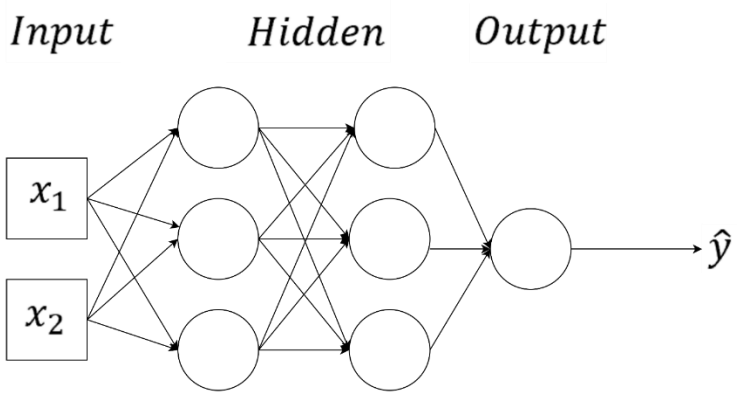


Though we do see this, we also see a significant class imbalance, and the overall accuracy is 52%.

B. Artificial Neural Networks

i. Multi-Layer Perceptron

The simplest *Artificial Neural Network* (ANN), yet surprisingly effective at many tasks, is the *Multi-Layer Perceptron* (MLP). As the name implies, this is simply an ANN made up of layers of perceptrons. From this, we can begin to stack them in a single layer and provide a final perceptron (which we will refer to as *neurons* from now on) to aggregate their outputs and make a classification or regression, giving us a *Shallow Neural Network*. Or, we can add additional layers, giving us a *Deep Neural Network* (DNN):



The general idea is that with more neurons and layers, we can capture increasingly complex non-linearities or patterns in the data.

In the *Primer on Artificial Neural Networks*, we discussed *Forward Propagation*, *Backpropagation*, and *Gradient Descent* to update our parameters to reduce our loss, so we do not discuss them here. However, we introduce different *Optimizers* which we can overcome some of gradient descent's shortcomings. In this manuscript, we chose to use *Adam* and *RMSProp* alongside regular stochastic gradient descent.

Recall from the *Primer* our formulation of simple parameter update equations:

$$g^k = \frac{1}{s^k} \sum_{p \in \{1,\dots,s^k\}} \nabla \mathcal{L}_{(W,b)^k}(Y^{(p)}, \hat{Y}(X^{(p)})$$

$$(W,b)^{k+1} = (W,b)^k - \alpha_k g^k$$

For RMSProp, we have some modifications. First, we define a recursive expectation [1]:

$$\mathbb{E}[(g^k)^2] = \rho \mathbb{E}[(g^{k-1})^2] + (1-\rho)(g^k)^2$$

Then, our update equation becomes [1]:

$$(W,b)^{k+1} = (W,b)^k - \frac{\alpha^k}{\sqrt{\mathbb{E}[(g^k)^2] + \epsilon}} g^k$$

Here, α^k, ρ, and ϵ are hyperparameters that can be tweaked.

For Adam, we have the following [1]:

$$m_k = \beta_1 m_{k-1} + (1-\beta_1) g^k$$

$$v_k = \beta_2 v_{k-1} + (1-\beta_2)(g^k)^2$$

$$\widehat{m}_k = \frac{m_k}{1-\beta_1^k}$$

$$\widehat{v}_k = \frac{v_k}{1-\beta_2^k}$$

$$(W,b)^{k+1} = (W,b)^k - \frac{\alpha_k}{\sqrt{\widehat{v}_{k+1}} - \epsilon}\,\widehat{m}_k$$

Here, $\alpha_k, \beta_1, \beta_2$ and ϵ are hyperparameters that can be tweaked and m_0 and v_0 are initialized to 0.

Now, one issue, or feature, depending on how one looks at it, of ANNs, is the *large* number of hyperparameters one can choose. The number of neurons, optimizer type, hyperparameters of the optimizer, hidden layers, activation functions, and more, can be chosen. In this manuscript, for our MLPs, we choose to do grid searches (which come with their own hyperparameters) over the number of hidden layers, neurons per hidden layer, activation function of the hidden layers, dropout rate (wherein we turn off neurons as we train to prevent overfitting), optimizer type, and the learning rate of our optimizer. We implement our model building in *Tensorflow* and *Keras.* A series of tables detailing the hyperparameters, both fixed and searched-for, follows.

Hyperparameters we range over for our MLP	Types/Numbers
Neurons	256, 512, 1024
Hidden Layers	2, 4, 6
Dropout Rate	0.5, 0.6
Activation Function	ReLU, tanh, sigmoid
Learning Rate	1×10^{-3}, 1×10^{-5}, 1×10^{-7}
Grid Search Function	
Epochs	20
Cross Validations	3

Our hyperparameters associated with called functions which we leave as default are:

RMSProp Optimizer [2]	Types/Numbers
ρ	0.9
ϵ	1×10^{-7}
Adam Optimizer [2]	
β_1	0.9
β_2	0.999
ϵ	1×10^{-7}
SGD Optimizer [2]	
Momentum	0
Nesterov	False
KerasClassifier Function used to build MLP	
Batch size	32

The rest of the hyperparameters (weight initializers, etc.) are the default values in Keras.

For both the grid search and the actual model building, we use *Binary Cross Entropy* as the loss function [4]:

$$\mathcal{L}_{(W,b)}\left(Y^{(i)}, \hat{Y}^{(i)}\right) = Y^{(i)} \log\left(\hat{Y}^{(i)}\right) + \left(1 - Y^{(i)}\right) \log\left(1 - \hat{Y}^{(i)}\right)$$

Our metric of interest is *Accuracy*, or how well our predicted labels match the known ones.

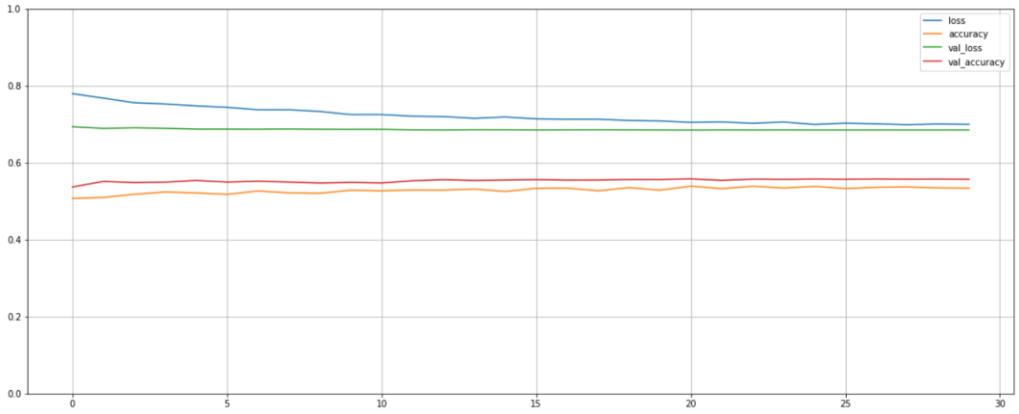
Now that we have our hyperparameters for our MLPs, our training data is from July 3rd of 2021 at 3:00 PM to December 31st of 2021 at 10:00 PM. We set the target for each instance as the label, which tells us whether we should buy or sell in that instance.

Then, we run a train and validation split on the training set with the validation size as 0.2, or 20% of the training data, with *random* shuffling with a seed of 100. We do this because we are not attempting to capture any temporal dependence between hours, as the MLP will see each hour and decide based on the information it has only for that instance. We then run our PCA creation function on all three sets.

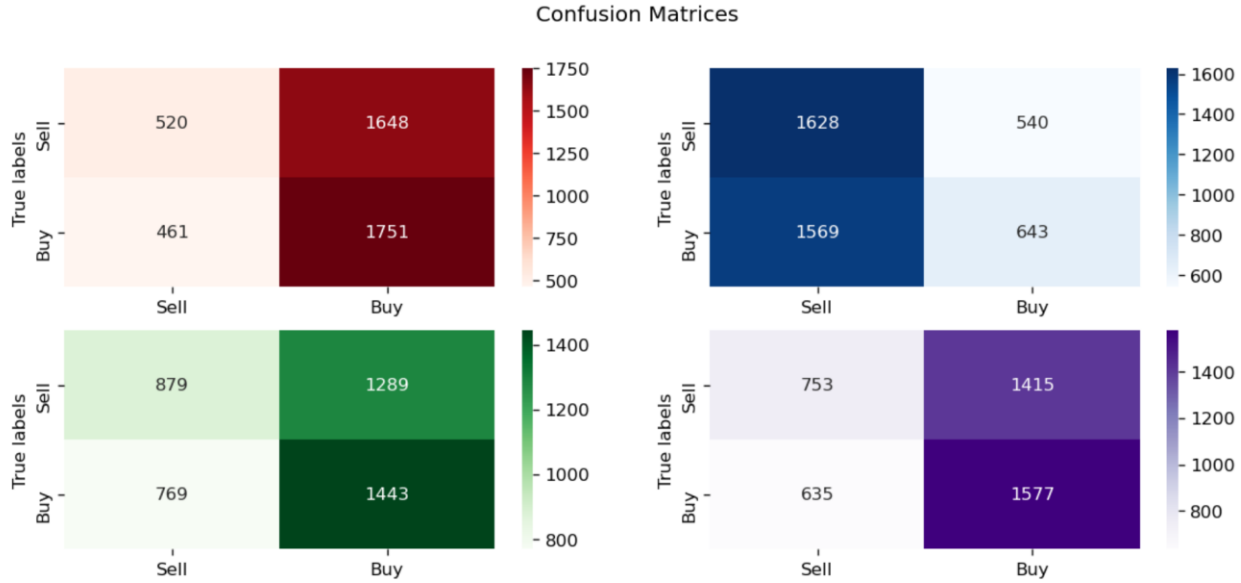
From here, we run our grid search with the above parameters on the training set, take the best hyperparameter set, and train a model with the training and validation data. We use a batch size of 32, the default, for training, and run it for 30 epochs. After testing, we found that the MLP quickly leveled off in performance, so more epochs are not needed. Once we have our trained model, we run our testing PCA matrix through it to arrive at decisions an agent based off the trained MLP would have made during the six months we chose as testing data. We can compare these decisions to the *actual* decisions it should have made with a confusion matrix, as we did with XGBoost.

After searching, we find the best parameter set is: ReLU activation function, 512 neurons, dropout rate of 0.6, 2 layers, and Adam optimizer with learning rate of 1×10^{-5}.

A plot of a training session with this architecture shows a slight, albeit existent shape indicating progress in *loss*, *validation loss*, *accuracy*, and *validation accuracy*. We also see that the progress quickly reaches a plateau.



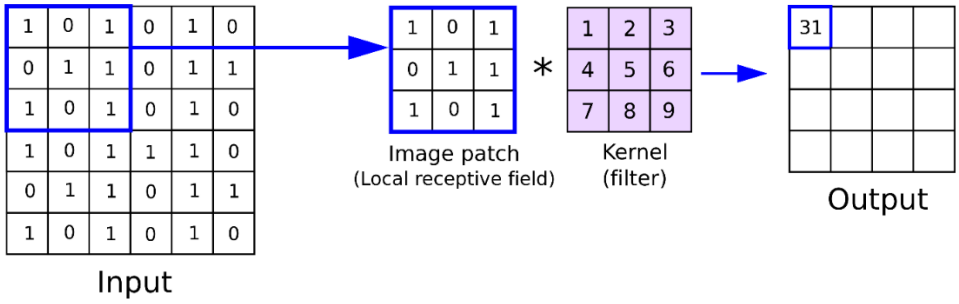
Due to the stochastic nature of model initialization with the *Glorot Uniform Initializer* in the MLP, CNN, and LSTM, there can be slight differences in model performance after training among models with the same architecture. To compensate for this, we will train and display the results for *four* trainings with the *same* architecture. It is important to note that *after* training, the algorithms are deterministic. The stochastic initialization only affects the models in training. The confusion matrices for four trainings are:



We see that these MLPs generally lean far in one direction of Buy or Sell. Given that we know that the data is roughly equal, this is an issue.

ii. Convolutional Neural Networks

Convolutional Neural Networks (CNN) take their name from their unique method of identifying important structures in the data, *convolutions*. Recall that in MLPs that we simply feed each data instance to the neural net as a series of numbers. If the object in question is an image, the data would be numbers associated with pixels and we would just flatten the 2D matrix into a single large row of numbers for the feed-forward portion. However, we might miss spatial dependencies in the image by doing so. Instead, we introduce two new operations which define the CNN and allow us to capture higher-level features and dependencies. The first is the *Convolutional Layer*. In the convolution layer, a *kernel* of size (s, s) is passed over the image, processed as one would normally do for data, but left as a 2D object (technically a 3D one with the final channel being 1 if it is grayscale or varying with RBG). This kernel can be learned by the neural net, and it extracts features by multiplying each pixel by the values in the kernel, then adds them. This is then one value in the *Feature Map*. After creating one value of the feature map, the kernel then moves the number of *Strides*, or steps along the image, creates a new value, and continues until we have a full feature map [5].



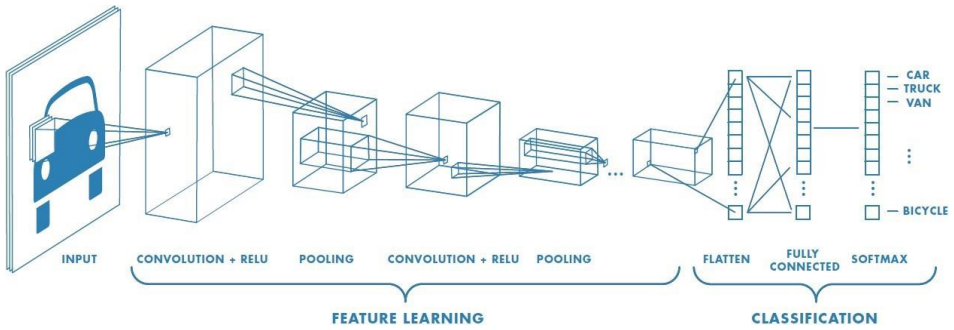
The learned kernels can be thought of as detecting patterns in the data of increasing abstraction as the number of convolutional layers increases. In the first layers, they might be finding simple geometries or edges and in later layers might be finding patterns within the spatial distribution of those geometries (such as two ovals close together being "eyes").

However, we see that the output is smaller than the original input. This is not necessarily a problem, but we can control this with *Padding*, wherein we add a layer of zeros around the input to maintain the size as we pass the kernel over it. The stride, kernel size, and padding are hyperparameters.

The second feature of CNNs is the *Pooling Layer*. The pooling layer further reduces the feature map by taking the values in an (n, n) box and finding the *Average* or *Maximum* of them. This is then a new value in our pooling layer's feature map.

So, our CNN is structured as a series of convolution layers followed by pooling layers. Once we have extracted important features for the image with these layers, we pass the flattened results to a regular, fully connected DNN for final analysis and classification. In this way, we harness the power of DNNs on the most relevant features of an image, as opposed to just the pixel values.

An informative summary image follows [6]:



Now, CNNs have proven very effective for image classification, and we will attempt to harness their power for our financial task. To do so, we will need to transform our financial data into "images."

Our "image" creation process is relatively simple. We take the last 12, 18, or 24 slices and stack them together to create a $(12,20)$, $(18,20)$, or $(24,20)$ array. We then add a final dimension to get a size $(n_{steps}, n_{features}, 1)$ "image."

Now, because we must subsume the first 12 hours into an "image", our training data runs from July 4th of 2018 at 2:00 AM (11 hours after the first several algorithms, as we roll all 11 hours together with the 12th for the "image" at that hour) to July 2nd of 2021 at 9:00 AM. We lost an hour from earlier algorithms at the end as our "image" creation function doesn't include the final hours. For the 18-hour "images", our training data is from July 4th of 2018 at 8:00 AM to July 2nd of 2021 at 9:00 AM. Finally, for our 24-hour "images," our training data is from July 4th of 2018 at 2:00 PM to July 2nd of 2021 at 9:00 AM.

For the fully connected portion, we reuse the best MLP structure found earlier with these extra layers added on in the beginning for feature extraction. The following table has our hyperparameters.

Hyperparameters we range over for our CNN	Types/Numbers
Pool Type	Max, Average
Convolutional Layer Activation Function	ReLU, tanh, sigmoid
Filters	16, 32, 64
Kernel Size	(2,2), (3,3)
Convolution and Pooling Layers	1, 2
CNN Hyperparameters that remain fixed	
Pool Size	(2,2)
Stride for Conv Layer	1
Stride for Pooling Layer	2 (Defaults to same as pooling)
Padding for Conv Layer	Valid (No padding)
Padding for Pooling Layer	Valid

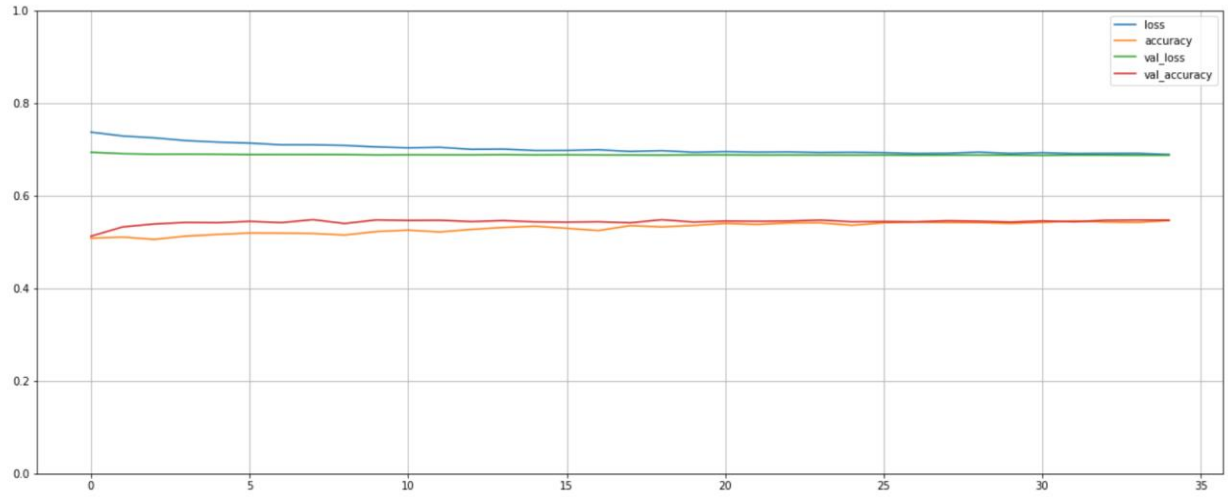
The rest of the hyperparameters in the called functions (weight initializers, optimizer hyperparameters, etc.) are the default values listed in the MLP section. We also use the same number of epochs and cross validations from the MLP grid search.

After searching, we find the best CNN parameters with 12 data instances making up an "image" are: average pool type, tanh convolutional layer activation function, 64 filters, a kern size of (3,3), and only 1 convolution and pooling layer.

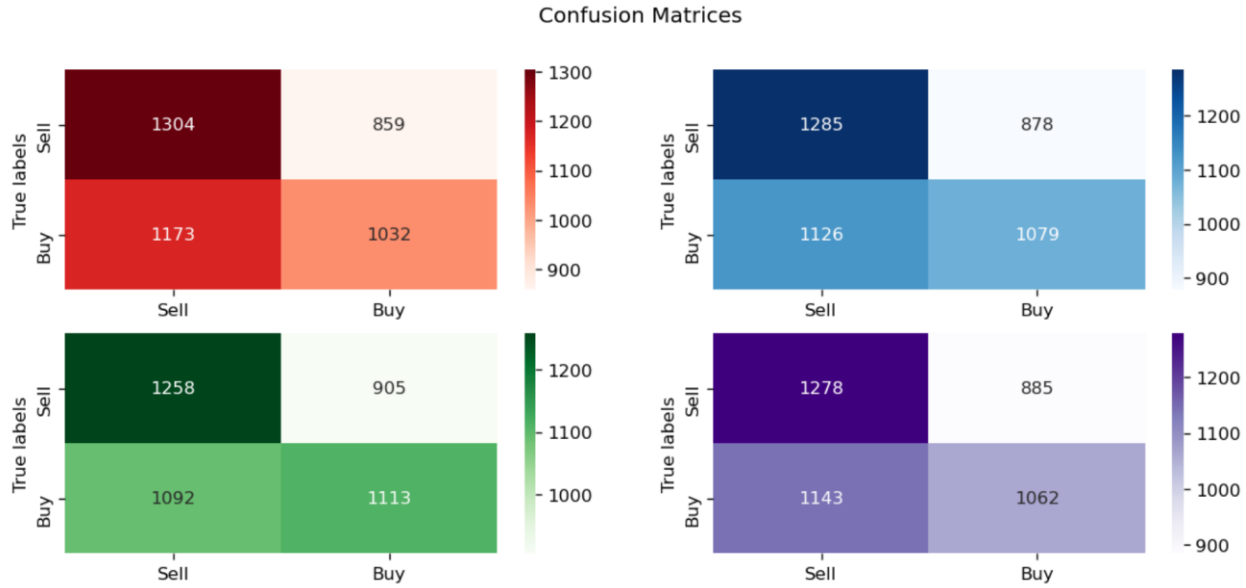
For an "image" of 18 data instances, we have the exact same parameters, but the number of filters has increased to 128.

For an "image" of 24 data instances, we ended up with the same architecture as the 12 data-instance "images".

The typical training with 35 epochs of batch size 32 looks like our MLP:

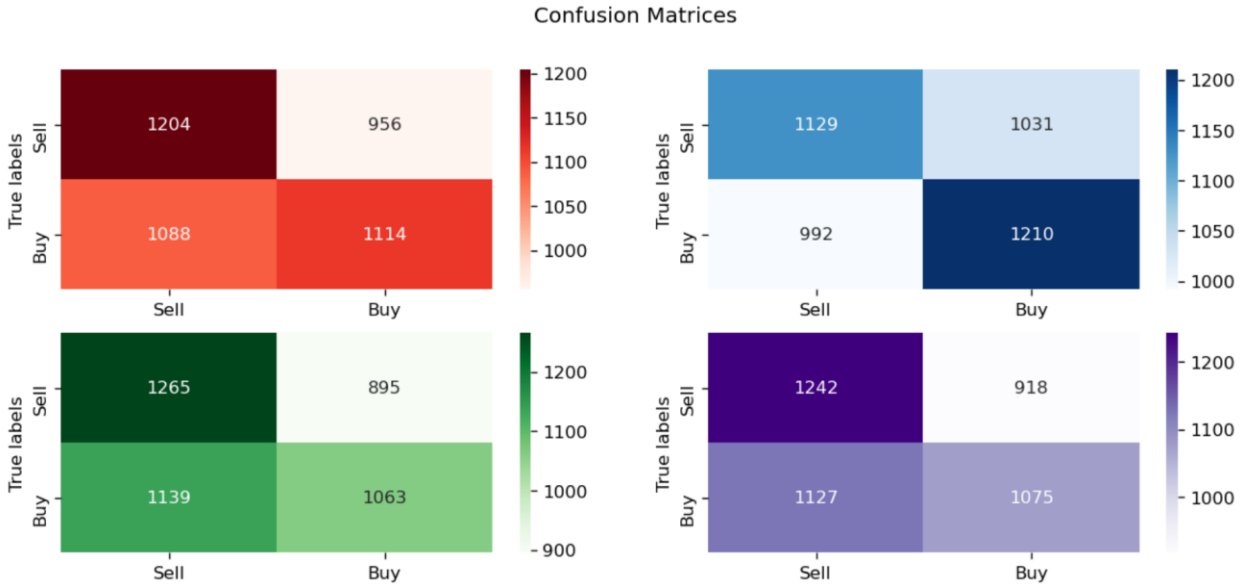


Finally, we create confusion matrices to determine how well our CNN is classifying the "images." same as we did with the MLP. For the 12 data-instance "images," we have:

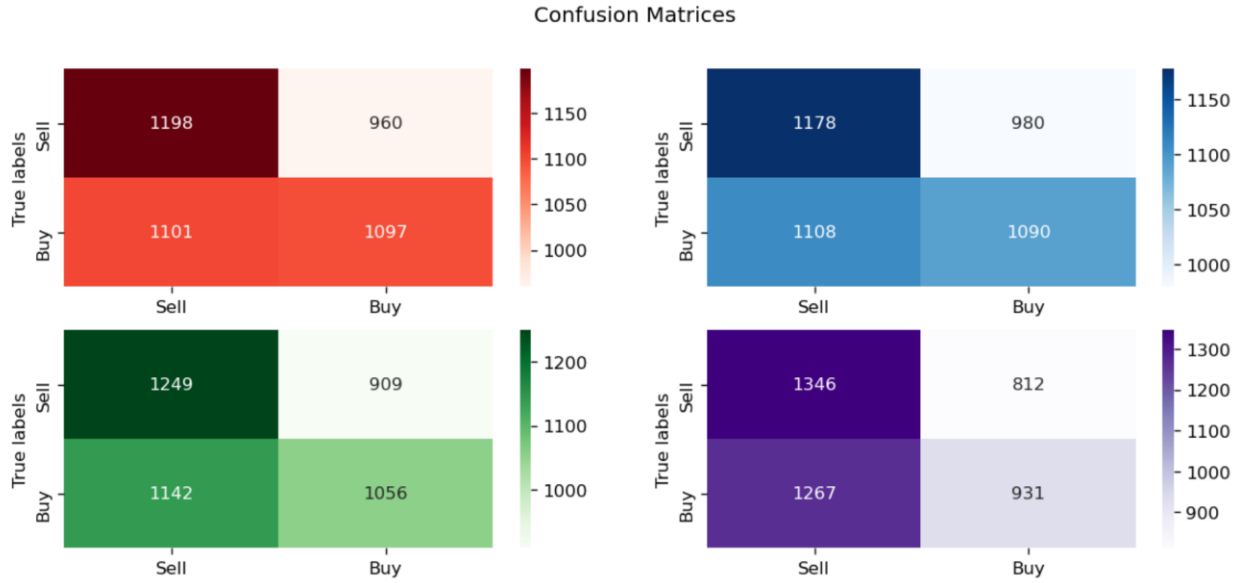


We immediately see several differences, as our CNN architecture has a slightly more even division between and Buy and Sell. We see in our investing section how this affects the agents' performance.

For the 18 data-instance "images," we have:



Finally, for the 24 data-instance “images,” we have:



iii. Recurrent Neural Networks

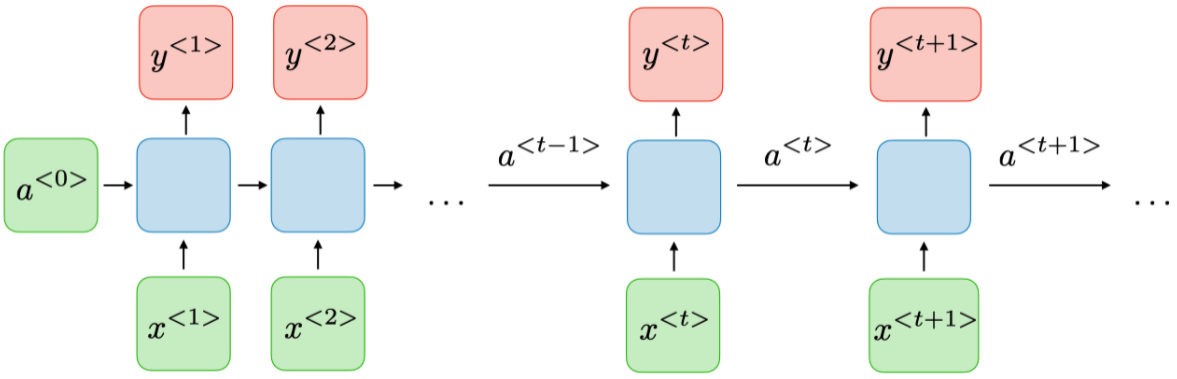
Thus far, the two ANN architectures do not have any method of capturing temporal dependency. We attempted to introduce a kind of temporality to our data by using consecutive instances to create "images" for the CNNs, but that is a bit artificial. Just as CNNs are powerful at image classification by their ability to capture abstract features in the data with convolutions, a type of ANN called a *Recurrent Neural Network* (RNN) has a structure that allows it to capture temporalities in the data.

Consider a simple translation ANN. If we wanted to translate the following sentence, "Dogs are nice," into Portuguese, we could try to do it word by word. If we did, we would find "Dogs" is "Cães" while "are" is "são". However, on the last word, "nice," we have a problem. In Portuguese, adjectives change depending on the gender and plurality of the noun they describe. Thus, our final word could be "simpátio," "simpática," "simpáticos," or "simpáticas". Without our ANN having the ability to receive information about the plurality and gender of the noun in the beginning of the sentence, the task is hopeless.

RNNs can learn this dependence by allowing the hidden state, output, or the activation (all three words appear in the literature) of a previous block to be an input *along with the regular input* for the current one.

Let $\boldsymbol{x}^{(i)}$ be the i^{th} instance of data with T time steps in it. For example, $\boldsymbol{x}^{(i)}$ could be our earlier sentence "Dogs are nice." Then, $\boldsymbol{x}^{(i)<1>}$ is "Dogs," $\boldsymbol{x}^{(i)<2>}$ is "are," and $\boldsymbol{x}^{(i)<3>}$ is "nice." Now, the RNN cell would receive $\boldsymbol{x}^{(i)}$ and analyze each piece sequentially, outputting a translation (in this case) for each word *while also providing information for the next piece to consider, as well.*

Graphically, while dropping the (i) superscript, this looks like [7]:



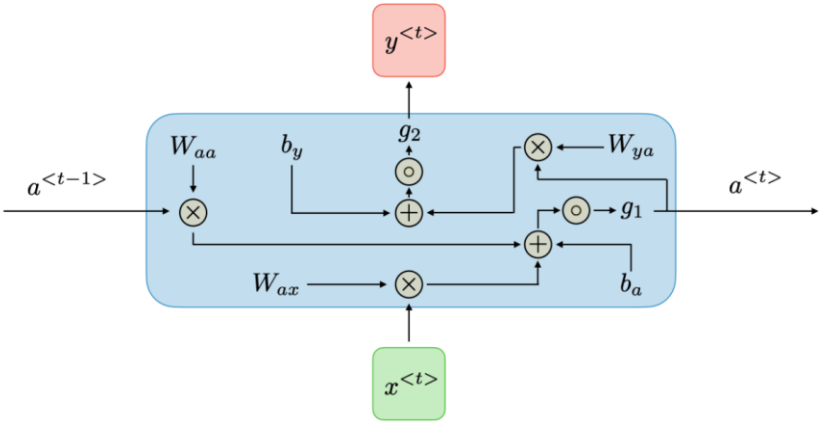
In this case, $\hat{y}^{<t>}$ is the prediction for that member of the sequence. This is the *unrolled* graphic, as it is often shown with a single blue, green, and red block with an arrow leaving and returning to the blue block, signifying the recurrent nature of the blue block. We see that in addition to providing a prediction, each blue block creates an output, or hidden state, $a^{<t>}$, and feeds that into the next block, as well. Thus, information about the beginning of the phrase, in the form of the activation, can travel along the block and influence subsequent predictions.

Opening the blue block, we find three sets of weights, two activation functions, and two sets of biases. Thus, our activation and prediction are calculated as [7]:

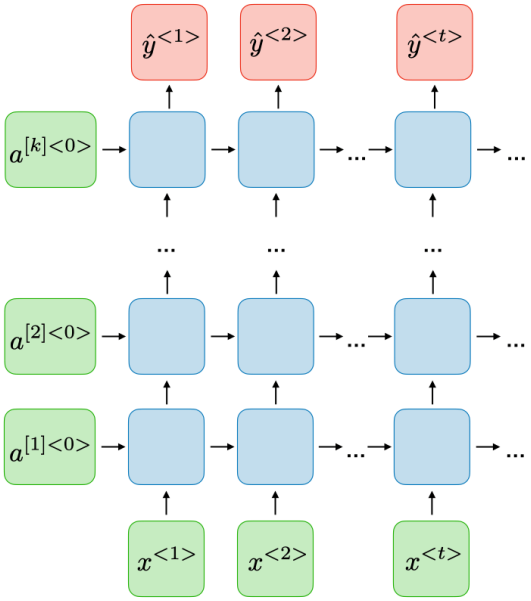
$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}\boldsymbol{x}^{<t>} + b_a)$$

$$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

Graphically, this is often represented as [7]:

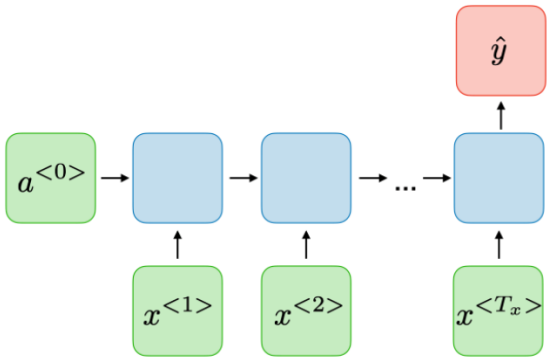


We note here that in creating an RNN layer, it has the number of blue blocks as the length of the sequences it is being fed. When we call the RNN layer creation function in Keras and give it a *Units* argument, we are specifying the size of the output or activation [8]. Thus, we can control the information each blue block recieves by changing the size of the output $a^{<t>}$. We can then stack RNN layers together to get a *Deep Recurrent Neural Network* [7]:



Here, the k in $a^{[k]<t>}$ does *not* represent the k^{th} training example, but rather, following the convention established in the *Primer*, represents the k^{th} layer. Written out fully, the output of the k^{th} layer on member t of the i^{th} training instance would be $a^{(i)[k]<t>}$.

Finally, we note that unlike the graphics above, for this project, we are not producing a prediction for each member of the sequence, but a single output, *Buy* or *Sell*, after the RNN has processed all the members of the $\boldsymbol{x}^{(i)}$ sequence. This looks like [7]:



It is called *Many-to-One*, as we take a sequence of length T and produce only a single output. For our RNN, this will be a classification.

- Long Short-Term Memory Recurrent Neural Network

Though a vanilla RNN can handle temporal dependencies, it suffers from *Vanishing Gradients*, wherein during the backpropagation phase, because each neuron must be updated going back in time, multiplying by increasingly small gradients makes it difficult to update the weights. To avoid this, two different architectures will be introduced. The first is the *Gated Recurrent Unit* (GRU) and the second is the *Long Short-Term Memory Unit* (LSTM). Each of these has *Gates* which allow the RNN to learn more about how it should process the data it is receiving. The GRU has an *Update Gate* and a *Relevance Gate*, while the LSTM has both and an additional *Forget Gate* and *Output Gate*. Given that the LSTM is essentially just a GRU with extra features, in this paper, we will focus only on the LSTM.

Each gate is written as, once again dropping the (i) superscript [7]:

$$\Gamma = \sigma(W\boldsymbol{x}^{<t>} + Ua^{<t-1>} + b)$$

Here, $W, U,$ and b are weights and the bias associated with each gate and can be learned, and σ is the sigmoid function.

Thus, each gate takes on a value between 0 and 1. We provide a table summarizing the gates and what they mean [7]:

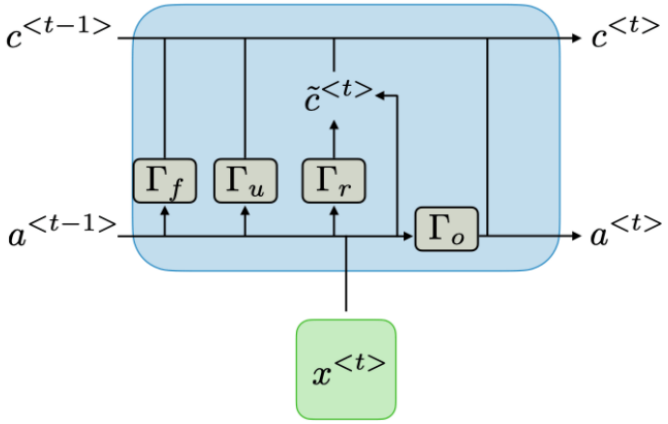
Gate	Meaning
Update Gate Γ_u	Determines how much past information matters
Relevance Gate Γ_r	When to remove previous information
Forget Gate Γ_f	When to erase a previous cell
Output Gate Γ_o	How much to give the next cell

Once we have our gates, we can calculate the output $a^{<t>}$ and the new information $c^{<t>}$ we will pass along to the next cell to determine how it will treat the data [7]:

Symbol	Calculation
$\tilde{c}^{<t>}$	$\tanh(W_c[\Gamma_r * a^{<t-1>}, \boldsymbol{x}^{<t>}] + b_c)$
$c^{<t>}$	$\Gamma_u * \tilde{c}^{<t>} + \Gamma_f c^{<t-1>}$
$a^{<t>}$	$\Gamma_o * c^{<t>}$

Note that * is element-wise matrix multiplication.

This can be represented graphically as [7]:



Thus, we use an LSTM RNN (LSTM, for short) to attempt to capture temporal information about the pricing. To do so, we create a sequence for each data instance we feed to the LSTM. Employing the same strategy as with the CNNs, we will take 12, 18, or 24 slices of data as a sequence (but not stacking them into an "image"). The training set for each is the same as its counterpart in the CNN, as the sequence creation function also must subsume the first 11, 17, or 23 hours with the next for the sequence. It also removes the final hour, as well.

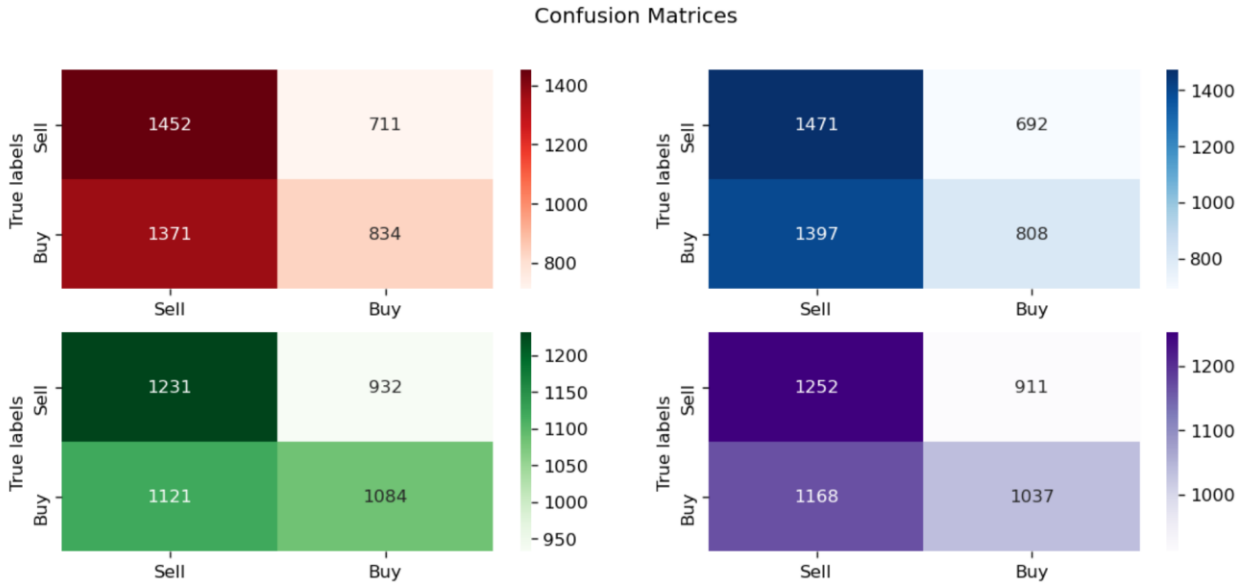
Our hyperparameters are the following:

Hyperparameters we range over for our LSTM	Type/Number
Activation Function	ReLU, tanh, sigmoid
Units	256, 512, 1024
Dropout Rate	0.5, 0.6
Layers	2, 3, 4
Optimizer	Adam, RMSProp, SGD
Learning Rate	1×10^{-3}, 1×10^{-5}, 1×10^{-7}
KerasClassifier Function used to build LSTM	
Batch size	32

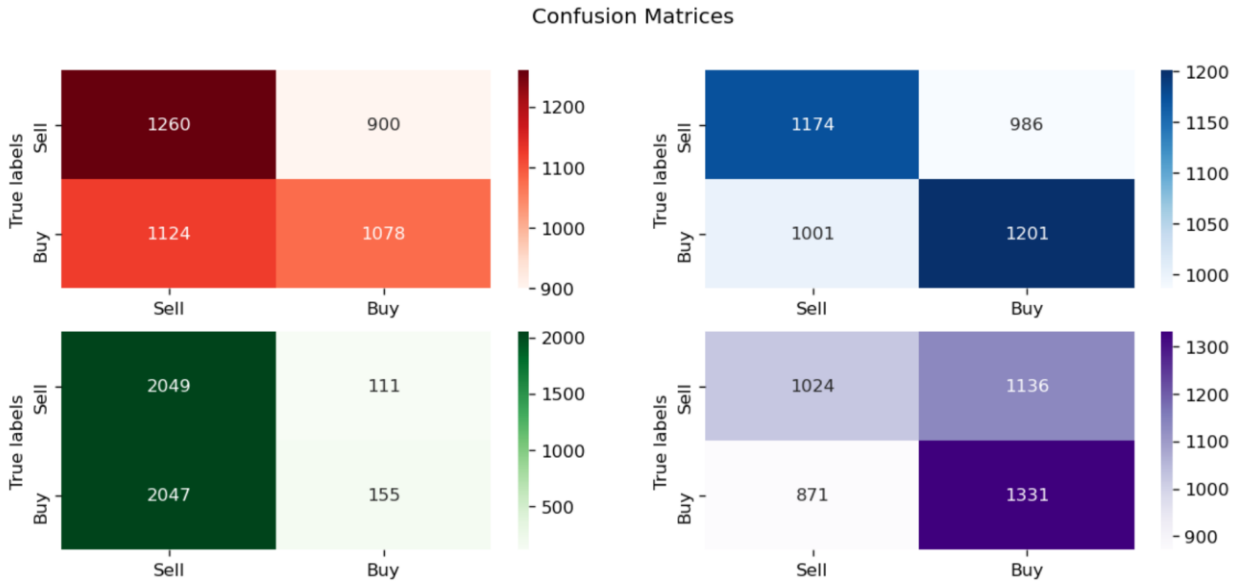
The rest of the hyperparameters in the called functions (weight initializers, optimizer hyperparameters, etc.) are the default values listed above, as are the grid search parameters.

We note that, after long training periods for the 12 and 18-hour sequences, with neither the optimizer SGD nor the Learning Rate of 1×10^{-3} appearing as candidates, we decided to leave those out of our 24-hour sequence grid search.

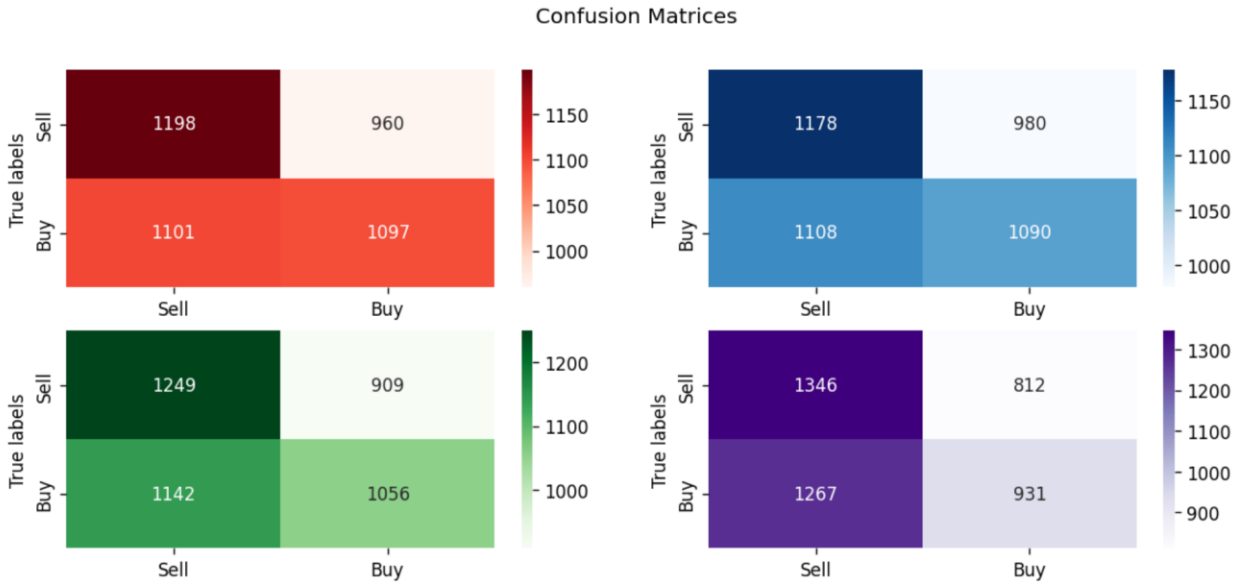
For the 12-hour sequences, we found the best architecture to be: sigmoid activation function, dropout rate of 0.5, learning rate of 1×10^{-5}, 4 layers, RMSProp optimizer, and 256 units in each layer. Following the same structure as with the CNNs and MLPs, we then trained four models for 20 epochs with batch size 32 with the same architecture to see how the random initialization might affect them, giving us the following confusion matrices:



For sequences of length 18, we found the same architecture, but with 1024 units per layer and 4 layers. The confusion matrices are:



For 24-hour sequences, we found the same architecture as the 12-hour sequences, but with only two layers and 512 units in each. The confusion matrices are:



A typical training session looks exactly like the MLP and CNN, so we won't include an example here.

C. Deep Reinforcement Learning:

Recall that in Reinforcement Learning, we seek to find a policy our agent should follow to maximize some reward function (often in terms of maximizing the state-value or action-value functions). In *Deep Reinforcement Learning* (DRL), we will be leveraging the power of neural networks in various fashions for their power in function approximation.

i. Deep Q-Networks

Recall the following update scheme for the action-value function in *Q-Learning* [1]:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{A_{t+1}} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

We also recall that Q-Learning is an off-policy and model-free algorithm, wherein we do not have a model of the transition probabilities between states, and we are not necessarily going to follow the policy that is used to generate the data, hence the inclusion of the $\max_{A_{t+1}} Q$ operator.

However, if we are using Q-Learning in a continuous state-space, as financial data is often modeled as and our PCA is mapping onto, then we cannot explore all possible states. Thus, our $Q^{\pi}(s, a)$ function is ultimately *unknown* and must be *estimated*. To do this, we use new notation [2]:

$$Q^{\pi}(s, a) \approx \hat{Q}(s, a, \boldsymbol{\Theta})$$

We attempt to approximate our true $Q^{\pi}(s, a)$ with a function of s and a that has parameters $\boldsymbol{\Theta}$ that depend on the policy in question and can be updated.

We then calculate the *Loss Function*, which tells us how far from our true action-value function we are [3]:

$$\mathcal{L}(\boldsymbol{\Theta}) = \mathbb{E}\left[\left(\mathrm{Q}^{\pi}(s, a) - \hat{Q}(s, a, \boldsymbol{\Theta})\right)^2\right]$$

However, because we are sampling in continuous state space with Monte Carlo methods, we must transform our expected value into an average [3]:

$$\mathcal{L}(\boldsymbol{\Theta}) = \frac{1}{\mathrm{N}} \sum_{i=1}^{N} \left(\mathrm{Q}^{\pi}(s_i, a_i) - \hat{Q}(s_i, a_i, \boldsymbol{\Theta})\right)^2$$

Now, with our approximation of the true action-state function, we simulate, explore, or otherwise generate N samples (a *batch*) of action-state pairs and their associated values. We then update the weights at time t of our approximate function to reduce the error in our approximation. We take the negative gradient of $\mathcal{L}(\boldsymbol{\Theta})$ with respect to $\boldsymbol{\Theta}$ and use it to update $\boldsymbol{\Theta}$ [4]:

$$\boldsymbol{\Theta}_{t+1} = \boldsymbol{\Theta}_t - \alpha \nabla_{\boldsymbol{\Theta}_{\mathrm{t}}} \mathcal{L}(\boldsymbol{\Theta}_{\mathrm{t}})$$

Written out, this is:

$$\boldsymbol{\Theta}_{t+1} = \boldsymbol{\Theta}_t + \alpha \frac{1}{N} \sum_{i=1}^{N} \left[Q^{\pi}(s_i, a_i) - \hat{Q}(s_i, a_i, \boldsymbol{\Theta}_{\mathrm{t}})\right] \nabla_{\boldsymbol{\Theta}_{\mathrm{t}}} \hat{Q}(s_i, a_i, \boldsymbol{\Theta}_t)$$

Note that due to the negative introduced by $\nabla_{\boldsymbol{\Theta}_t} - \hat{Q}(s_i, a_i, \boldsymbol{\Theta}_t)$, the $-\alpha$ becomes positive. Unfortunately, this brings us to another problem. In supervised learning, we would *know* the correct labels or regression values of our training data, and thus have a *target* to aim for. However, in the above equation we do not actually know our target $Q^{\pi}(s_i, a_i)$, as that is what we are actively trying to estimate, and so seem to be unable to update our weights. To adjust for this, we will form a new target [5]:

$$Q^{\pi}(s_i, a_i) = r_i + (1 - done_i)\,\gamma \max_{a_i'} \tilde{Q}\left(s'_i, a'_i, \boldsymbol{\Theta}_t^{target}\right)$$

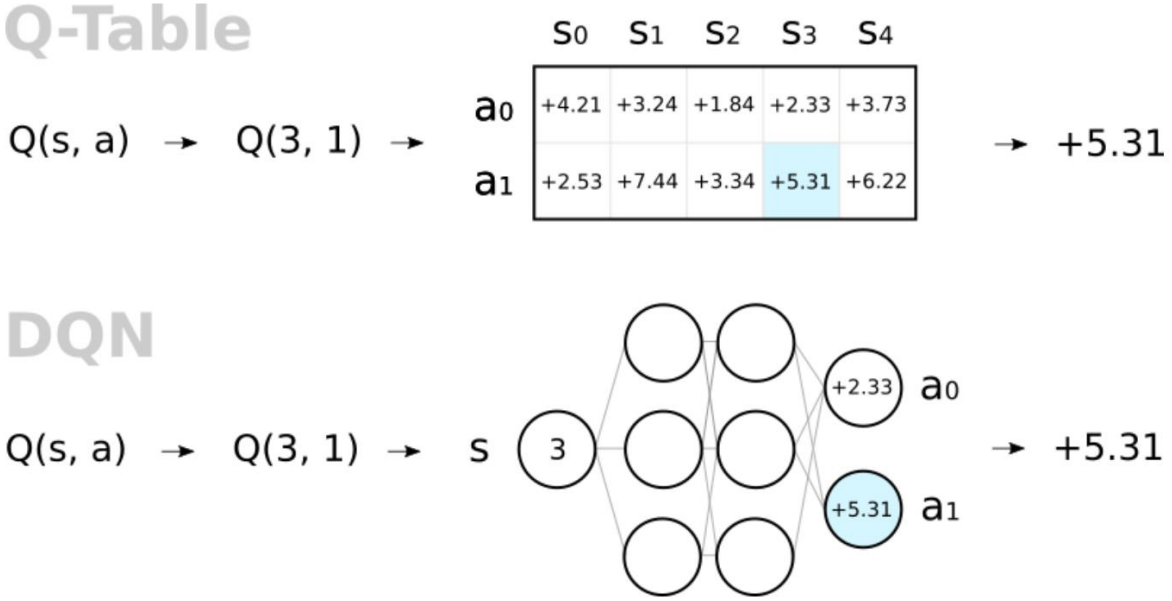
We are using a *different* set of weights, $\boldsymbol{\Theta}^{target}$, which we will update less often, to estimate our target. Notation wise, we will use the $\tilde{Q}$ with the $\boldsymbol{\Theta}^{target}$ weights and $\hat{Q}$ with the regular $\boldsymbol{\Theta}$ weights. We thus have *two* networks going, the *online* one being updated after every batch, and the *offline* one, which is updated far less often to serve as a more stationary target, and our update equation becomes [6]:

$$\boldsymbol{\Theta}_{t+1} = \boldsymbol{\Theta}_t + \alpha \frac{1}{N} \sum_{i=1}^{N} \Bigg[r_i + (1 - done_i)\gamma \max_{a_i'} \tilde{Q}\left(s'_i, a'_i, \boldsymbol{\Theta}_t^{target}\right) - \hat{Q}(s_i, a_i, \boldsymbol{\Theta}_t) \Bigg] \nabla_{\boldsymbol{\Theta}_t} \hat{Q}(s_i, a_i, \boldsymbol{\Theta}_t)$$

The plus sign in front of α has the same origin as above. To summarize the approach, we run the agent through the environment with an ϵ-greedy policy to collect experiences/transitions, tuples of $(s, a, r, s', done)$, in a *buffer*, $\mathcal{B}$. We sample from this to update the weights for the online network. After enough batches, we update the offline weights, as well, and use these new Q-values to explore more, collecting more experiences, and so on [7].

Now, we arrive at a significant piece of this paper, combining neural networks with Q-Learning by using neural networks for both $\tilde{Q}\left(s'_i, a'_i, \boldsymbol{\Theta}_t^{target}\right)$ and $\hat{Q}(s_i, a_i, \boldsymbol{\Theta}_t)$. This gives us a *Deep Q-Network* (DQN).

Compare the tabular method and the deep method visually [8]



We are estimating our action-value function and using it to select a best action (the max operator above). Thus, we are *indirectly* searching for an optimal policy by means of our value function. These kinds of algorithms are called *Critic* algorithms, as the neural network *critiques* the action taken by our agent with the Q-function [9].

- Prioritized Experience Replay

One might wonder if each transition or experience we pull from in our buffer $\mathcal{B}$ is equally important to training our agent? Surely some experiences will be very valuable and others less so. *Prioritized Experience Replay* (PER) is an attempt to provide that measure. Once we know which experiences are more important, we can choose from them with higher probability than those less interesting for our agent. Recall our TD Error from the Primer. We write it in Monte Carlo form, so it is recognizable from the parameter update equations above [10]:

$$\delta_i = r_i + \left[(1 - done_i)\,\gamma \max_{a'_i} \tilde{Q}\left(s'_i, a'_i, \boldsymbol{\Theta}_t^{target}\right)\right] - \hat{Q}(s_i, a_i, \boldsymbol{\Theta}_t)$$

We can use this δ_i to create a priority score [11]:

$$p_i = |\delta_i| + \epsilon$$

Here, ϵ is just a small number in case the TD error were zero. We then convert these into probabilities for how likely it is that the experience will be sampled [12]:

$$P(i) = \frac{p_i^\alpha}{\sum p_i^\alpha}$$

The parameter α is chosen empirically and in this manuscript, we choose $\alpha = 0.6$ as the original creators found. Now, we need to fix the bias we just introduced by the following [13]:

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^{\beta}$$

Here, β is again another empirically found hyperparameter. We will choose $\beta = 0.4$. We then normalize our weight with [14]:

$$w_i = \frac{1}{\max_i w_i} w_i$$

Finally, our new loss equation is [15]:

$$\mathcal{L}\left(\boldsymbol{\Theta}_{\mathrm{t}}, \boldsymbol{\Theta}_{\mathrm{t}}^{target}\right) = \frac{1}{N}\sum_{i=1}^{N}\left[\left(r_i + \left[(1 - done_i)\,\gamma \max_{a_i'} \tilde{Q}\left(s'_i, a'_i, \boldsymbol{\Theta}_t^{target}\right)\right] - \hat{Q}(s_i, a_i, \boldsymbol{\Theta}_{\mathrm{t}})\right) w_i\right]^2$$

- Double Deep Q-Networks

Though we have a network architecture capable of acting in continuous-state spaces, we run into an issue. Notice that we are using the same network for selecting the maximizing action as well as the Q-value for that maximum action. This leads to an *overestimation bias* [16]. We can avoid this issue with a small change.

Recall the i^{th} target in a DQN [17]:

$$Y_{DQN}^{target} = r_i + (1 - done_i)\gamma \max_{a_i'} \tilde{Q}\left(s'_i, a'_i, \boldsymbol{\Theta}_t^{target}\right)$$

To avoid this, we replace the selection of the best action with the target network by the online network, giving us a *Double Deep Q-Network*. So, our new equation for the i^{th} target is [17]:

$$Y_{Double\ DQN}^{target} = r_i + (1 - done_i)\gamma \tilde{Q}\left(s_i', \max_{a_i'} \hat{Q}(s_i', a_i', \boldsymbol{\Theta}_{\mathrm{t}}), \boldsymbol{\Theta}_t^{target}\right)$$

So, the online network is used to select the best action, and the target network is used to select the Q-value for that action. Thus, our loss equation becomes [18]:

$$\mathcal{L}\left(\boldsymbol{\Theta}_{\mathrm{t}}, \boldsymbol{\Theta}_{\mathrm{t}}^{target}\right) =$$

$$\frac{1}{N}\sum_{i=1}^{N}\left[r_i + \left[(1 - done_i)\gamma \tilde{Q}\left(s_i', \max_{a_i'} \hat{Q}(s_i', a_i', \boldsymbol{\Theta}_{\mathrm{t}}), \boldsymbol{\Theta}_t^{target}\right)\right] - \hat{Q}(s_i, a_i, \boldsymbol{\Theta}_t)\right]^2$$

- Dueling Deep Q-Networks

Another variation that we make to our standard DQN is called a *Dueling Deep Q-Network*. To begin, we will take a closer look at the relationship between $V^{\pi}(s)$ and $Q^{\pi}(s, a)$. Recall that $Q^{\pi}(s, a)$ differs from $V^{\pi}(s)$ only in that instead of following policy π for each action, we might

take a different action as a first step, then follow π from then on. To determine if that was advantageous, we define the *Advantage* as [19]:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

The *dueling* aspect of the Dueling Deep Q-Network comes from the idea that we begin to calculate $V^{\pi}(s)$ and $A^{\pi}(s,a)$ with the same network, but then split them at the end (hence, they are dueling). We thus have *three* sets of weights, $\boldsymbol{\Theta}_1, \boldsymbol{\Theta}_2$, and $\boldsymbol{\Theta}_3$. The first set, $\boldsymbol{\Theta}_1$, is shared by both $A^{\pi}(s,a)$ and $V^{\pi}(s)$, and then each recieves their own set for the dueling portion. Written with approximation notation, we have:

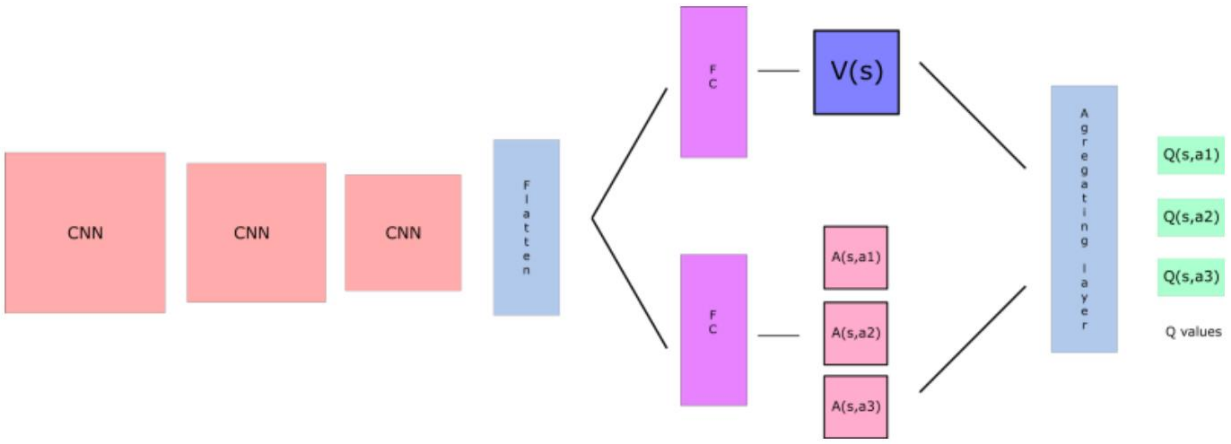
$$A^{\pi}(s,a) \approx \hat{A}(s,a,\boldsymbol{\Theta}_1,\boldsymbol{\Theta}_3)$$

$$V^{\pi}(s) \approx \hat{V}(s,\boldsymbol{\Theta}_1,\boldsymbol{\Theta}_2)$$

Now, we calculate our approximate Q-value by: [20]

$$\hat{Q}(s,a,\boldsymbol{\Theta}_1,\boldsymbol{\Theta}_2,\boldsymbol{\Theta}_3) = \hat{V}(s,\boldsymbol{\Theta}_1,\boldsymbol{\Theta}_2) + \left[\hat{A}(s,a,\boldsymbol{\Theta}_1,\boldsymbol{\Theta}_3) - \frac{1}{|A|}\sum_{a'}\hat{A}(s,a',\boldsymbol{\Theta}_1,\boldsymbol{\Theta}_3)\right]$$

From here, we continue as normal in our DQN architecture. However, a picture will help make sense of what this neural network looks like [21]:



We note here that Double DQN and Dueling DQN structures are not exclusive, and in this paper, we will implement them both, as their inclusion should perform better than a simpler DQN.

Thus, our DQN will be in the form of a Dueling Double DQN with PER.

We will adopt the structure of our best ANN, the CNN with 12-hour "images," as the structure of the neural network used to estimate the Q-function, just as in the above graphic.

Our model is thus a *Convolutional Neural Network-Based Dueling Double Deep Q-Network with Prioritized Experience Replay*.

For our reward system, we chose a simple one. The reward at a time step is the difference in the portfolio from the time step previous. So, if the algorithm makes a choice which leads to an

increase in the portfolio, we award it that change, and vice versa. We adopt the same system for the second RL algorithm, as well, though we will note something interesting when it comes to training.

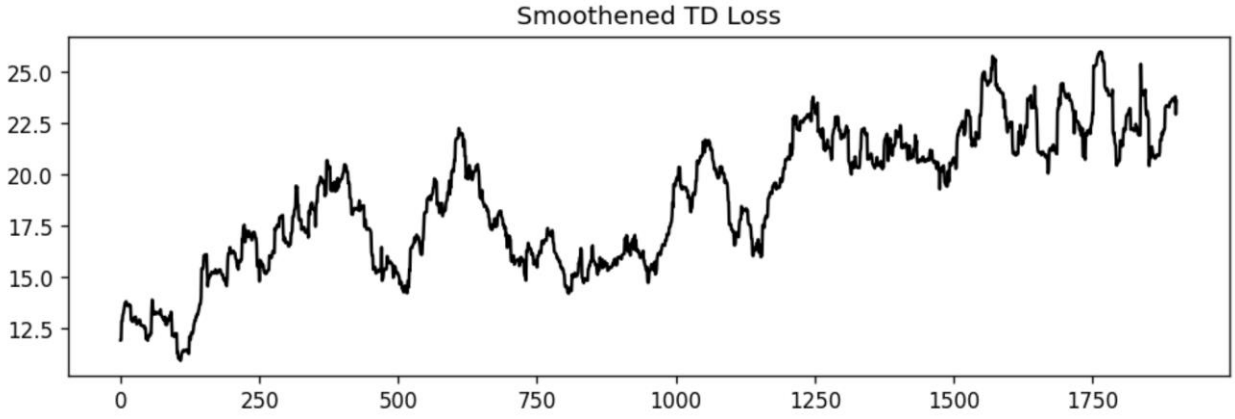
Given the *very* large number of parameters we could choose for the non-CNN portion of our model, we have chosen to use the default ones as used in *Deep Reinforcement Learning with Python*, while only increasing the *total steps* parameter and running through 10 different γ values. The training data are the same as the 12-hour CNN and LSTM.

As such, our hyperparameter search looks like:

Hyperparameter ranged over for our DQN	Type/Number
γ	0, 0.15, 0.25, 0.35, 0.5, 0.75, 0.8, 0.9, 0.95, 1
Hyperparameters left as is for PER	
α	0.6
β	0.4
ϵ	1×10^{-5}
Exploration ϵ	
Start	1
End	0.05
Decay Final Step	20000
Frequency for Logging and Updating	
Loss Frequency	20
Refresh Target Network	100
Evaluation Frequency	1000
Training Parameters	
Timesteps per Epoch	1
Batch Size	32
Total Steps	40000
Max Gradient Norm	5000

After training, we find our best model is one with the above parameters and $\gamma = 0.25$. Given that our algorithm is looking to maximize the reward function and not necessarily to correctly guess the classification (though, of course, the more it can do that, the higher the reward should be), we do not feel a confusion matrix is appropriate. In a similar vein as the ANNs, there is randomness in the training, but not in the implementation, so we will train this four times and run each model through the testing environment, giving us an average reward of close to 500.

A typical training session with these parameters looks like:



ii. Deep Deterministic Policy Gradient

For the second and final DRL algorithm, we need to take a small detour to talk about something we glossed over in the beginning. Recall that in using neural networks to find Q-values and then choosing a policy based on those, we are not solving for the policy directly. Is there a way to circumvent all this business with the Q-values and go directly to the policy? Yes, there exists a whole family of algorithms to do this, called *Policy Gradient* algorithms. However, we will not be implementing them in this paper for reasons that will be made clear soon. Before we skip over them, though, we need to understand a few fundamentals about policy gradient algorithms first, as we will be combining Q-Learning and policy gradient methods for our final algorithm.

- Policy Gradients

Recall that we decided to use Q-learning, as opposed simpler Dynamic Programming methods, because of the continuous nature of our state space. Of course, with our current formulation, we are only either buying or selling the full amount of our investment, so our action space $\mathcal{A} = \{Buy, Sell\}$ is discreet (and quite small). What would we need to do if we wanted to invest or sell only a *portion* of our investment at each step? In this case, depending on exactly how we formulated this, the number of actions possible in each state would grow very large or be functionally continuous. To deal with this, we will use policy gradient algorithms.

To begin, we will use approximation notation for our policy, which is normally *stochastic* in nature [22]:

$$a \sim \pi(s) \approx \hat{\pi}(s, \boldsymbol{\theta})$$

Now, this policy could be deterministic or continuous. When it is continuous with dimension d, our action will be $\boldsymbol{a} \in \mathbb{R}^d$, we can write it as [22]:

$$\hat{\pi}(s, \boldsymbol{\theta}) \sim \mathcal{N}(\boldsymbol{\mu}(s, \boldsymbol{\theta}), \sigma^2(s, \boldsymbol{\theta})\boldsymbol{I}_d) = \mathcal{N}_{\boldsymbol{\theta}}(s)$$

So, the output of this is then a d-dimensional vector $\boldsymbol{a}$ where each component $a_i \sim \mathcal{N}(\mu_i(s, \boldsymbol{\theta}), \sigma_i^2(s, \boldsymbol{\theta}))$ for $i = 1, \dots, d$.

When it is deterministic with $|\mathcal{A}|$ actions, it is written as [22]:

$$\hat{\pi}(s, \boldsymbol{\theta})_i = \frac{e^{h(s, a_i, \boldsymbol{\theta})}}{\sum_{i=1}^{|\mathcal{A}|} e^{h(s, a_i, \boldsymbol{\theta})}}$$

In the above, $h(s, a, \boldsymbol{\theta})$ is the *logits* or *action preferences*, and the function $\frac{e^{h(s, a_i, \boldsymbol{\theta})}}{\sum_{i=1}^{|\mathcal{A}|} e^{h(s, a_i, \boldsymbol{\theta})}}$ is the *Softmax* function. Thus, $\boldsymbol{a}$ is a vector where each component $a_i = \frac{e^{h(s, a_i, \boldsymbol{\theta})}}{\sum_{i=1}^{|\mathcal{A}|} e^{h(s, a_i, \boldsymbol{\theta})}}, i = 1, \dots, |\mathcal{A}|$.

For this project, our action will be a one-dimensional, continuous *deterministic* value, so we will use the notation [23]:

$$\hat{\pi}(s, \boldsymbol{\theta}) = \mu(s, \boldsymbol{\theta})$$

As with the Q-values case, we are now estimating our policy with respect to parameters $\boldsymbol{\theta}$**.** Note here that we will use $\boldsymbol{\Theta}$ to refer to weights associated with Q-values and $\boldsymbol{\theta}$ to refer to weights associated with policies.

Now, we will not be focusing on the loss function which we seek to minimize, but on the *Rewards Function* we seek to *maximize* [24]:

$$R(\boldsymbol{\theta}) = \sum_{t} \gamma^{t} r_{t+1}$$

Here, $r_{t+1} = r(s_t, a_t)$. We note that this $R(\boldsymbol{\theta})$ can have a finite or infinite horizon (resulting in an ∞ or $T-1$ in the limit of the summation) and can be discounted or not (giving us $\gamma = 1$). We could also formulate it as an average and take a limit as the number of instances goes to infinity.

From here, we look at the trajectory of the agent τ, starting at time $t = 1$ and ending at $t = T$ [25]:

$$s_1 \rightarrow a_1 \rightarrow \cdots \rightarrow s_{T-1} \rightarrow a_{T-1} \rightarrow s_T \rightarrow a_T$$

We then ask for the probability of this trajectory given our parameters $\boldsymbol{\theta}$ [26]:

$$p_{\boldsymbol{\theta}}(\tau) = p_{\boldsymbol{\theta}}(s_1, a_1, \ldots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \hat{\pi}(s_t, \boldsymbol{\theta})\, p(s_{t+1} | s_t, a_t)$$

Note here that we could pull out $p(s_1)$ from the rest, as that does not depend on $\hat{\pi}$; it is just the state the agent first finds itself in when it begins the trajectory τ.

Now, the expected rewards from following τ using a policy based off parameters $\boldsymbol{\theta}$ is [27]:

$$J(\boldsymbol{\theta}) = R_{expected}(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim p_{\boldsymbol{\theta}}(\tau)} \left[\sum_{t=1}^{T} \gamma^{t-1} r_{t+1} \right]$$

We write $\gamma = \gamma^{t-1}$ so that at $t = 1$, the first step, there is no discount, as $\gamma^0 = 1$. If we started at $t = 0$, we could write $\gamma = \gamma^t$, but we have chosen to follow the time convention in the description of τ, which begins at $t = 1$. We are thus asking for $\boldsymbol{\theta}$ that maximizes $R_{expected}(\boldsymbol{\theta})$. For ease of notation, we will refer to $R_{expected}(\boldsymbol{\theta})$ as $R(\boldsymbol{\theta})$, as that is what we are interested in. We also need to write this expected Reward Function in Monte Carlo form, as we will be sampling to find it [28]:

$$J(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \gamma_i^{t-1} r_{t+1}^{i}$$

Here, $r_{t+1}^{i} = r(s_t^i, a_t^i)$, and is the reward earned at time $t+1$ in our trajectory from taking action $a = a_t$ in state $s = s_t$ in the i^{th} Monte Carlo iteration.

Now, to update our $\boldsymbol{\theta}$, we need an update equation, and thus need $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. *Deep Reinforcement Learning with Python*, pages 214 through 215, offer an excellent derivation of this, which we skip for brevity.

However, the final formulations in both expected value and Monte Carlo forms are [29]:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim p_{\boldsymbol{\theta}}(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \log \left(\hat{\pi}(s_t, \boldsymbol{\theta}) \right) \right) \left(\sum_{t=1}^{T} \gamma^{t-1} r_{t+1} \right) \right]$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{i=1}^{N} \left[\left(\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \log \left(\hat{\pi}(s_t^i, \boldsymbol{\theta}) \right) \right) \left(\sum_{t=1}^{T} \gamma_i^{t-1} r_{t+1}^i \right) \right]$$

Finally, we will use *Gradient Ascent*, not *Descent*, to update our parameters [30]:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \eta \nabla_{\boldsymbol{\theta}_k} J(\boldsymbol{\theta}_k)$$

Here, we need to be careful, as we have switched from updating $\boldsymbol{\theta}$ at time $t + 1$ to a new index $k + 1$. We chose this because for each iteration of $\boldsymbol{\theta}$, we run N Monte Carlo simulations/trajectories, and *in each trajectory*, we run through T steps. So, we don't want to use t again to avoid confusion between our trajectory steps in a single Monte Carlo simulation and our parameter updating scheme.

We note here that algorithms which use policy gradient methods are call *Actor* algorithms, as our agent *acts* with regards to the policy it has learned. Algorithms which use both a value function and policy gradients are call *Actor-Critic* models, as these have an actor act according to a policy and then have that policy critiqued by the value-function. At this point, if we were going to talk about using policy gradient algorithms, we would begin to describe algorithms like *REINFORCE*, *Advantage Actor-Critic* (A2C), *Asynchronous Advantage Actor-Critic* (A3C), *Trust Region Optimization Policy* (TRPO), *Proximal Policy Optimization* (PPO), and others. However, we note that all of these are on-policy [31].

We will now combine Deep Q-Networks, an off-policy critic algorithm, with policy gradients, an actor one, giving us *Deep Deterministic Policy Gradient* (DDPG). This is a model-free, actor-critic algorithm suitable for continuous state and action-spaces off-policy learning.

- Deep Deterministic Policy Gradients

Recall that in Q-Learning, if we have the optimal policy, π_*, we can find the optimal action/actions to take in any given state by [32]:

$$a^*(s) = \max_a Q^{\pi_*}(s, a)$$

However, we are now approximating our policy with a continuous and *deterministic* function $a = \hat{\pi}(s, \boldsymbol{\theta}) = \mu(s, \boldsymbol{\theta})$. Note that if we have multiple actions (such as movement in two-dimensional space), $a = \boldsymbol{a} \in \mathbb{R}^d$, where d is the number of dimensions the action can take.

So, if we have the optimal, or approximately optimal, parameters $\boldsymbol{\theta}$, we have a function $\mu(s, \boldsymbol{\theta})$ which will give us $a^*(s)$.

We thus use one neural network with parameters $\boldsymbol{\theta}$ for our policy to find $\mu(s, \boldsymbol{\theta})$ and then feed that into the second one with parameters $\boldsymbol{\Theta}$ for our Q-values to get $\hat{Q}(s, \mu(s, \boldsymbol{\theta}), \boldsymbol{\Theta})$. When both sets of parameters have been properly optimized, with will have an optimum policy that will yield optimal Q-values [32].

Just as with the DQN case, wherein we needed some stationary target to aim for, and thus kept a second neural network with parameters updated less frequently, we will have a target neural network for both $\boldsymbol{\theta}$ and $\boldsymbol{\Theta}$. However, we will not be updating them after a certain number of batches, but at the same time as we update the online network with the *Polyak Averaging* [33]:

$$\boldsymbol{\Theta}_{\mathrm{t+1}}^{\mathrm{target}} \leftarrow \rho \boldsymbol{\Theta}_{\mathrm{t}}^{\mathrm{target}} + (1-\rho)\boldsymbol{\Theta}_{t}$$

$$\boldsymbol{\theta}_{\mathrm{t+1}}^{\mathrm{target}} \leftarrow \rho \boldsymbol{\theta}_{\mathrm{t}}^{\mathrm{target}} + (1-\rho)\boldsymbol{\theta}_{t}$$

Now, we need to describe our online network update equations. Focusing on the Q-Learning aspect, we have our original loss equation from earlier in expected value and Monte Carlo forms as:

$$\mathcal{L}(\boldsymbol{\Theta}) = \mathbb{E}\left[\left(\mathrm{Q}^{\pi}(s,a) - \hat{Q}(s,a,\boldsymbol{\Theta})\right)^{2}\right]$$

$$\mathcal{L}(\boldsymbol{\Theta}) \approx \frac{1}{\mathrm{N}}\sum_{\boldsymbol{i=1}}^{\boldsymbol{N}}\left(\mathrm{Q}^{\pi}(s_i,a_i) - \hat{Q}(s_i,a_i,\boldsymbol{\Theta})\right)^{2}$$

However, our target $\mathrm{Q}^{\pi}(s_i,a_i)$ has changed, and is now [34]:

$$Y_{DDPG}^{target} = r_i + (1-done_i)\gamma\, \tilde{Q}\left(s_i', \mu\left(s_i', \boldsymbol{\theta}_t^{target}\right), \boldsymbol{\Theta}_{\mathrm{t}}^{target}\right)$$

We can thus write the Q-Learning loss function in DDPG, which is only a function of $\boldsymbol{\Theta}$ and $\mathcal{D}$, as: [34]

$$\mathcal{L}(\boldsymbol{\Theta}_{\mathrm{t}},\mathcal{D}) = \mathbb{E}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left((\mathrm{r} + (1-done)\gamma\, \tilde{Q}(s', \mu\left(s', \boldsymbol{\theta}_t^{target}\right), \boldsymbol{\Theta}_{\mathrm{t}}^{target}) - \hat{Q}(s,a,\boldsymbol{\Theta}_{\mathrm{t}})\right)^{2}\right]$$

$$\mathcal{L}(\boldsymbol{\Theta}_{\mathrm{t}},\mathcal{D}) \approx \frac{1}{|\mathcal{B}|}\sum_{(s,a,r,s',d)\in B}\left[\left((\mathrm{r}_i + (1-done_i)\gamma\, \tilde{Q}(s_i', \mu\left(s_i', \boldsymbol{\theta}_t^{target}\right), \boldsymbol{\Theta}_{\mathrm{t}}^{target}) - \hat{Q}(s_i,a_i,\boldsymbol{\Theta}_{\mathrm{t}})\right)^{2}\right]$$

Here, (s,a,r,s',d) is a tuple of information gained by exploring the environment and comes from some distribution $\mathcal{D}$, and $\mathcal{B}$ is a batch of them used in the Monte Carlo formulation.

We would then update the weights with the standard gradient descent update equation:

$$\boldsymbol{\Theta}_{t+1} = \boldsymbol{\Theta}_{t} - \alpha \nabla_{\boldsymbol{\Theta}_{\mathrm{t}}} L(\boldsymbol{\Theta}_{\mathrm{t}},\mathcal{D})$$

For the policy learning portion, we note that we are seeking to find $\boldsymbol{\theta}$ to maximize the Q-function. So, we want $\boldsymbol{\theta}_t$ that maximizes our function J [35]:

$$J(\boldsymbol{\theta}_t,\mathcal{D}) = \mathbb{E}_{s\sim\mathcal{D}}\left[\hat{Q}(s,\mu(s,\boldsymbol{\theta}_t),\boldsymbol{\Theta}_{\mathrm{t}})\right]$$

$$J(\boldsymbol{\theta}_t,\mathcal{D}) \approx \frac{1}{|\mathcal{B}|}\sum_{s\in\mathcal{B}}\hat{Q}(s,\mu(s,\boldsymbol{\theta}_t),\boldsymbol{\Theta}_{\mathrm{t}})$$

So, we can take the gradient of $J(\boldsymbol{\theta}, \mathcal{D})$ and use gradient ascent:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \beta \nabla_{\boldsymbol{\theta}_t} J(\boldsymbol{\theta}_t, \mathcal{D})$$

We now have almost all the machinery for DDPG. However, recall that with DQN, we wanted our agent to explore early during training to visit and evaluate as many action-state pairs as possible. We will apply the same idea here. However, as our actions are continuous, we will take the actual action and add some noise to it, to get $a + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma^2)$, or, in the multi-dimensional case, $\boldsymbol{a} + \boldsymbol{\epsilon}$, where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$, [33, 36].

- Twin Delayed Deep Deterministic Policy Gradient

Recall that in our discussion of DQNs, we ran into an issue of the maximization bias. Given the structure of our DDPG, we will have the same problem. So, we will modify our vanilla DDPG in several ways, leading to *Twin Delayed Deep Deterministic Policy Gradient* (T3D). The first is called *Target Policy Smoothing*.

In our DDPG formulation, when we calculate the targets, we have the following equation:

$$Y_{DDPG}^{target} = r_i + (1 - done_i)\gamma \, \tilde{Q}\left(s'_i, \mu\left(s'_i, \boldsymbol{\theta}_t^{target}\right), \boldsymbol{\Theta}_{\mathrm{t}}^{target}\right)$$

Recall also that we add a noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$ to the action when we take it. However, that modification is not present in our target calculation. In Target Policy Smoothing, we will still have that noise added to the action when we take it, but instead of a simple calculation of $a'_i = \mu\left(s'_i, \boldsymbol{\theta}_t^{target}\right)$, which we would then evaluate in our $\tilde{Q}$, we instead use the following [37]:

$$a'_i = clip\left(\mu\left(s'_i, \boldsymbol{\theta}_t^{target}\right) + clip(\epsilon, -c, c), a_{low}, a_{high}\right)$$

The goal of this is to make it harder for the policy to exploit Q-function estimation errors.

We will also use *two independent Q-functions* and calculate our target from the minimum of them. This makes our target [38]:

$$Y_{T3D}^{target} = r_i + (1 - done_i)\gamma \min_{j=1,2} \tilde{Q}\left(s'_i, a_i', \boldsymbol{\Theta}_{\mathrm{j,t}}^{target}\right)$$

So, we are choosing which of the two target Q-value networks yields the minimum for our target.

Our Q-value losses are modified [39]:

$$Q_{Loss,1} = \frac{1}{|\mathcal{B}|} \sum_{(s,a,r,s',d) \in \mathcal{B}} \left(Y_{T3D}^{target} - \hat{Q}(s, a, \boldsymbol{\Theta}_1)\right)^2$$

$$Q_{Loss,2} = \frac{1}{|\mathcal{B}|} \sum_{(s,a,r,s',d) \in \mathcal{B}} \left(Y_{T3D}^{target} - \hat{Q}(s, a, \boldsymbol{\Theta}_2)\right)^2$$

$$Q_{Loss} = \sum_{j=1,2} Q_{Loss,j}$$

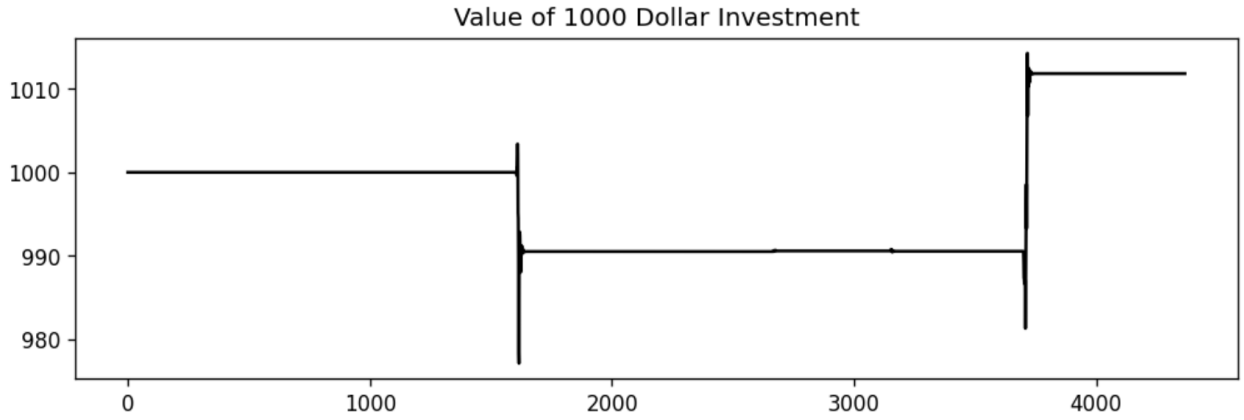
This is where the *Twin* in the T3D comes from.

The *Delayed* comes from the fact that we will no longer be updating our target policy and agent parameters at the same time as the online ones, but only every *other* online policy and agent network update.

Thus, our DDPG algorithm will have both modifications to become a T3D. Our algorithm is then a *Convolutional Neural Network-Based Twin Delayed Deep Deterministic Policy Gradient*.

We maintain the same reward system from the previous RL algorithm, but here, we found that we had to modify the hyperparameters just to get the algorithm to learn. It would often start with a negative reward in the beginning of training, then slowly build to zero reward, then stay there. The way we interpreted this behavior is that the algorithm was not able to learn how to *increase* the portfolio, but *did* learn how *not to decrease it*, by simply not doing anything.

When we took the trained model plateauing at a reward of zero and applied to the testing environment, we would see something like the following:



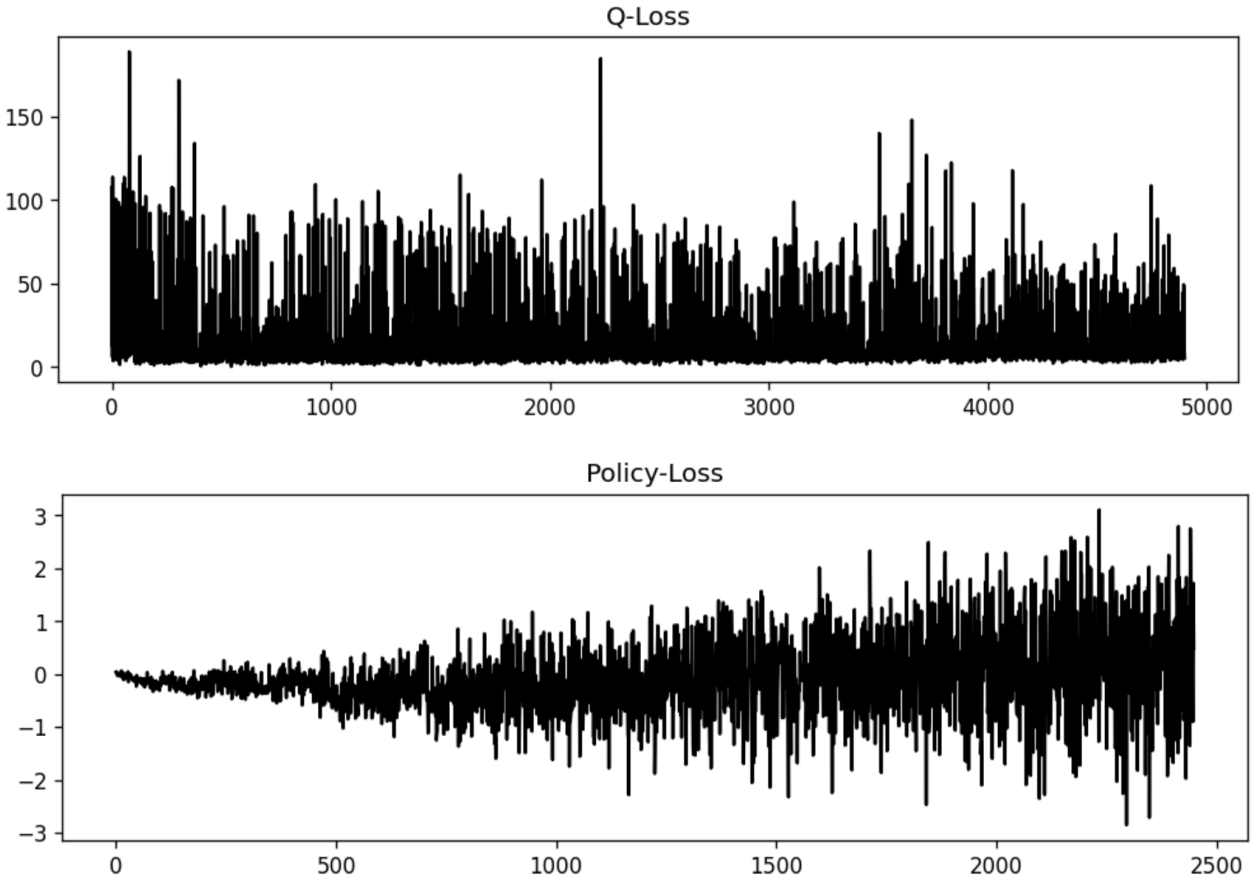
We see it is making very few decisions over the testing period, showing extreme conservativism. While this could be an advantage in some situations, we would like to have our algorithm trading more often.

After significant tweaking of the hyperparameters by hand, mostly by increasing the noise to force the algorithm to accidently find a good decision and learn from it, we have the following parameters:

Hyperparameters that remain the same	Type/Number
Steps per epoch	1000
Epochs	5
Replay Size	1×10^6
Polyak	0.995
Policy Learning Rate	1×10^{-5}
Q-function Learning Rate	1×10^{-5}
Batch Size	16
Start Steps	10000
Update After (How many steps we take before we begin updating)	100
Update Every (How often we update *after* the Update After steps)	50
Act Noise	0.2
Target Noise	0.1
Noise Clip	0.6
Policy Delay	2
Number of Test Episodes	10
Max Episode Length	100
Policy and Q-Function Optimizer	Adam

We find the best algorithm is one where $\gamma = 1$, indicating that our model does best when it considers *each reward to be equally important*.

We can also see the Q-loss and Policy-loss through a typical training session with $\gamma = 1$:



Here, we can see a slight overall decline in the Q-loss, but a growing oscillation in the Policy-loss, which can help us fine-tune the hyperparameters. We then train four models with these hyperparameters and find an average reward of the high 300s. Though that is not as high an average as we found with the DQN model, when we see the investment results, we will see the advantages of this model over the DQN. Another interesting note is that this algorithm often quickly found a local reward maximum per epoch and stayed there, sometimes in the first training epoch. This made it the fastest algorithm to train, as it often took exactly one epoch to find this place, where it would then settle. Sometimes, it was 0, which we would then simply restart the training, but once it found one, it rarely changed. By reducing the learning rate, we could see the learning more clearly, watching the rewards per epoch increasing, but it would still just reach the original (and always the same) and remain fixed. Perhaps with more tweaking of hyperparameters, we might be able to get rewards per epoch past this value.

7. Investing Results

In the last section, we focused on the mathematical and computational aspects of the various algorithms in use and presented their initial performance on more abstract metrics, such as accuracy. In this section, we give each algorithm the responsibility of making trading decisions for us. The set-up is simple, as all non-RL algorithms but one will either output a 1, indicating *Buy*, or a 0, indicating *Sell*. If the algorithm is a regression-based one, a positive output will be treated as *Buy* and a negative one *Sell*. The RL algorithms will be left to play in their environment and their performance is based on their success there.

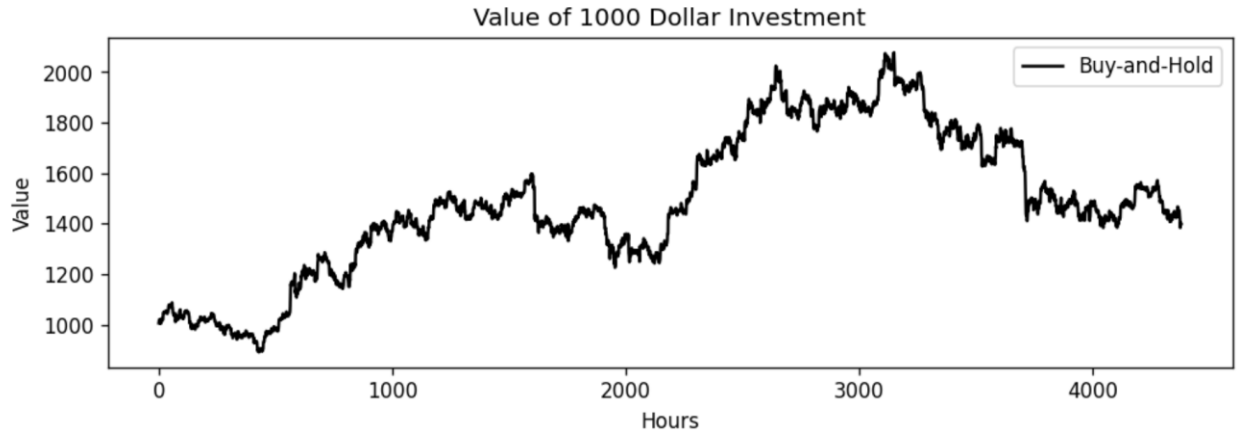
We begin by providing a baseline, wherein we explore some basic strategies and their performance. All agents begin with 1000 dollars and will trade for roughly six months, or 4380 hours, the entire testing duration. We note that because some of the algorithms, like the CNN and LSTM, required past data to be built into their data instances. For example, the first 11 data instances are grouped with the 12^{th} to create an "image." So, instead of there being 4380 testing instances, there are only $4369 - 1 = 4368$. Due to the method in which the images and sequences are generated, the final hour is not present, either. Hence, the -1. This is the case with all CNN and LSTM algorithms, and we mention in their respective sections what the testing timeframe is.

The agents' performance is determined both by the final dollar amount, but also in relation to the simplest strategy, *Buy-and-Hold*, wherein one simply invests in an asset and waits. This means that an agent that beats Buy-and-Hold might not necessarily be deemed successful depending on how the two interact. For example, an algorithm might spend most of the six months *below* what the investor would have earned had they just used Buy-and-Hold, though it might ultimately end up higher, perhaps rallying in the last weeks. In real trading, we do not have an *end*, so, though it might work out in our favor in this instance, it could be lower depending on when we decide to check it and/or leave the market. Let us begin.

A. Buy-and-Hold

Had an investor bought 1000 dollars' worth of BTC on July 2nd of 2021 at 11:00 AM and held their asset, not buying or selling, until December 31st of 2021 at 10:00 PM, the value of their investment would look like:

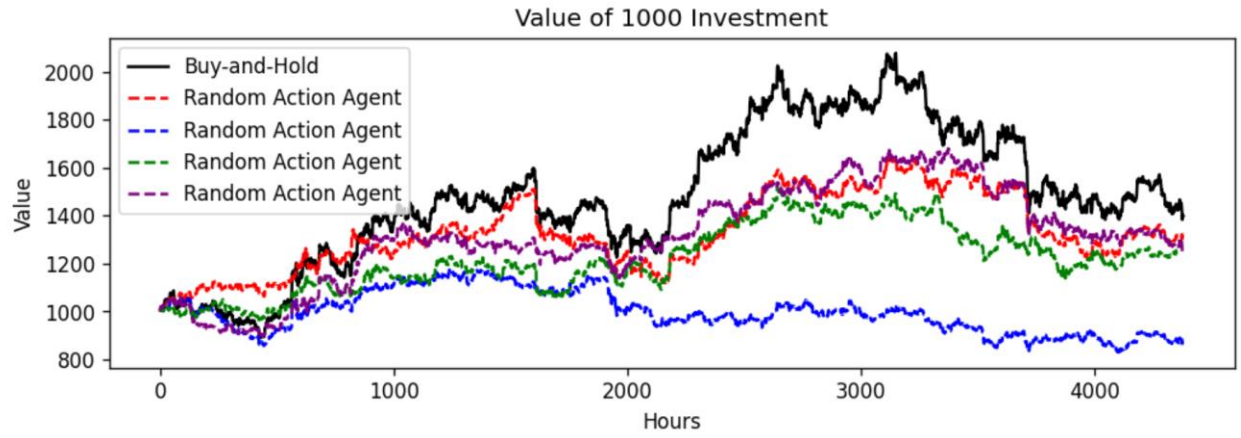
```
The final value with Buy-and-Hold is 1399.21
```



B. Random Agent

We run a random agent four times for comparison.

```
The final value with Buy-and-Hold is 1399.21
The final value with Random Agent is 1322.17
The final value with Random Agent is 866.85
The final value with Random Agent is 1276.3
The final value with Random Agent is 1244.23
```



Obviously, each time we run this, we get a different behavior, but it is important to have a baseline of what randomness looks like in comparison to Buy-and-Hold.

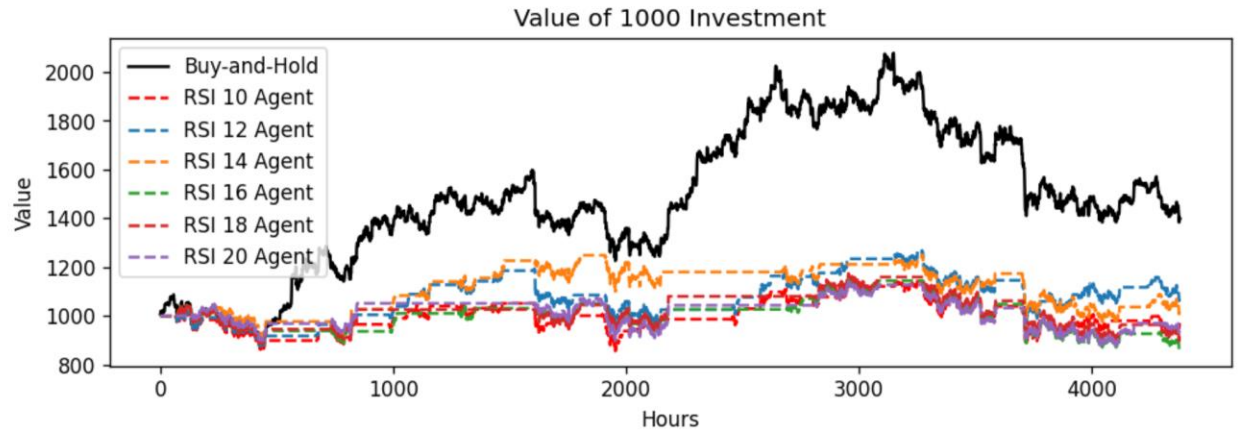
C. Technical Indicator Agents

Now, we explore agent trading based off a few of the technical indicators used in our data matrix creation.

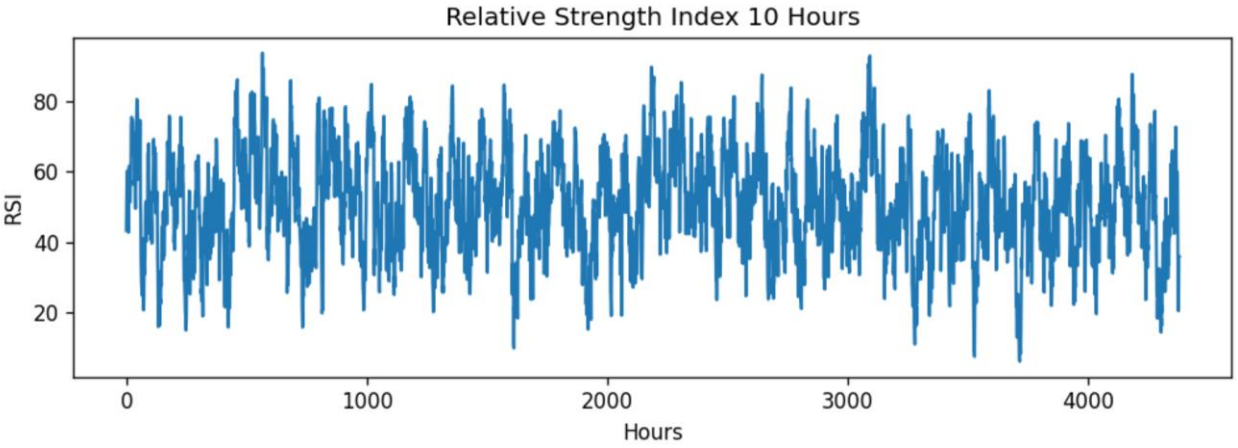
i. Relative Strength Index

For our data matrix creation, we used this technical indictor for 10, 12, 14, 16, 18, and 20 hours. So, an agent investing with this indicator, wherein it would buy below a score of 30, hold between 30 and 70, and sell at 70, would have the following performance:

```
The final value with Buy-and-Hold is 1399.21
The final value with RSI 10 is 960.83
The final value with RSI 12 is 1073.62
The final value with RSI 14 is 1019.09
The final value with RSI 16 is 873.04
The final value with RSI 18 is 907.68
The final value with RSI 20 is 933.14
```



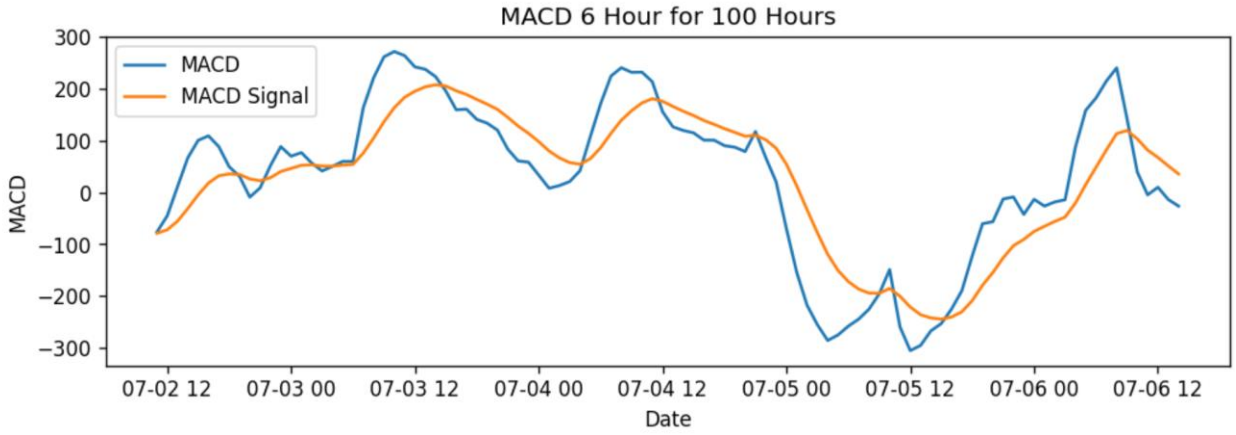
It appears the agents simply did not act often. Looking at the data, we see most of the time this indictor was between 30 and 70, so this is not a surprise. If we look at one instance of the RSI, say, for 10 hours, we get the following:



We see that it rarely goes too high or too low, corresponding to large flat sections in the performance of the agent trading with this indicator.

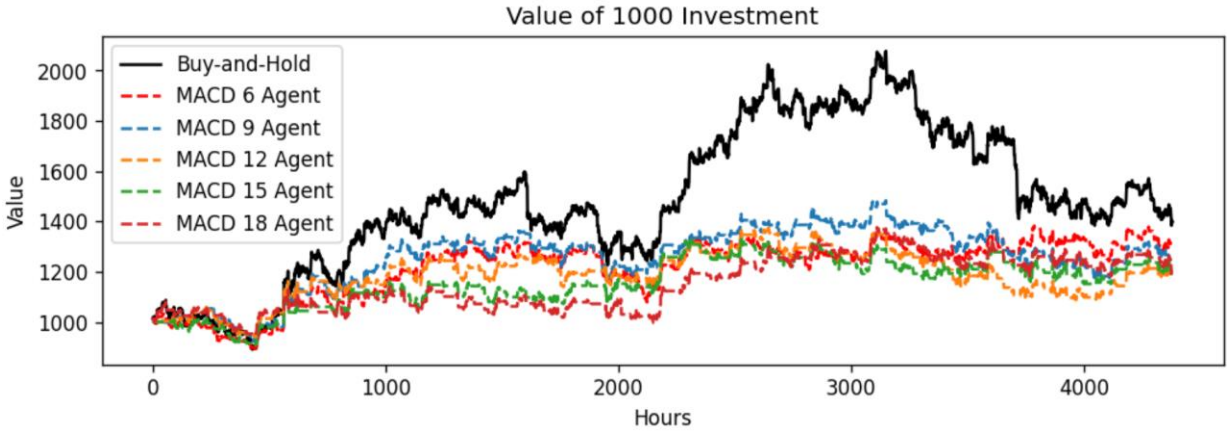
ii. Moving Average Convergence Divergence

For this indicator, we should note that there are a few ways to trade with it. We implement the simplest version, wherein we have two lines, the *MACD* and the *Signal*, and their crossing is taken to be a signal. If the MACD line crosses the Signal from below, that will be a Buy signal, and vice versa. Looking at just a few hours, we can see their interaction:



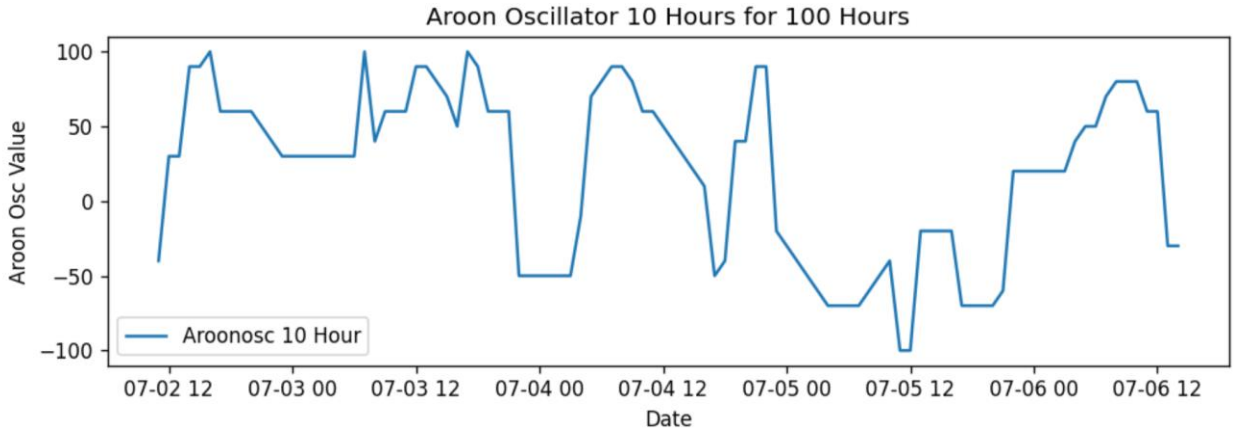
However, we note here that to determine the actions an agent would take trading with MACD, we need to look one time step into the past to compare it to the present to determine if a cross has happened. This means that the 0th hour (July 2nd of 2021 at 11:00 AM) cannot have an action, so our agent is now trading from July 2nd of 2021 at 12:00 PM to December 31st of 2021 at 10:00 PM. The performances are:

```
The final value with Buy-and-Hold is 1399.21
The final value with MACD 6 is 1313.5
The final value with MACD 9 is 1236.47
The final value with MACD 12 is 1187.55
The final value with MACD 15 is 1193.53
The final value with MACD 18 is 1191.22
```



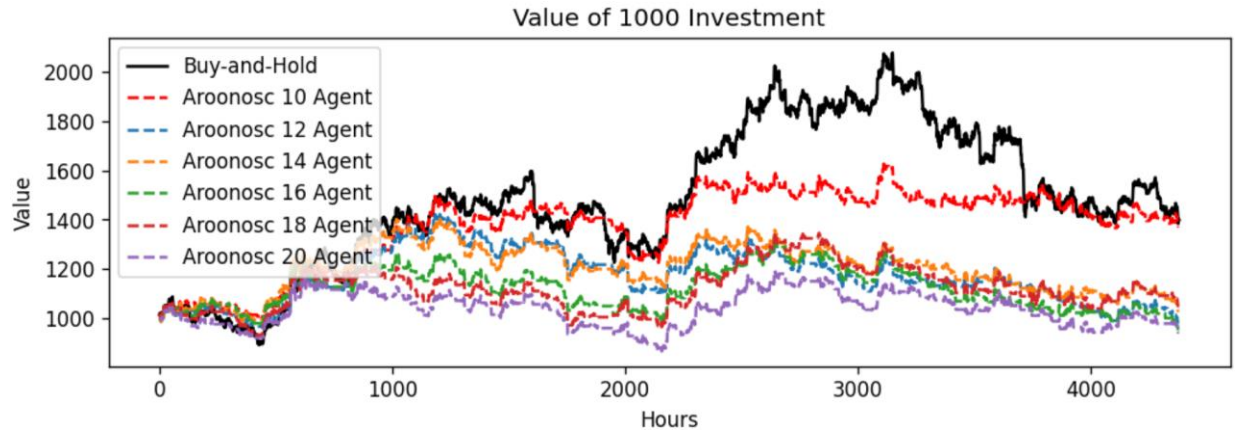
iii. Aroon Oscillator

With this technical indicator, we again look for crossings, this time when the line crosses the zero. Crossing from the negative is a Buy signal and from the positive is a Sell signal. Let's see the first 100 hours of this indicator:



To determine the actions an agent would need to take trading with this indicator, we adopt the same method as with MACD, meaning that our agent is not trading on the first hour, same as above. Our results are:

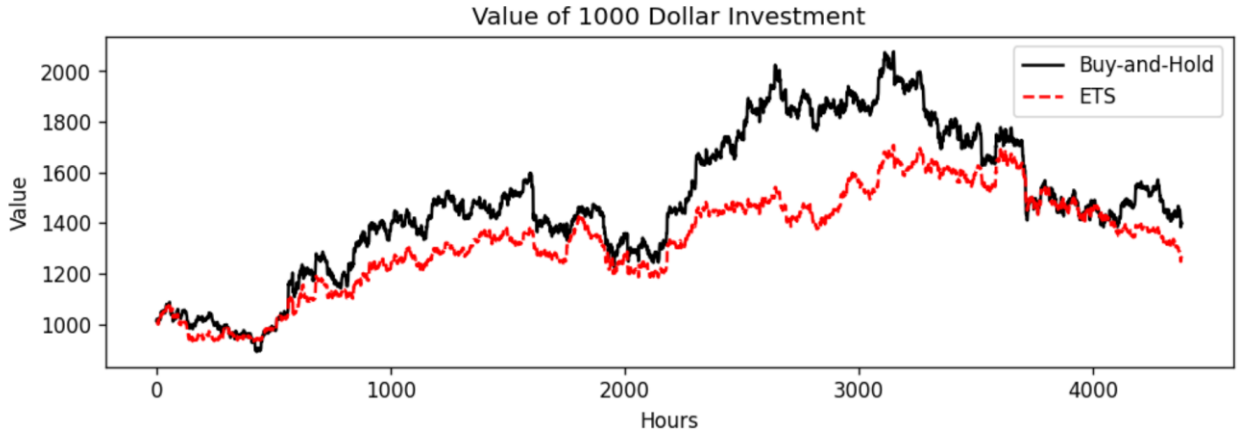
```
The final value with Buy-and-Hold is 1399.21
The final value with Aroonosc 10 is 1373.46
The final value with Aroonosc 12 is 975.8
The final value with Aroonosc 14 is 1029.37
The final value with Aroonosc 16 is 955.98
The final value with Aroonosc 18 is 1049.27
The final value with Aroonosc 20 is 942.16
```



D. Exponential Smoothing

For this algorithm, we note that it works by forecasting the next time step from the beginning until the requested number of future time steps. As such, the first forecast of our agent will be for the price of July 2nd of 2021 at 12:00 PM and will forecast until one step after the final hour of our testing dataset, December 31st of 2021 at 11:00 PM. Of course, since our last datum is for December 31st at 10:00 PM, we will remove the final hour of our ETS forecast, as we cannot judge that value. Now, using the ETS algorithm trained earlier, we have the following performance:

```
The final value with Buy-and-Hold is 1399.21
The final value with Exponential Smoothing is 1261.26
```

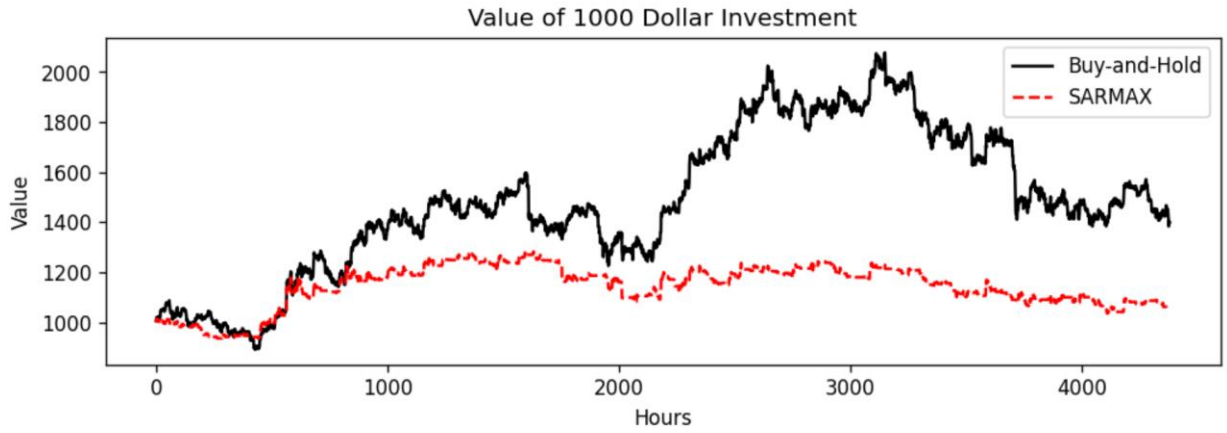


This is very similar to the random agent, and, recalling the accuracy to be 50.92%, this is not a surprise.

E. SARIMAX

The performance of our $ARIMA(2,0,2)(1,0,1,24)$ model is:

```
The final value with Buy-and-Hold is 1399.21
The final value with SARIMAX is 1051.14
```

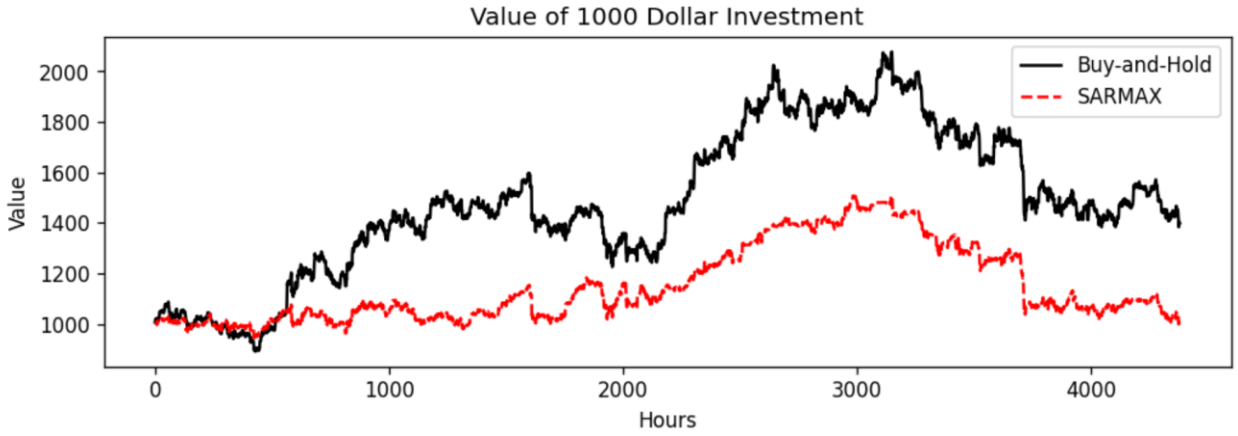


This is significantly poorer, and honestly, quite a surprise. It appears to have actively *learned the wrong thing*, as its accuracy is 47.37%. Further tuning is needed to determine the extent to which this algorithm might improve, though some suggestions are given in the relevant training section.

F. XGBoost

The performance of our XGBoost agent is:

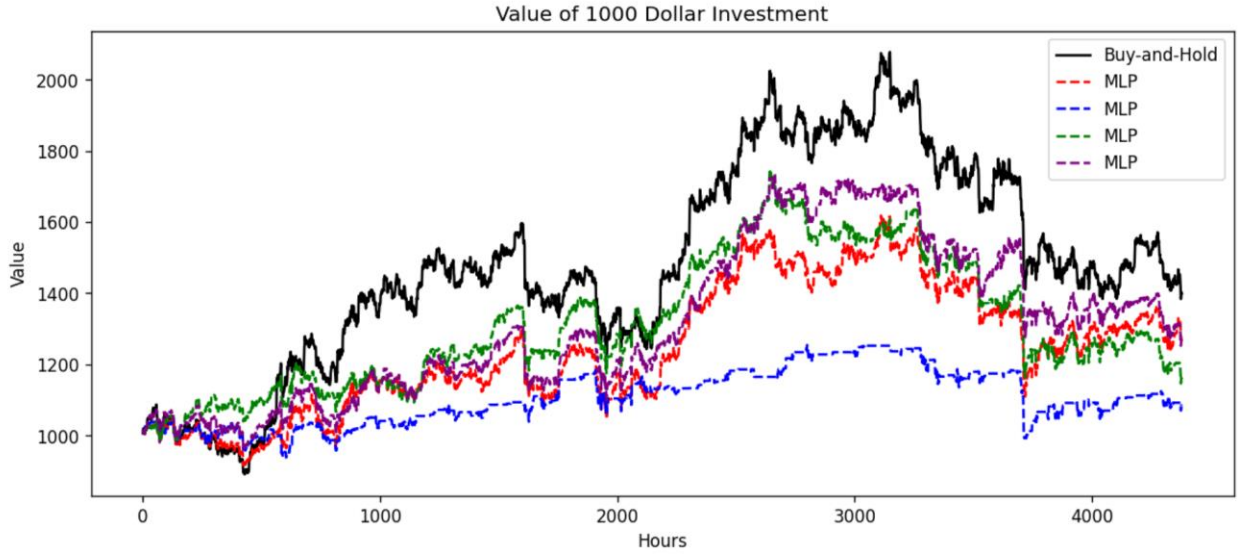
```
The final value with Buy-and-Hold is 1399.21
The final value with XGBoost is 1009.89
```



G. Multi-Layer Perceptron

The performance of our MLP agents trained with the same architecture are:

```
The final value with Buy-and-Hold is 1399.21
Our accuracies and final values for these models are:
 52.0 percent and 1282.51 dollars,
 52.0 percent and 1082.19 dollars,
 53.0 percent and 1159.06 dollars,
 53.0 percent and 1266.32 dollars.
```

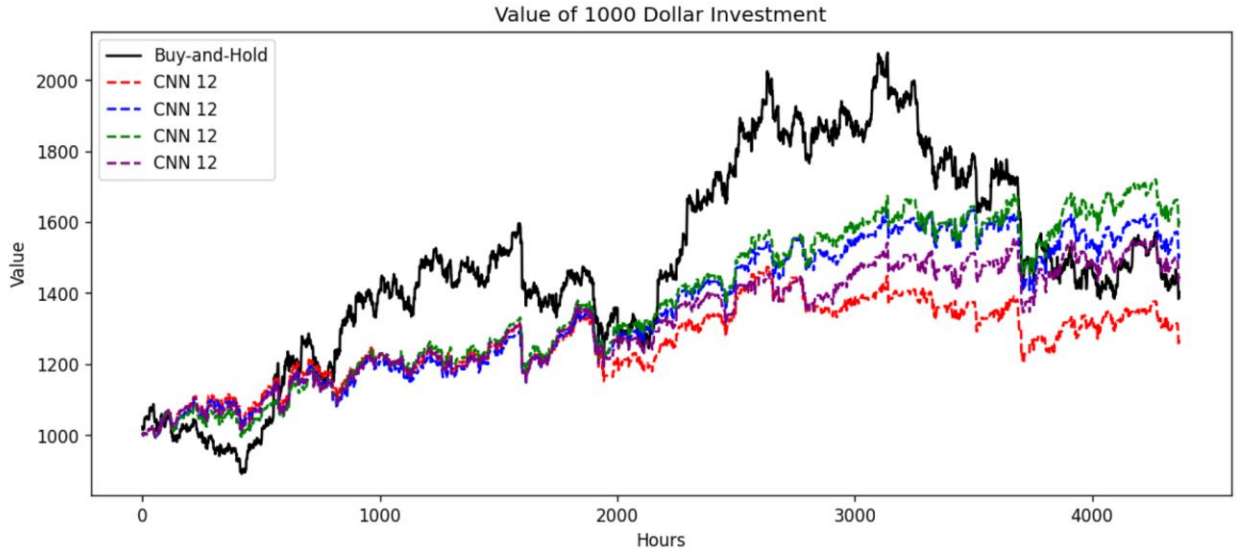


They generally have the same shape, but we do see, especially with the blue one, that having a significant imbalance in class predictions can negatively affect the performance of an agent.

H. Convolutional Neural Network

Here, as noted in the opening of this section, we needed to aggregate the first for 12 hours into an "image," and our "image" creation algorithm does not include the final hour of the dataset given to it. As such, our agent is trading from July 2nd of 2021 at 10:00 PM to December 31st at 9:00 PM. The performance of our CNNs with 12 data-instance "images" is:

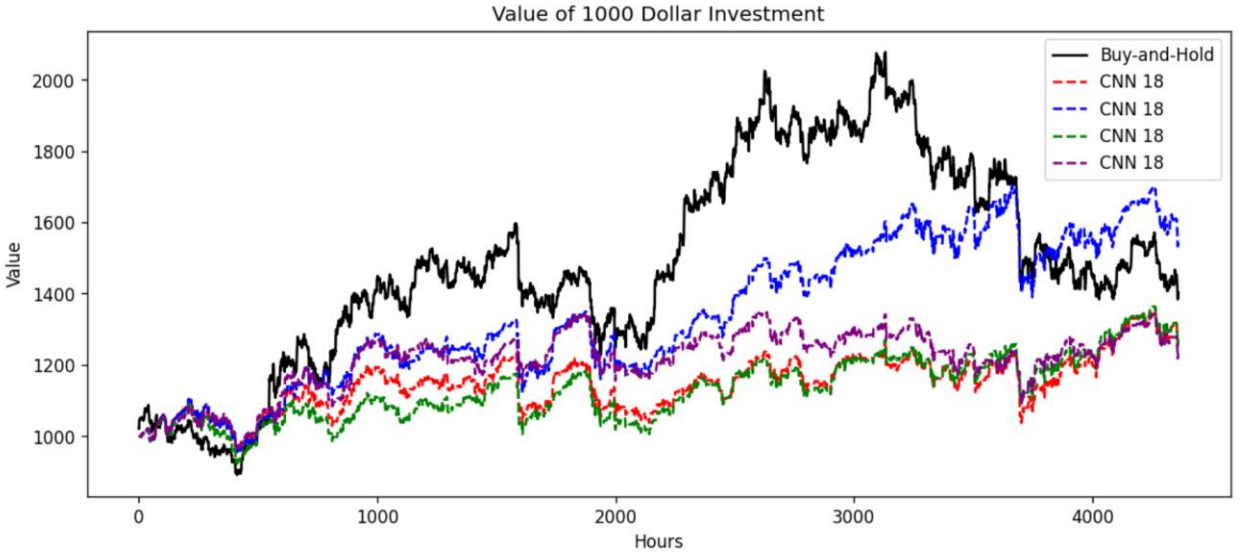
```
The final value with Buy-and-Hold is 1402.49
Our accuracies and final values for these models are:
 53.0 percent and 1268.99 dollars,
 54.0 percent and 1518.36 dollars,
 54.0 percent and 1605.3 dollars,
 54.0 percent and 1448.82 dollars.
```



We note here again that the final value of Buy-and-Hold is slightly more than above, as we are removing the final value so the lengths of our trading period with our CNN-based agent and our Buy-and-Hold are the same. We see that, in general, even the worst CNN agent did roughly as well as the best MLP agent, and three outperformed the Buy-and-Hold. We attribute this to more even class balance overall in trainings. However, we note that though several finished higher, the fact that all of them completely failed to take advantage of the large increases around the 1500 and 2500-hour mark is worrying.

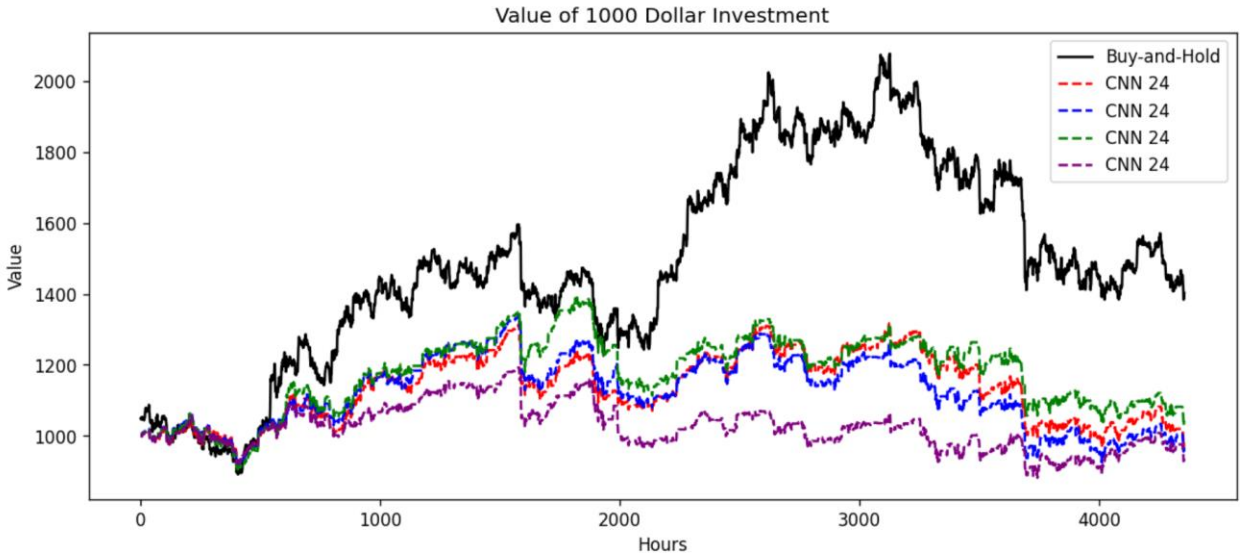
For our 18 data-instance "images," beginning July 3rd of 2021 at 4:00 AM and finishing on December 31st at 9:00 PM, we have:

```
The final value with Buy-and-Hold is 1402.49
Our accuracies and final values for these models are:
 53.0 percent and 1266.6 dollars,
 54.0 percent and 1553.55 dollars,
 53.0 percent and 1271.88 dollars,
 53.0 percent and 1234.28 dollars.
```



For our 24 data-instance "images," beginning on July 3rd of 2021 at 10:00 AM and finishing on December 31st at 9:00 PM, we have:

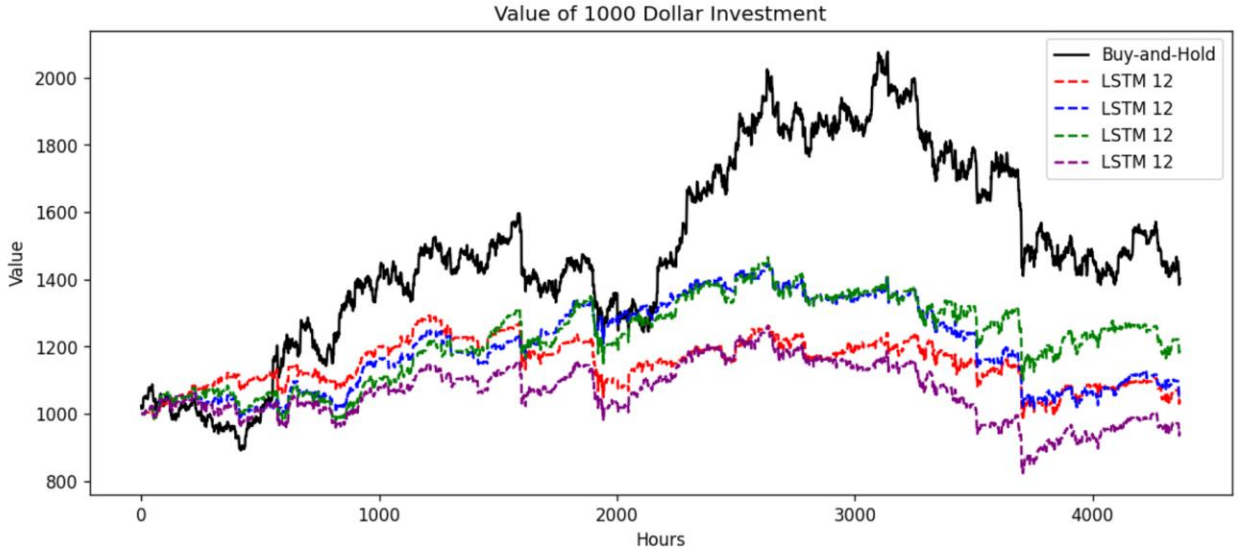
```
The final value with Buy-and-Hold is 1402.49
Our accuracies and final values for these models are:
 53.0 percent and 982.91 dollars,
 52.0 percent and 968.54 dollars,
 53.0 percent and 1043.2 dollars,
 52.0 percent and 940.81 dollars.
```



I. Long-Short Term Memory Recurrent Neural Network

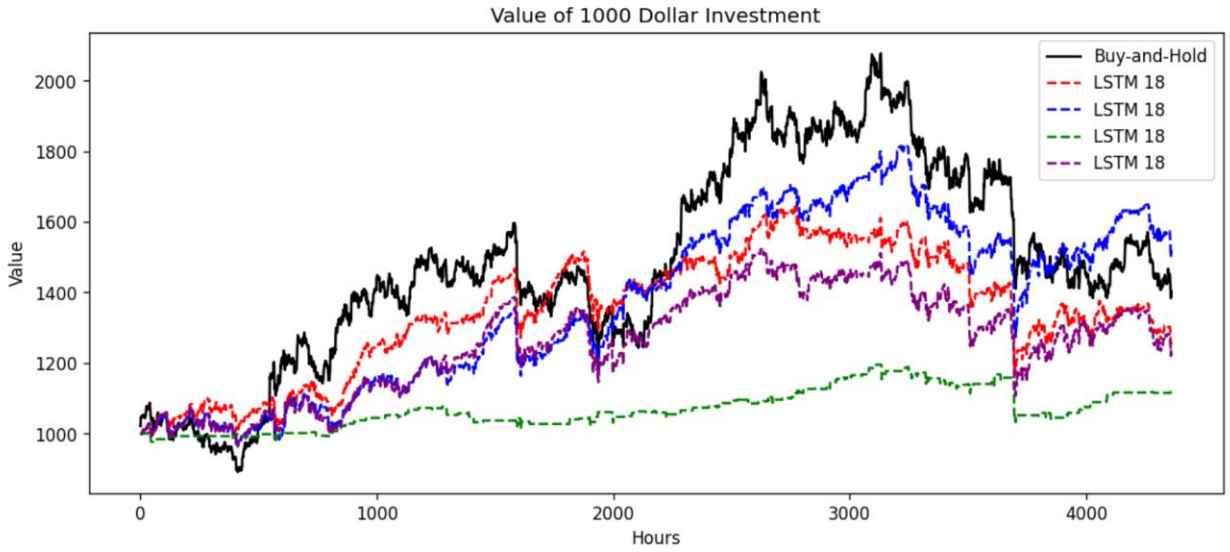
The performance of our LSTM agents with 12-hour sequences is:

```
The final value with Buy-and-Hold is 1402.49
Our accuracies and final values for these models are:
 52.0 percent and 1043.84 dollars,
 52.0 percent and 1049.14 dollars,
 53.0 percent and 1190.84 dollars,
 52.0 percent and 947.98 dollars.
```



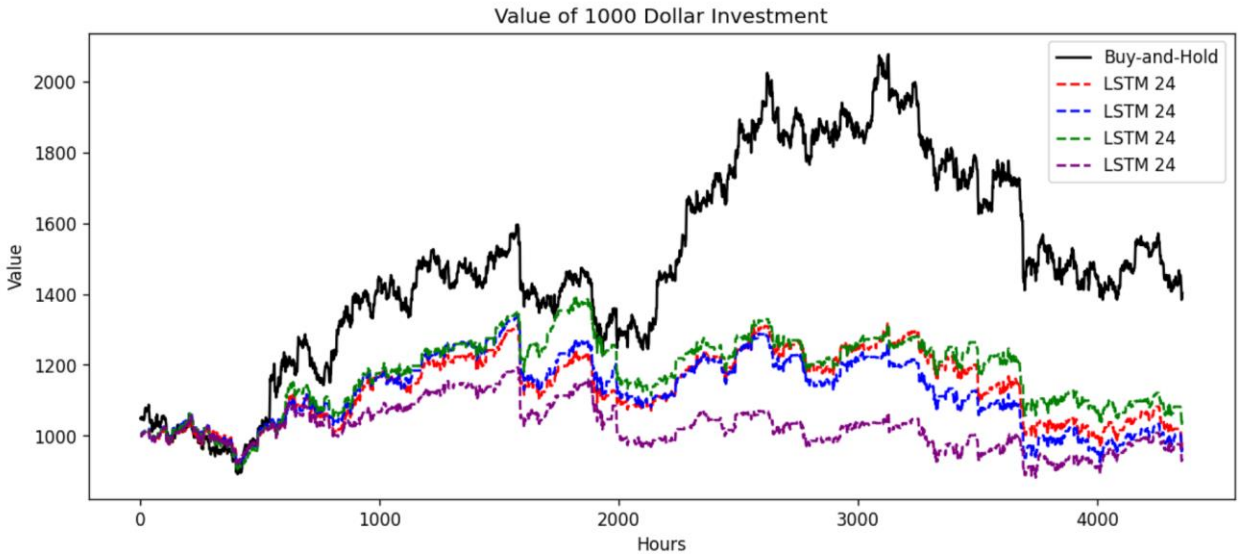
For our 18-hour sequences, we have:

```
The final value with Buy-and-Hold is 1402.49
Our accuracies and final values for these models are:
 54.0 percent and 1269.94 dollars,
 54.0 percent and 1519.76 dollars,
 51.0 percent and 1119.54 dollars,
 54.0 percent and 1236.74 dollars.
```



Finally, for our 24-hour sequences, we have:

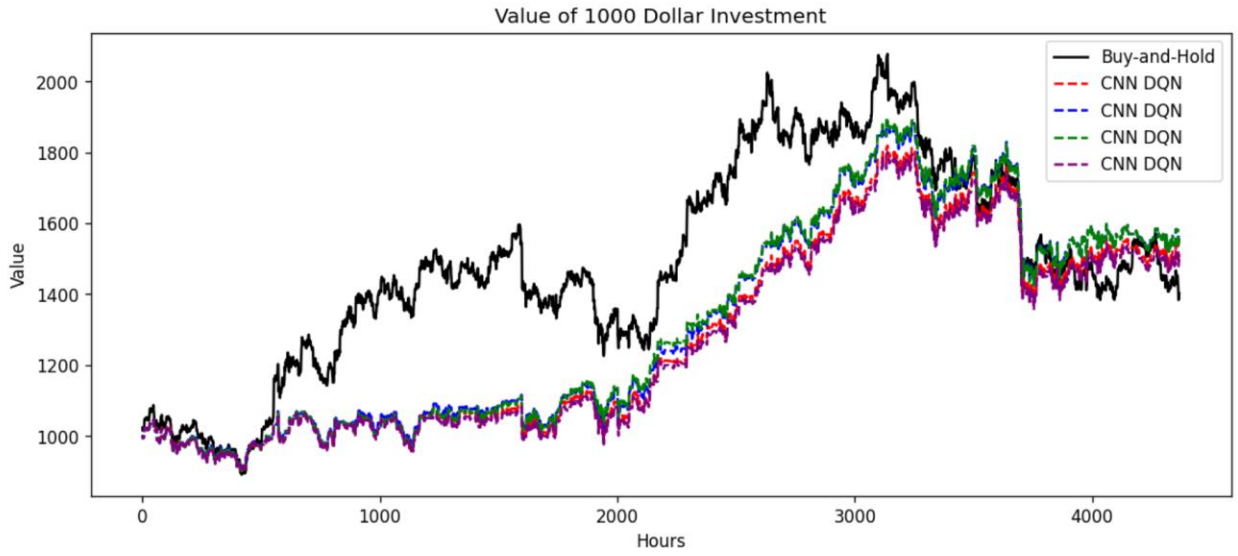
```
The final value with Buy-and-Hold is 1402.49
Our accuracies and final values for these models are:
 53.0 percent and 982.91 dollars,
 52.0 percent and 968.54 dollars,
 53.0 percent and 1043.2 dollars,
 52.0 percent and 940.81 dollars.
```



J. Convolutional Neural Network-Based Dueling Double Q-Network with Prioritized Experience Replay

With $\gamma = 0.25$ and 12-hour "images," on the same testing data as with the 12-hour CNN and LSTM, we have the following performance of our DQN agents, each trained on the same architecture:

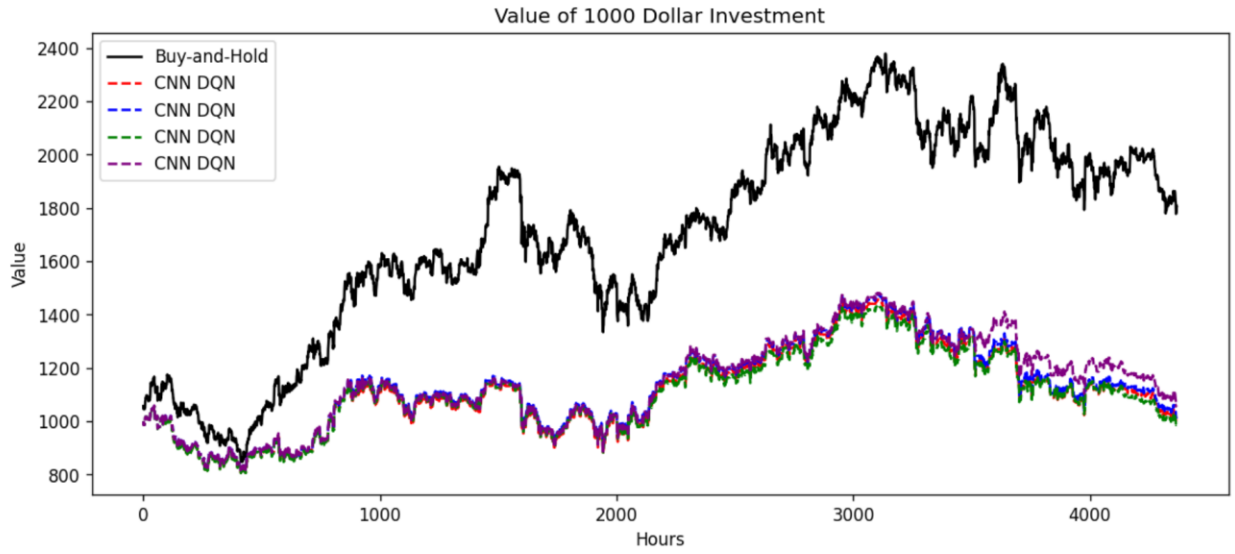
```
The final value with Buy-and-Hold is 1402.49
Our final values for these models are:
 1514.08 dollars,
 1557.84 dollars,
 1556.13 dollars,
 1499.35 dollars.
```



Given that this is our best performing algorithm thus far, we will use the same trained agents with *Ethereum* (ETH) and *Litecoin* (LTC) data. Note that we are using the *exact* same agents, those trained on BTC, to see if their performance generalizes to other coins or if one would need to train agents on those coins' data to achieve similar performance.

On ETH on the same time frame, we have:

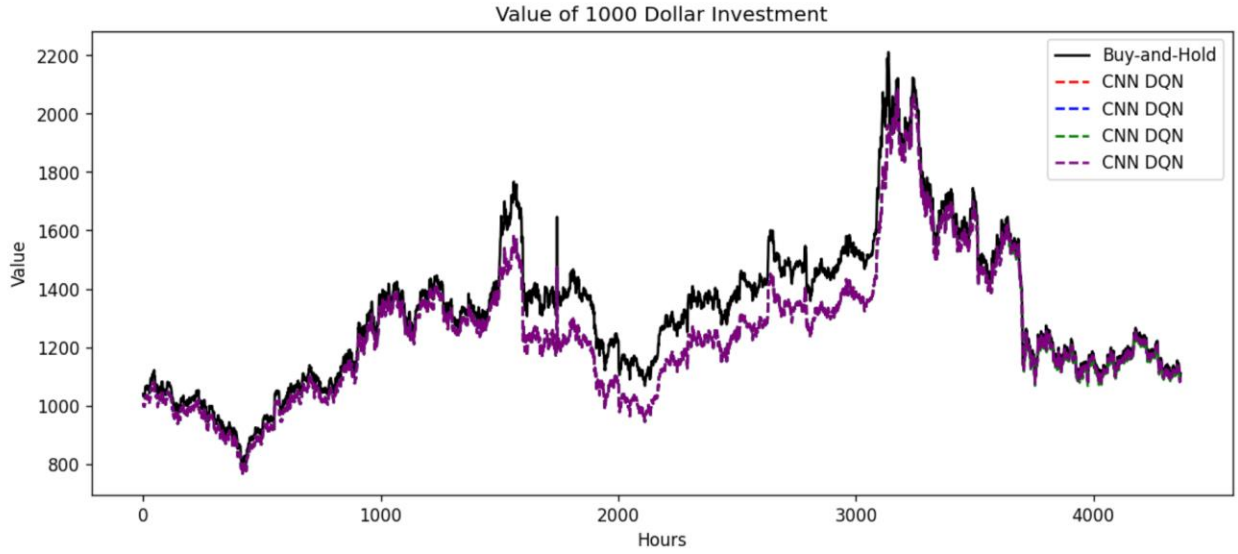
```
The final value with Buy-and-Hold is 1806.5
Our final values for these models are:
 1002.52 dollars,
 1017.8 dollars,
 983.05 dollars,
 1061.91 dollars.
```



Unfortunately, its performance is quite poor. Perhaps training it with ETH data would be better.

With LTC, we have:

```
The final value with Buy-and-Hold is 1111.26
Our final values for these models are:
 1098.36 dollars,
 1098.36 dollars,
 1092.43 dollars,
 1098.36 dollars.
```

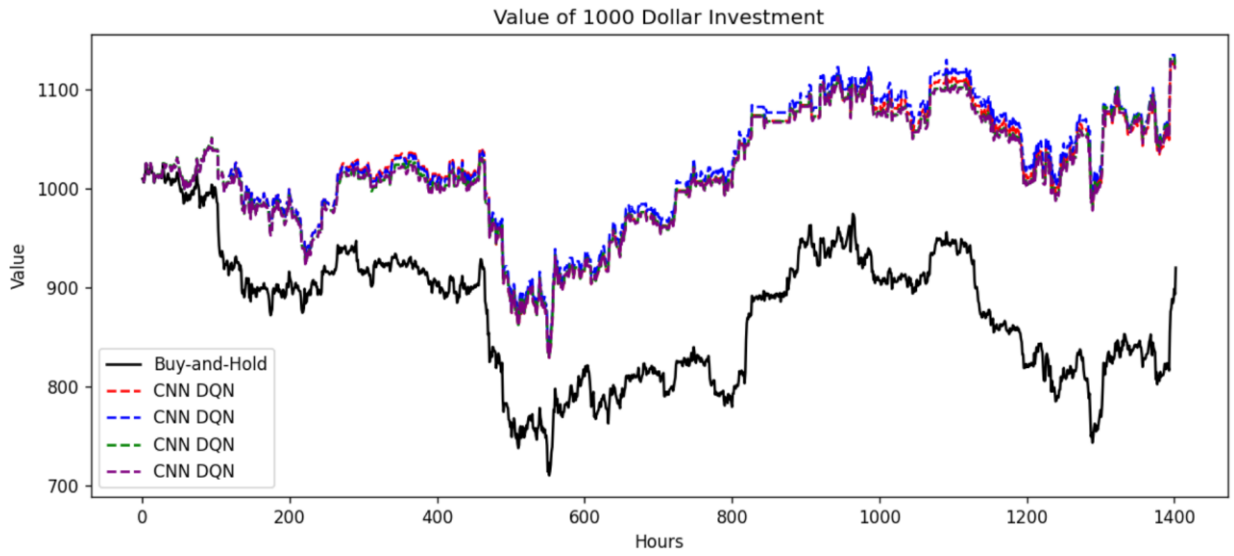


This is interesting, as we see very little divergence in the four agents' paths, but we also see that all of them struggle particularly in the middle, around the 1500-hour mark, and then catch back up to Buy-and-Hold.

We can also see how this agent would perform in more recent data, as the cryptocurrency markets have lost large amounts of value in early 2022. After "image" creation, our testing environment is now from January 1st of 2022 at 11:00 AM to February 28th of 2022 at 9:00 PM.

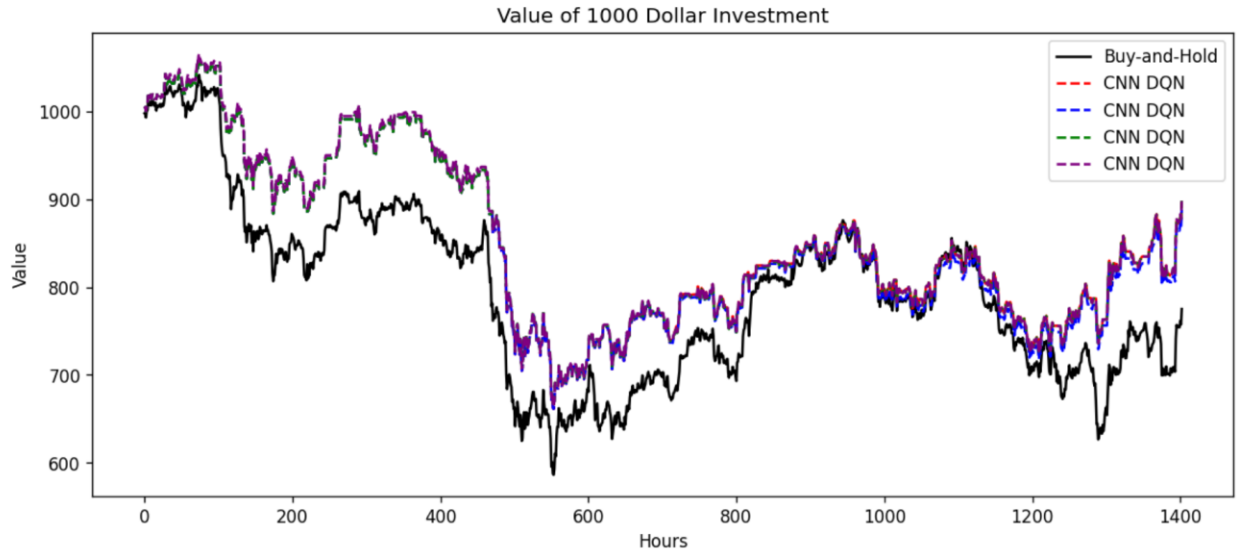
For BTC, we have:

```
The final value with Buy-and-Hold is 919.76
Our final values for these models are:
 1118.95 dollars,
 1127.83 dollars,
 1125.43 dollars,
 1121.38 dollars.
```



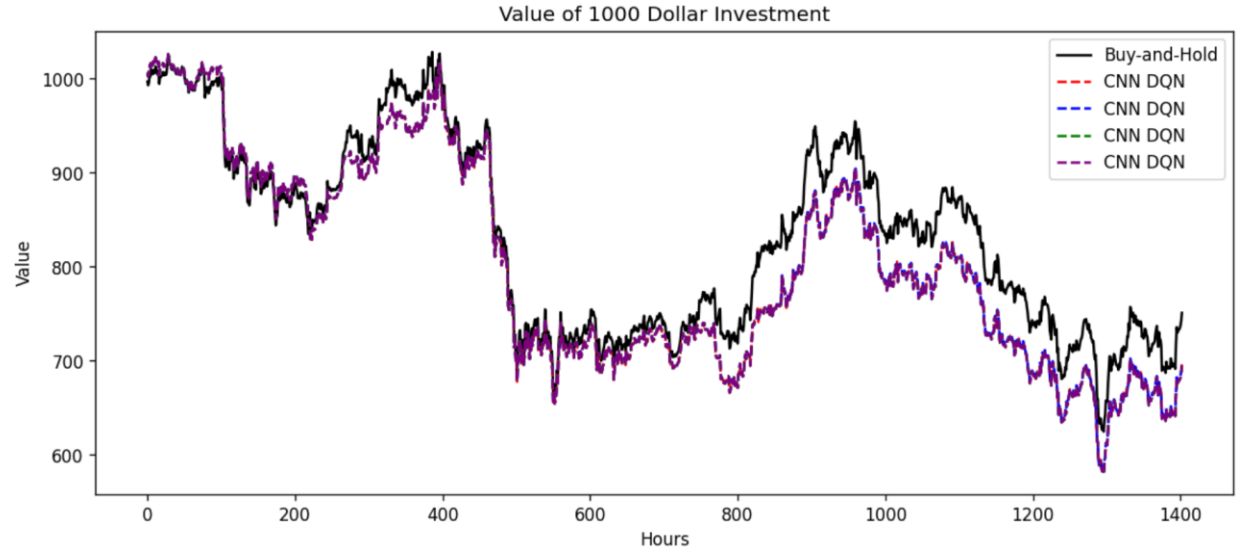
For ETH, we have:

```
The final value with Buy-and-Hold is 775.04
Our final values for these models are:
 896.6 dollars,
 888.17 dollars,
 898.3 dollars,
 898.35 dollars.
```



For LTC, we have:

```
The final value with Buy-and-Hold is 750.87
Our final values for these models are:
 696.17 dollars,
 696.45 dollars,
 694.98 dollars,
 694.98 dollars.
```

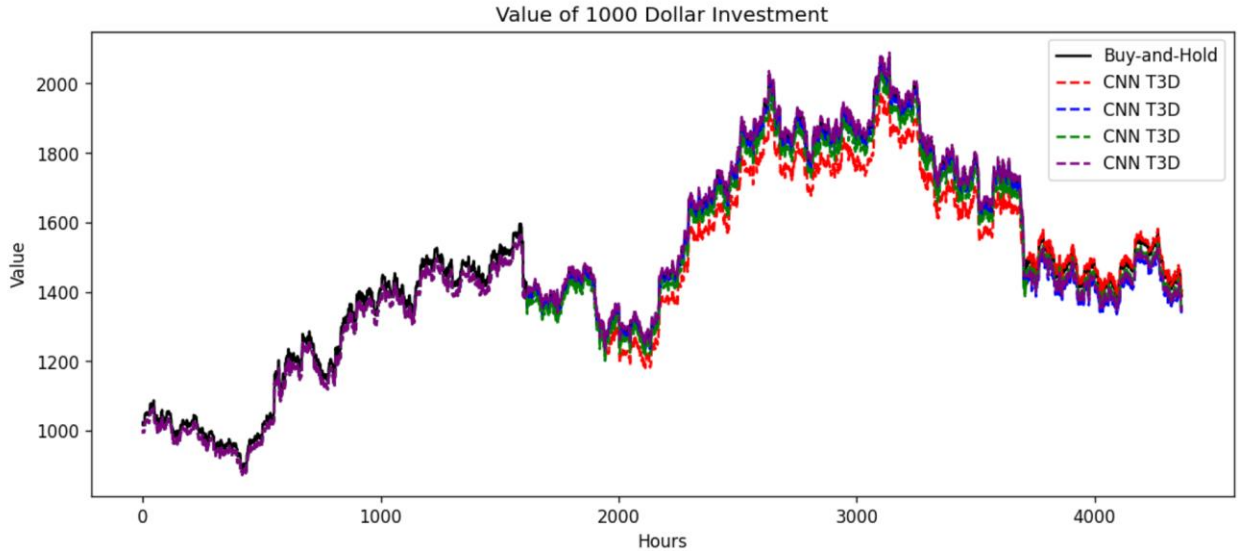


These performances are a bit of a surprise, as BTC and ETH did far better than LTC, the opposite to the pre-2022 data.

K. Convolutional Neural Network-Based Twin Delayed Deep Deterministic Policy Gradient

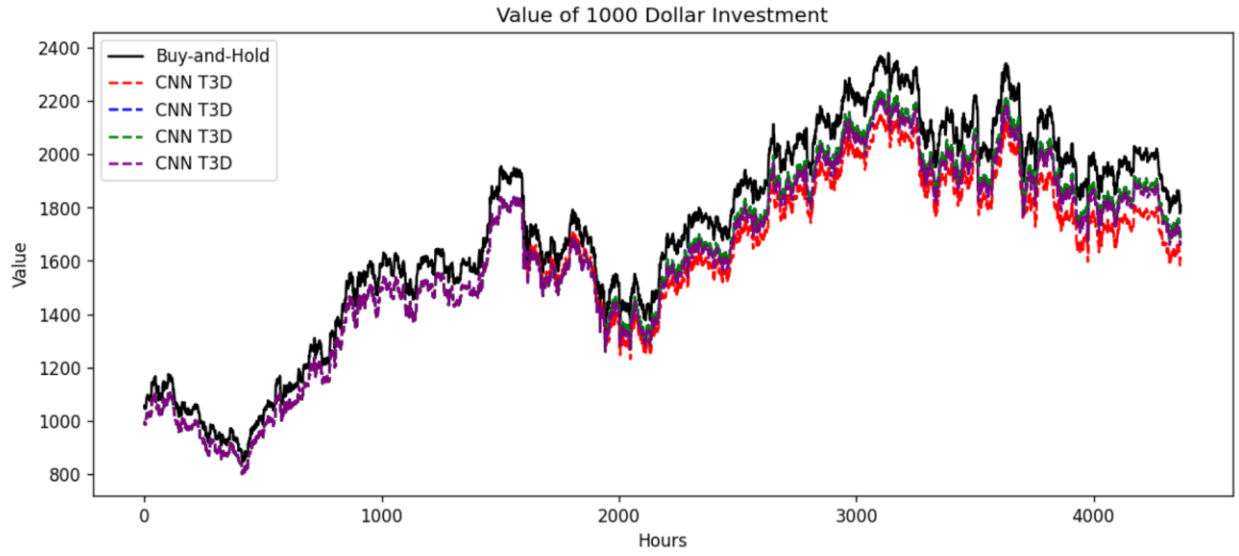
With $y = 1$ and 12-hour "images," we have the following performance of our T3D agents, each trained on the same architecture:

```
The final value with Buy-and-Hold is 1402.49
Our final values for these models are:
 1411.88 dollars,
 1352.92 dollars,
 1374.09 dollars,
 1364.54 dollars.
```



We now run the *same* agents trained on BTC data on ETH and LTC. For ETH, we have:

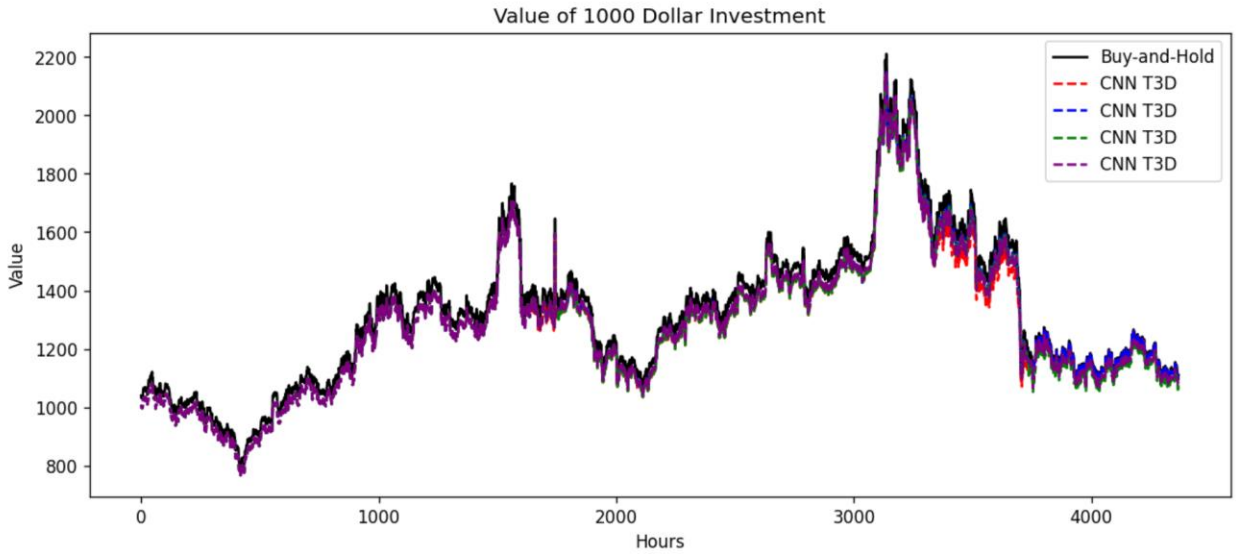
```
The final value with Buy-and-Hold is 1806.5
Our final values for these models are:
 1608.979 dollars,
 1705.4036 dollars,
 1705.4026 dollars,
 1684.6353 dollars.
```



We see that the CNN-based T3D agents also follow closely the shape of Buy-and-Hold, though do not perform quite so well. That is to be expected, as it hasn't been fine-tunes as much as the BTC agent.

Performing the same thing on LTC, we see the following:

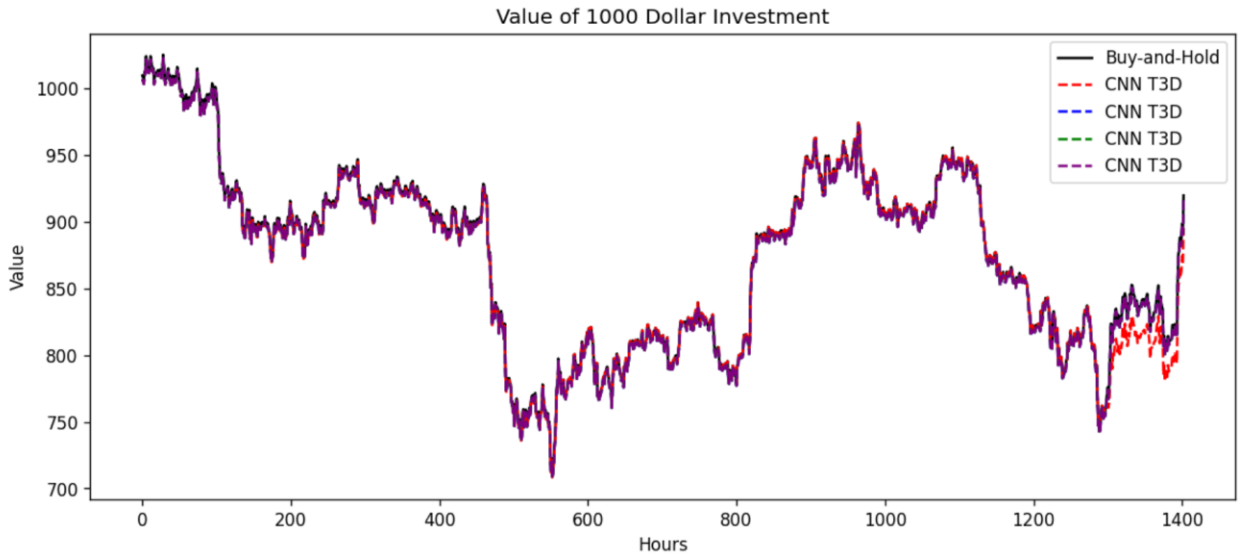
```
The final value with Buy-and-Hold is 1111.26
Our final values for these models are:
 1105.4114 dollars,
 1105.7826 dollars,
 1076.486 dollars,
 1088.9675 dollars.
```



We see similar behavior. The agents stick close to the Buy-and-Hold, though they do not surpass it by the end. Given that we used the hyperparameters for BTC, this isn't a surprise, but still indicates potential with this algorithm.

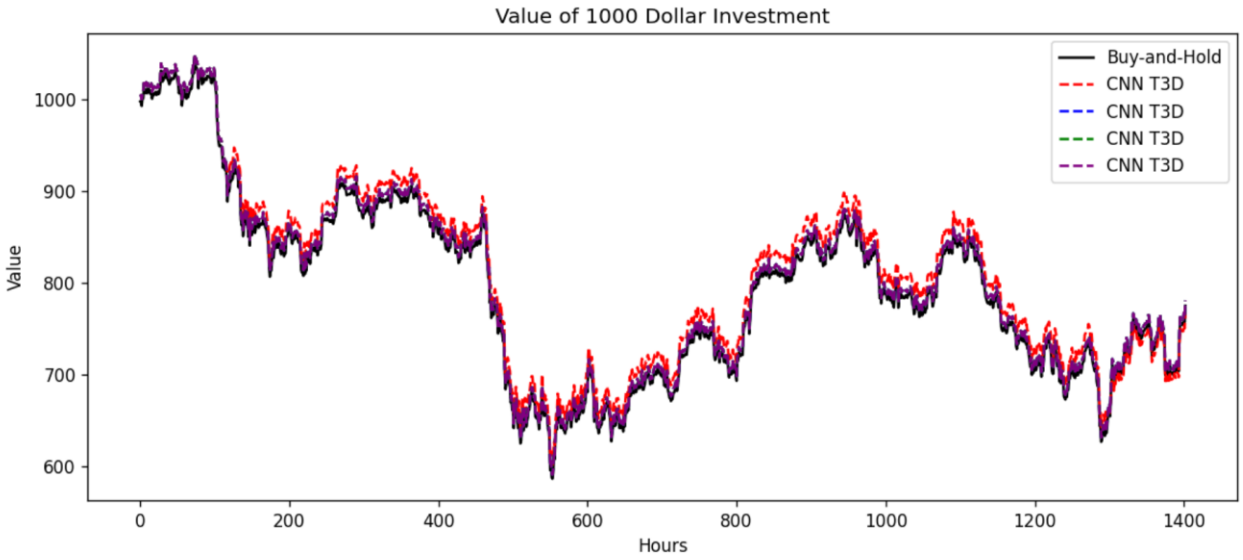
Finally, we can also ask about their performance with more recent data. For BTC, we have:

```
The final value with Buy-and-Hold is 919.76
Our final values for these models are:
 894.9804 dollars,
 917.4617 dollars,
 917.46423 dollars,
 917.4474 dollars.
```



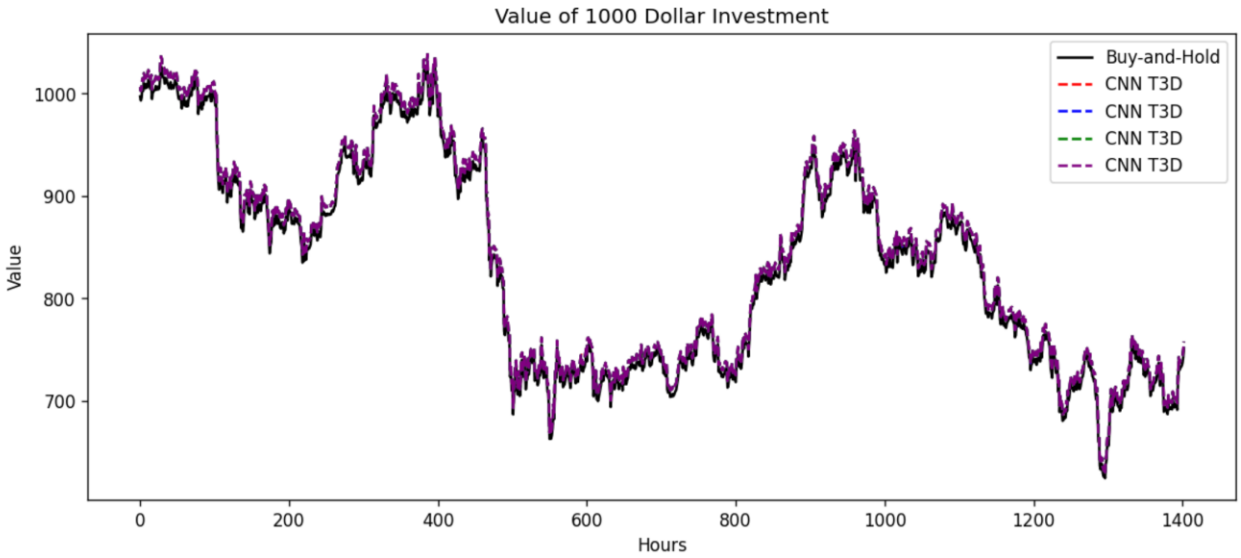
For ETH, we have:

```
The final value with Buy-and-Hold is 775.04
Our final values for these models are:
 767.8205 dollars,
 780.9074 dollars,
 780.91 dollars,
 780.9003 dollars.
```



For LTC, we have:

```
The final value with Buy-and-Hold is 750.87
Our final values for these models are:
 757.9406 dollars,
 757.9418 dollars,
 757.9452 dollars,
 757.94336 dollars.
```



We see similar behavior from all three coins; the agents remain close to the Buy-and-Hold, neither gaining nor losing much value.

We should also mention the strong resemblance between the shapes of the coins' values, with LTC being slightly more divergent. Given that BTC and ETH are the number one and two market-valued assets, their strikingly similar behavior is not a surprise.

8. Discussion of Investing Results, Moving Forward, and Conclusion

A. Discussion of Investing Results

From the *Investing Results*, several things are worth noting. First, none of the agents trading on the three common technical indicators did well (though, we admit that the MACD and Aroon Oscillator did a decent job of not losing money, with one of the Aroon Oscillator-agents doing very well overall). Second, none of the Classical Statistical algorithms fared much better. Speculating on the possible reasons, we argue that the ETS is too simple an algorithm, best suited for time series with regular seasonal trends, and the SARIMAX algorithm, while significantly more complex, is still a sum of linear terms.

We see similar performance from the XGBoost and the MLP agents, both scoring a few percentage points about 50%. Perhaps if they had more information, they might improve that.

Moving to the CNN and LSTM, we see that with the 12-hour "images" and 18-hour sequences, we have a marked improvement. This could be due to having more information, the structure of the algorithms themselves, or a combination. However, and much to our surprise, increasing the information to 24-hour "images" and sequences *dramatically decreased* their performance. Given that the seasonal decomposition performed in preparation for fitting an ETS and SARIMAX model showed 24-hour seasonality, we expected a full day of information, a full season, to provide a greater advantage. Why did it not? Perhaps it was *too much information*. Would increasing the complexity of the neural nets have helped? Though our grid search had a wide range of layers and filters/units possible, there is still more we could have done.

Finishing with our RL agents, we found that, on average, the augmented CNN-based DQN with 12-hour "images" did the best. However, it is important to recognize that we had a cut-off point from which to judge its performance. Throughout the testing environment, we see the augmented CNN-based DQN agents were consistently *below* the Buy-and-Hold agent. So, had this been a real-life implementation, at any given time when we checked our portfolio, we would see that we *are not doing as well as we could have been had we just used Buy-and-Hold*. With future information, we know to hold out and that our algorithm would eventually surpass (on average) Buy-and-Hold, but without knowing this, it does appear as if, at any given moment, the CNN-based-DQN agent is not performing well.

When we switched to the most recent data, wherein the market had significant losses, we see the CNN-based DQN agents for BTC and ETH *stay above* Buy-and-Hold most of the time, which is a positive. We can make a hypothesis that this algorithm is less capable of predicting large increases in price. This would explain why it generally misses the jumps in price, resulting in lower overall performance when the price is increasing. However, it does seem more capable, at least with BTC and ETH, of leaving the market when large decreases are happening, preserving the portfolio as the value of BTC falls. Further testing would be needed to support this hypothesis.

Finally, in our opinion, the best overall algorithm is the CNN-based T3D. There are three main reasons. The first is that each training produced a very similarly behaving agent independent of

whether the price was increasing or decreasing. We see that with the ANNs and CNN-based DQN, retraining with the same hyperparameters and seeing the result produces, sometimes markedly, different trajectories. Though some are tighter (particularly the CNN-based DQN in both LTC environments), that divergence is worrying.

The second reason is that *none* of the other algorithms stayed very close to Buy-and-Hold, except for the CNN-based DQN in the LTC environments (and even then, the CNN-based T3D agent was slightly closer). Some did better in the beginning of the testing environment, but once the large price jumps happened, none could keep pace. However, the CNN-based T3D agent *could*. We attribute this to its ability to make smaller changes in the portfolio, as opposed to all-in or all-out. Though it did not perform so well as the price decreased, staying close to Buy-and-Hold as the portfolio lost value, there are ways that we could incorporate strong signals to completely leave the market until it is more favorable, combining an AI agent with other indicators to bolster its performance. One of which is mentioned briefly in the next section.

Third, as we noted when we introduced and implemented the CNN-based DQN and CNN-based T3D algorithms, these have a very large number of hyperparameters. We did very little searching, mostly just the γ value for both. So, to see the CNN-based T3D remain close to the Buy-and-Hold values the entire six months of BTC testing, surpass them on occasion, and have consistent results across trainings *with very little optimization* hints to us the power of this algorithm. If the purpose of this project was to test the capabilities of several different architectures for algorithmic cryptocurrency trading, we feel confident that these results point toward the T3D algorithm and others like it, algorithms capable of acting in a continuous action space, having the highest chance of positive results.

B. Moving Forward

Though we find that, after approximately six months of trading BTC, our CNN-based T3D algorithm does the best, we cannot simply stop here. There are several factors that need to be addressed should one decide to move forward with an implementation of this algorithm as a trading agent. First, and probably the most arbitrary, is the choice of TA used. Though we tried to get a nice mix of some popular ones, as well as one from each general category, there is some randomness to this. One could retrain with completely different sets of TA on different time scales and might find better or worse performance. Even with the three types we chose, one of them, the Aroon Oscillator with a 10-hour timeframe, *far* outperformed the others. Perhaps running through different TA on different timeframes and choosing the best of those to give to our agents would fare better.

Beyond just testing different TA, one could move to different time slices of asset data. Hourly seemed to have enough patterns for our models to learn, whereas, when this project originally began with daily information, we struggled under a lack of sufficient data. However, one might decide to move to lower time slices, intra-hour, minute, or even lower.

What about outside information? Besides just TA, there exists another type of analysis, *Fundamental Analysis*, which can be harder to quantify but might be more important. How is a company *doing*, not just their stock, but all the other factors going into their business? Do they

have a strong CEO? Have their quarterly reports been favorable? Do they have a product on the horizon that might give them an edge? These can be more difficult to put into numerical terms, but many have tried, and incorporating this data into our models might provide better predictive power.

Further, what about the *public perception of an asset*? Looking at, for example, *Dogecoin* (DOGE), Elon Musk has been known to use social media to express his feeling about the coin. Could one show a correlation between his vocalization (for or against) and DOGE price changes? If so, then perhaps simply having a *Natural Language Processing* (NLP) agent monitoring his social media for the moment he uses it and processing his posts would provide enough of a Buy or Sell indicator. Moving beyond one person, could one use an NLP agent to read several websites and social media posts at once, creating a "score" of sorts, a measure of public sentiment, at any given time for an asset and incorporating that into our models?

There is another indictor, the *Cboe Volatility Index*, that can be used to gauge the volatility of the market, perhaps just as a good signal to leave when the market gets too volatile, but as also yet another datum to give to our models.

We chose indicators that would be recognizable by those with a passing familiarity with trading, but also whose nature yielded easily to both processing and quantification. In doing so, we dramatically limited the information our models might use.

There are also many arbitrary choices made in the models themselves. The number of hyperparameters for neural networks and neural-network-based models are incredible, and there is always a chance that we missed on an optimization, or perhaps even a whole range of hyperparameters, that would have led to better performance. This is regretful, and more computing power and time might yield larger searches and more profitable models.

Though we tried to get a nice mix of classical, neural network, and reinforcement learning models, there are some that we did not explore. LSTMs come in many more varieties, such as the popular *Long-Short Term Memory with Attention* model. We mentioned several alternative RL algorithms such as A2C and TRPO, both used in the paper *Deep Reinforcement Learning for Automated Stock Trading: An Ensemble Strategy*, but we did not implement them.

In short, the space of algorithm types, data to feed to them, and unique instantiations of each algorithm is vast, and we only explored the smallest portion in this work.

C. Conclusion

To summarize our approach, we gathered enough technical indicators for each hour of Bitcoin data to have, adding the High, Low, Close, and Volume, 101 pieces of information, or features. We pared this down to a mere 20 with PCA. Then, we trained each algorithm with data from roughly, depending on the specific algorithm, July 3rd of 2018 at 3:00 PM to July 2nd of 2021 at 10:00 AM. The algorithms include the following: ETS, SARIMAX, XGBoost, MLP, CNN, LSTM, CNN-based augmented DQN, and CNN-based T3D. We compared their performance as trading agents in the Bitcoin market on data from roughly, depending the on the specific algorithm, July 2nd of 2021 at 11:00 AM until December 31st of 2021 at 10:00 PM. We also

compared their performance to trading with three technical indicators, RSI, MACD, and Aroon Oscillator (over several time iterations each), a random agent, and the simplest investment strategy, Buy-and-Hold. At the end, we tested how the CNN-based DQN and CNN-based T3D agents' performance would respond to newer data from the beginning of 2022 and whether they could generalize their performance on two other coins, Ethereum and Litecoin. Reviewing all the agents' performance, we find that, thought our CNN-based DQN agents had the highest, on average, portfolio value at the end of the testing period in 2021, we determined that the CNN-based T3D algorithm, and others of a similar structure, likely have the highest potential due to their overall positive value compared to Buy-and-Hold, as well as their ability to keep pace with the large price changes other algorithms failed to capture.

We originally conceived of this project as a survey of classical and modern forecasting/prediction methods with the goal of parsing through the daunting collection of algorithms available to be used as the foundation of a cryptocurrency trading agent. We believe that we have shown that though this problem is quite difficult, there are promising avenues to pursue for someone interested in algorithmic cryptocurrency trading.

9. Bibliography, Software/Packages, and GitHub

A. Bibliography

This bibliography was generated in MLA style from https://www.mybib.com/tools/mla-citation-generator with slight modification, especially for the online book *Forecasting: Principles and Practice (3rd ed)*. We also note that we do not have dates when we accessed the webpages, as those were not recorded.

1. **Abstract**
2. **Introduction**

1. *Nakamoto, Satoshi. "Bitcoin: A Peer-To-Peer Electronic Cash System." 31 Oct. 2008. Business Insider*, Jan 16th, 2021.
2. Hernandez, Joe. "El Salvador Just Became the First Country to Accept Bitcoin as Legal Tender." *NPR.org*, 7 Sept. 2021, www.npr.org/2021/09/07/1034838909/bitcoin-el-salvador-legal-tender-official-currency-cryptocurrency.
3. "China Makes Cryptocurrency Illegal." *NPR.org*, www.npr.org/2021/09/25/1040669103/china-makes-cryptocurrency-illegal.
4. "CNBC Transcript: Billionaire Investors Warren Buffett & Charlie Munger Sit down with CNBC's Becky Quick for CNBC's 'Buffett & Munger: A Wealth of Wisdom.'" *CNBC*, 30 June 2021, www.cnbc.com/2021/06/29/cnbc-transcript-billionaire-investors-warren-buffett-charlie-munger-sit-down-with-cnbcs-becky-quick-for-cnbcs-buffett-munger-a-wealth-of-wisdom-.html.
5. GmbH. "These 13 Banks Have Invested the Most in Crypto and Blockchain to Date." *Markets.businessinsider.com*, markets.businessinsider.com/news/currencies/13-top-banks-investing-cryptocurrency-blockchain-technology-funding-blockdata-bitcoin-2021-8.
6. Staff of the U.S. Securities and Exchange Commission. *Staff Report on Algorithmic Trading in U.S. Capital Markets*. U.S. Securities and Exchange Commission, 5 Aug. 2020.

3. **Background and Literature Review**

1. Sezer, Omer Berat, and Ahmet Murat Ozbayoglu. "Algorithmic Financial Trading with Deep Convolutional Neural Networks: Time Series to Image Conversion Approach." *Applied Soft Computing*, vol. 70, Sept. 2018, pp. 525–538, www.sciencedirect.com/science/article/pii/S1568494618302151, 10.1016/j.asoc.2018.04.024.
2. Lu, Wenjie, et al. "A CNN-LSTM-Based Model to Forecast Stock Prices." *Complexity*, vol. 2020, 23 Nov. 2020, pp. 1–10, 10.1155/2020/6622927. Accessed 16 Aug. 2021.
3. Yang, Hongyang, et al. "Deep Reinforcement Learning for Automated Stock Trading: An Ensemble Strategy." *SSRN Electronic Journal*, 2020, 10.2139/ssrn.3690996. Accessed 16 Nov. 2020.
4. Tidor-Vlad Pricope, 2021. "Deep Reinforcement Learning in Quantitative Algorithmic Trading: A Review," Papers 2106.00123, arXiv.org. Recommended citation from ideas.repec.org.
5. Jia, Zhichao, et al. "LSTM-DDPG for Trading with Variable Positions." *Sensors*, vol. 21, no. 19, 30 Sept. 2021, p. 6571, 10.3390/s21196571. Accessed 19 Feb. 2022.

4. Primers

a. Primer on Classical Statistical Forecasting and XGBoost

1. Hyndman, R. J., & Athanasopoulos, G. (2021, May). *Forecasting: Principles and Practice (3rd ed)*. 8.1 Simple exponential smoothing. Retrieved from https://otexts.com/fpp3/AR.html.
2. Hyndman, R. J., & Athanasopoulos, G. (2021, May). *Forecasting: Principles and Practice (3rd ed)*. 8.2 Methods with trend. Retrieved from https://otexts.com/fpp3/AR.html.
3. Hyndman, R. J., & Athanasopoulos, G. (2021, May). *Forecasting: Principles and Practice (3rd ed)*. 8.3 Methods with seasonality. Retrieved from https://otexts.com/fpp3/AR.html.
4. Hyndman, R. J., & Athanasopoulos, G. (2021, May). *Forecasting: Principles and Practice (3rd ed)*. 8.5 Innovations state space models for exponential smoothing. Retrieved from https://otexts.com/fpp3/AR.html.
5. Hyndman, R. J., & Athanasopoulos, G. (2021, May). *Forecasting: Principles and Practice (3rd ed)*. 9.3 Autoregressive models. Retrieved from https://otexts.com/fpp3/AR.html.
6. Sosna, Matt. "A Deep Dive on ARIMA Models - towards Data Science." *Medium*, Towards Data Science, 12 Aug. 2021, towardsdatascience.com/a-deep-dive-on-arima-models-8900c199ccf.
7. Hyndman, R. J., & Athanasopoulos, G. (2021, May). *Forecasting: Principles and Practice (3rd ed)*. 9.2 Backshift notation. Retrieved from https://otexts.com/fpp3/AR.html.
8. Wikipedia Contributors. "Autoregressive Integrated Moving Average." *Wikipedia*, Wikimedia Foundation, 17 Apr. 2019, en.wikipedia.org/wiki/Autoregressive_integrated_moving_average.
9. "From AR to SARIMAX: Mathematical Definitions of Time Series Models." *Phosgene89.Github.io*, phosgene89.github.io/sarima.html.
10. Hyndman, R. J., & Athanasopoulos, G. (2021, May). *Forecasting: Principles and Practice (3rd ed)*. 9.4 Moving average models. Retrieved from https://otexts.com/fpp3/AR.html.
11. Hyndman, R. J., & Athanasopoulos, G. (2021, May). *Forecasting: Principles and Practice (3rd ed)*. 9.1 Stationarity and differencing. Retrieved from https://otexts.com/fpp3/AR.html.
12. "Statsmodels.tsa.statespace.sarimax.SARIMAX — Statsmodels." *Www.statsmodels.org*, www.statsmodels.org/dev/generated/statsmodels.tsa.statespace.sarimax.SARIMAX.html.
13. "Xgboost/Demo at Master · Dmlc/Xgboost." *GitHub*, github.com/dmlc/xgboost/tree/master/demo#machine-learning-challenge-winning-solutions.
14. "Introduction to Boosted Trees — Xgboost 1.6.0-Dev Documentation." *Xgboost.readthedocs.io*, xgboost.readthedocs.io/en/latest/tutorials/model.html.
15. dmlc. "Dmlc/Xgboost." *GitHub*, 25 Oct. 2019, github.com/dmlc/xgboost.
16. "Classification and Regression Trees (CART)-Classifier." *Www.geo.fu-Berlin.de*, 27 Aug. 2021, www.geo.fu-berlin.de/en/v/geo-it/gee/3-classification/3-1-methodical-background/3-1-1-cart/index.html.
17. Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 785–794). New York, NY, USA: ACM. https://doi.org/10.1145/2939672.2939785. APA Style

18. Tseng, Gabriel. “Gradient Boosting and XGBoost.” *Gabriel Tseng*, 25 Feb. 2018, gabrieltseng.github.io/posts/2018-02-25-XGB/.

b. Primer on Artificial Neural Networks

1. SHARMA, SAGAR. “Activation Functions in Neural Networks.” *Medium*, Towards Data Science, 6 Sept. 2017, towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.
2. Ng, A, Katanforoosh, K., & Mourri, Y.B. *Neural Networks and Deep Learning* [MOOC]. Coursera. https://www.coursera.org/learn/neural-networks-deep-learning?specialization=deep-learning. APA Style
3. Dixon, M. F., Halperin, I., & Bilokon, P. (2020). *Machine Learning in Finance: From Theory to Practice.* Chapter 4: Feedforward Neural Networks. Section 2: Feedforward Architectures, Equations (4.1) and (4.2), (p. 113).
4. Dixon, M. F., Halperin, I., & Bilokon, P. (2020). *Machine Learning in Finance: From Theory to Practice*. Chapter 4: Feedforward Neural Networks. Section 5: Stochastic Gradient Descent, (p. 142-143).
5. Kathuria, Ayoosh. “Intro to Optimization in Deep Learning: Gradient Descent.” *Paperspace Blog*, Paperspace Blog, 2 June 2018, blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/

c. Primer on Reinforcement Learning

1. Dixon, M. F., Halperin, I., & Bilokon, P. A. (2020). *Machine Learning in Finance: From Theory to Practice* Chapter 9: Reinforcement Learning, Section 2: Elements of Reinforcement Learning, Sub-Section 2.1: Rewards, (p. 284).
2. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 2: Markov Decision Process, Section: Definition of Reinforcement Learning, (p. 20).
3. Dixon, M. F., Halperin, I., & Bilokon, P. A. (2020). *Machine Learning in Finance: From Theory to Practice* Chapter 9: Reinforcement Learning, Section 2: Elements of Reinforcement Learning, Sub-Section 2.2: Value and Policy Functions, (p. 286).
4. Dixon, M. F., Halperin, I., & Bilokon, P. A. (2020). *Machine Learning in Finance: From Theory to Practice*. Chapter 9: Reinforcement Learning, Section 2: Elements of Reinforcement Learning, Sub-Section 2.3: Observable Versus Partially Observable Environments, Equation (9.4), (p. 287).
5. Dixon, M. F., Halperin, I., & Bilokon, P. A. (2020). *Machine Learning in Finance: From Theory to Practice* Chapter 9: Reinforcement Learning, Section 3: Markov Decision Processes, (p. 289).
6. Dixon, M. F., Halperin, I., & Bilokon, P. A. (2020). *Machine Learning in Finance: From Theory to Practice*. Chapter 9: Reinforcement Learning, Section 3: Markov Decision Processes, Equations (9.9), (p. 290).
7. Dixon, M. F., Halperin, I., & Bilokon, P. A. (2020). *Machine Learning in Finance: From Theory to Practice*. Chapter 9: Reinforcement Learning, Section 3: Markov Decision Processes, Sub-Section 3.2: Value Functions and Bellman Equations, Equations (9.10), (9.11), (9.12), and (9.13), (p. 294).

8. Dixon, M. F., Halperin, I., & Bilokon, P. A. (2020). *Machine Learning in Finance: From Theory to Practice.* Chapter 9: Reinforcement Learning, Section 3: Markov Decision Processes, Sub-Section 3.2: Value Functions and Bellman Equations, Equation (9.10), (p. 293).
9. Roy, Baijayanta. "All about Backup Diagram." *Medium*, 23 Feb. 2020, towardsdatascience.com/all-about-backup-diagram-fefb25aaf804.
10. Dixon, M. F., Halperin, I., & Bilokon, P. A. (2020). *Machine Learning in Finance: From Theory to Practice*. Chapter 9: Reinforcement Learning, Section 3: Markov Decision Processes, Sub-Section 3.2: Value Functions and Bellman Equations, Equations (9.14) and (9.15), (p. 295).
11. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym*. Chapter 2: Markov Decision Process, Section: Bellman Equations, Equation (2.16), (p. 41).
12. Dixon, M. F., Halperin, I., & Bilokon, P. A. (2020). *Machine Learning in Finance: From Theory to Practice.* Chapter 9: Reinforcement Learning, Section 3: Markov Decision Processes, Sub-Section 3.3: Optimal Policy and Bellman Optimality, Equations (9.16), (9.17), (9.19), and (9.20), (p. 297).
13. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 2: Markov Decision Process, Section: Optimality Bellman Equations, Equations (2.22) and (2.23), (p. 45).
14. Dixon, M. F., Halperin, I., & Bilokon, P. A. (2020). *Machine Learning in Finance: From Theory to Practice.* Chapter 9: Reinforcement Learning, Section 3: Markov Decision Processes, Sub-Section 3.3: Optimal Policy and Bellman Optimality, Equation (9.21), (p. 298).
15. Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction (2nd ed).* Chapter 6: Temporal Difference Learning, (p. 119).
16. "Part 2: Kinds of RL Algorithms — Spinning up Documentation." *Openai.com*, 2018, spinningup.openai.com/en/latest/spinningup/rl_intro2.html.
17. Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction (2nd ed).* Chapter 4: Dynamic Programming, Section 4.1: Policy Evaluation (Prediction), Equation (4.5), (p. 74).
18. Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction (2nd ed).* Chapter 4: Dynamic Programming, Section 4.2: Policy Improvement, Equation (4.7), (p. 78).
19. Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction (2nd ed).* Chapter 4: Dynamic Programming, Section 4.1: Policy Evaluation (Prediction), Equation (4.9), (p. 79).
20. Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction (2nd ed)* Chapter 4: Dynamic Programming, Section 4.3: Policy iteration, (p. 80).
21. Dixon, M. F., Halperin, I., & Bilokon, P. A. (2020). *Machine Learning in Finance: From Theory to Practice.* Chapter 9: Reinforcement Learning, Section 5: Reinforcement Learning Methods, Sub-Section 5.1: Monte Carlo Methods, (p. 307).
22. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 4: Model-Free Approaches, Section: Control with Monte Carlo, Equation (4.3), (p. 86).

23. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym* Chapter 4: Model-Free Approaches, Section: Temporal Difference Learning Methods, Equation (4.4), (p. 93).
24. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 4: Model-Free Approaches, Section: Control with Monte Carlo, Equation (4.5), (p. 95).
25. Dixon, M. F., Halperin, I., & Bilokon, P. A. (2020). *Machine Learning in Finance: From Theory to Practice*. Chapter 9: Reinforcement Learning, Section 5: Reinforcement Learning Methods, Sub-Section 5.4: SARSA and Q-Learning, (p. 313).
26. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym*. Chapter 4: Model-Free Approaches, Section: On-Policy SARSA, Equation (4.6), (p. 99).
27. Dixon, M. F., Halperin, I., & Bilokon, P. A. (2020). *Machine Learning in Finance: From Theory to Practice.* Chapter 9: Reinforcement Learning, Section 5: Reinforcement Learning Methods, Sub-Section 5.4: SARSA and Q-Learning, (p. 315).
28. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 4: Model-Free Approaches, Section: Q-Learning: An Off-Policy TD Control, Equation (4.10), (p. 104).

5. Technical Indicators, Data Processing, and Principal Component Analysis

a. Technical Indicators

1. Mitchell, Cory. "Understanding a Candlestick Chart." *Investopedia*, 25 Mar. 2021, www.investopedia.com/trading/candlestick-charting-what-is-it/.
2. "Bitstamp Data." *Www.cryptodatadownload.com*, www.cryptodatadownload.com/data/bitstamp/#google_vignette.
3. Chen, James. "Technical Indicator Definition." *Investopedia*, www.investopedia.com/terms/t/technicalindicator.asp.
4. Lobel, Ben, and Ben Lobel. "Technical Indicators Defined and Explained." DailyFX, www.dailyfx.com/education/technical-analysis-tools/technical-indicators.html.
5. "TA-Lib." *Mrjbq7.Github.io*, mrjbq7.github.io/ta-lib/.

6. Algorithm Training and Performance

a. Classical Statistical Algorithms

1. "Error Sum of Squares." *Stanford.edu*, 2020, hlab.stanford.edu/brian/error_sum_of_squares.html.
2. "XGBoost Parameters — Xgboost 1.5.2 Documentation." *Xgboost.readthedocs.io*, xgboost.readthedocs.io/en/stable/parameter.html.

b. Artificial Neural Networks

1. “An Overview of Gradient Descent Optimization Algorithms.” *Sebastian Ruder*, 19 Jan. 2016, ruder.io/optimizing-gradient-descent/index.html#adam.
2. Team, Keras. “Keras Documentation: Optimizers.” *Keras.io*, keras.io/api/optimizers/.
3. Team, Keras. “Keras Documentation: Dense Layer.” *Keras.io*, keras.io/api/layers/core_layers/dense/.
4. Gupta, Sparsh. “Most Common Loss Functions in Machine Learning.” *Medium*, 28 June 2020, towardsdatascience.com/most-common-loss-functions-in-machine-learning-c7212a99dae0.
5. Reynolds, Anh H. “Anh H. Reynolds.” *Anh H. Reynolds*, anhreynolds.com/blogs/cnn.html.
6. Saha, Sumit. “A Comprehensive Guide to Convolutional Neural Networks—the ELI5 Way.” *Towards Data Science*, Towards Data Science, 15 Dec. 2018, towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.
7. Amidi, Afshine, and Shervine Amidi. “CS 230 - Recurrent Neural Networks Cheatsheet.” *Stanford.edu*, 2019, stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks.
8. Team, Keras. “Keras Documentation: LSTM Layer.” *Keras.io*, keras.io/api/layers/recurrent_layers/lstm/.

c. Deep Reinforcement Learning

1. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 4: Deep Q-Learning, Section: Deep Q Networks, Equation (6.1), (p.155).
2. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 5: Function Approximation, Section: Introduction, Equation (5.1), (p. 124).
3. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 5: Function Approximation, Section: Batch Methods (DQN), Equation (5.22), (p. 150).
4. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 5: Function Approximation, Section: Batch Methods (DQN), Equation (5.24), (p. 150).
5. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 5: Function Approximation, Section: Batch Methods (DQN), (p. 150).
6. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 5: Function Approximation, Section: Batch Methods (DQN), Equation (5.25), (p. 151).
7. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 5: Function Approximation, Section: Batch Methods (DQN), (p. 151).
8. “Reinforcement Learning Tutorial Part 3: Basic Deep Q-Learning.” *Valohai.com*, valohai.com/blog/reinforcement-learning-tutorial-basic-deep-q-learning/.

9. Tidor-Vlad Pricope, 2021. "Deep Reinforcement Learning in Quantitative Algorithmic Trading: A Review," Papers 2106.00123, arXiv.org. Recommended citation from ideas.repec.org.
10. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 6: Deep Q-Learning, Section: Prioritized Replay, Equation (6.6), (p. 176).
11. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 6: Deep Q-Learning, Section: Prioritized Replay, Equation (6.7), (p. 176).
12. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 6: Deep Q-Learning, Section: Prioritized Replay, Equation (6.8), (p. 176).
13. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 6: Deep Q-Learning, Section: Prioritized Replay, Equation (6.9), (p. 177).
14. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 6: Deep Q-Learning, Section: Prioritized Replay, Equation (6.10), (p. 177).
15. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 6: Deep Q-Learning, Section: Prioritized Replay, Equation (6.11), (p. 177).
16. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 6: Deep Q-Learning, Section: Double Q-Learning, (p. 181).
17. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 6: Deep Q-Learning, Section: Double Q-Learning, (p. 182).
18. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 6: Deep Q-Learning, Section: Double Q-Learning, Equation (6.12), (p. 182).
19. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 6: Deep Q-Learning, Section: Dueling DQN, Equation (6.13), (p. 185).
20. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 6: Deep Q-Learning, Section: Dueling DQN, Equation (6.14), (p. 185).
21. Yoon, Chris. “Dueling Deep Q Networks.” *Medium*, Towards Data Science, 19 Oct. 2019, towardsdatascience.com/dueling-deep-q-networks-81ffab672751.
22. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 7: Policy Gradient Algorithms, Section: Policy Representation, (p. 210-211).
23. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 8: Combining Policy Gradient and Q-Learning, Section: General Framework to Combine Policy Gradient and Q-Learning, (p. 255).
24. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 7: Policy Gradient Algorithms, Section: Policy Gradient Representation, (p. 212).

25. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 7: Policy Gradient Algorithms, Section: Policy Gradient Representation, (p. 213).
26. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 7: Policy Gradient Algorithms, Section: Policy Gradient Representation, Equation (7.2), (p. 213).
27. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 7: Policy Gradient Algorithms, Section: Policy Gradient Representation, Equation (7.3), (p. 214).
28. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 7: Policy Gradient Algorithms, Section: Policy Gradient Representation, Equation (7.4), (p. 214).
29. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 7: Policy Gradient Algorithms, Section: Policy Gradient Representation, Equation (7.15 and 7.16), (p. 215).
30. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 7: Policy Gradient Algorithms, Section: Policy Gradient Representation, Equation (7.17), (p. 216).
31. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 8: Combining Policy Gradient and Q-Learning, Section: General Framework to Combine Policy Gradient and Q-Learning, (p. 251).
32. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 8: Combining Policy Gradient and Q-Learning, Section: General Framework to Combine Policy Gradient and Q-Learning, (p. 255).
33. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 8: Combining Policy Gradient and Q-Learning, Section: General Framework to Combine Policy Gradient and Q-Learning, Equation (8.2), (p. 256).
34. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 8: Combining Policy Gradient and Q-Learning, Section: Deep Deterministic Policy Gradient, Equation (8.5), (p. 258).
35. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 8: Combining Policy Gradient and Q-Learning, Section: Deep Deterministic Policy Gradient, Equation (8.6), (p. 259).
36. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 8: Combining Policy Gradient and Q-Learning, Section: Deep Deterministic Policy Gradient, (p. 259).
37. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 8: Combining Policy Gradient and Q-Learning, Section: Deep Deterministic Policy Gradient, Equation (8.8), (p. 278).
38. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 8: Combining Policy Gradient and Q-Learning, Section: Deep Deterministic Policy Gradient, Equation (8.9), (p. 278).
39. Sanghi, N. (2021). *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym.* Chapter 8: Combining Policy Gradient and Q-Learning, Section: Deep Deterministic Policy Gradient, (p. 279).

B. Software/Packages

Packages Used:

Package	Version
Python	3.7.11
Numpy	1.18.5
Pandas	1.3.4
Matplotlib	3.3.2
Ski-kit learn	1.02
Seaborn	0.11.2
Statsmodels	0.13.2
XGBoost	1.5.2
Tensorflow	2.8.0
Pytorch	1.10.1+cpu
Gym	0.19.0

C. GitHub

All code and data are hosted in the following GitHub: https://github.com/ClarkUCD

1. "MNIST: CNN, Grid Search, Data Augmentation." *Kaggle.com*, www.kaggle.com/code/cedricb/mnist-cnn-grid-search-data-augmentation/notebook. Accessed 28 Mar. 2022. In the Jupyter Notebook *Artificial Neural Networks - Multi-Layer Perceptron Hour*
2. https://www.facebook.com/rtipaday. "ARIMA Model - Complete Guide to Time Series Forecasting in Python | ML+." *Machine Learning Plus*, 18 Feb. 2019, www.machinelearningplus.com/time-series/arima-model-time-series-forecasting-python/. In the Jupyter Notebook *Classical Statistical Methods and eXtreme Gradient Boosting - SARIMAX*