Deep Reinforcement Learning-based Portfolio Management

By

NITIN KANWAR

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington
in Partial Fulfilment of the Requirements for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

MAY 2019

ACKNOWLEDGEMENT

I would like to convey my deepest gratitude to my advisor, Dr. Jia Rao for giving me the opportunity to carry out this research. Without his trust and encouragement, I would never have been introduced to the exciting and burgeoning field of Deep Reinforcement Learning. His supervision throughout this thesis has been a stimulating intellectual experience. His continued guidance and support have contributed in the completion of an otherwise difficult and esoteric thesis.

I want to thank my committee members, Dr. Junzhou Huang and Dr. Dajiang Zhu, for their interest in my research and taking out the time to be a part of my thesis committee.

I would like to thank my friends and colleagues, most importantly Sudheer Raja and Akib Zaman for always taking out the time to discuss and debate ideas with me.

I would also like to extend my appreciation to the Computer Science and Engineering department for providing me with all the necessary facilities and infrastructure to carry out my Master's research.

Lastly, I would like to thank my beloved family for always encouraging me to pursue my goals. I would like to extend my thanks to all my friends who have been there to support and guide me during this research period.

ABSTRACT

Machine Learning is at the forefront of every field today. The subfields of Machine Learning called Reinforcement Learning and Deep Learning, when combined have given rise to advanced algorithms which have been successful at reaching or surpassing the human-level performance at playing Atari games to defeating multiple times champion at Go. These successes of Machine Learning have attracted the interest of the financial community and have raised the question if these techniques could also be applied in detecting patterns in the financial markets.

Until recently, mathematical formulations of dynamical systems in the context of Signal Processing and Control Theory have attributed to the success of Financial Engineering. But because of Reinforcement Learning, there has been improved **sequential decision making** leading to the development of **multistage stochastic optimization**, a key component in **sequential portfolio optimization** (asset allocation) strategies.

In this thesis, we explore how to optimally distribute a fixed set of stock assets from a given set of stocks in a portfolio to maximize the long term wealth of the Deep Learning trading agent using Reinforcement Learning. We treat the problem as context-independent, meaning the learning agent directly interacts with the environment, thus allowing us to apply model free Reinforcement Learning algorithms to get optimized results. In particular, we focus on Policy Gradient and Actor Critic Methods, a class of state-of-the-art techniques which constructs an estimate of the optimal policy for the control problem by iteratively improving a parametric policy.

We perform a comparative analysis of the Reinforcement Learning based portfolio optimization strategy vs the more traditional “Follow the Winner”, “Follow the Loser”, and "Uniformly Balanced" strategies, and find that Reinforcement Learning based agents either far out perform all the other strategies, or behave as good as the best of them.

The analysis provides conclusive support for the ability of model-free Policy Gradient based Reinforcement Learning methods to act as universal trading agents.

Chapter 1

INTRODUCTION

Financial market applications routinely make use of engineering methods and systems like signal processing, control theory, and advanced statistical methods. Markets have been highly computerized and its advantages have been well understood and studied by the engineering communities.
More recently, Machine Learning (ML) has proven its mettle in different fields, which has made the finance community look in this direction. Many quantitative researchers have asked themselves if the techniques that have proven so successful in classifying images or beating Go world champions, could perform equally well on financial markets.

In this thesis, we explore how to optimally distribute stocks in a portfolio over a certain time interval via Reinforcement Learning (RL), a branch of ML that allows finding an optimal strategy for a sequential decision-making problem by directly interacting with the environment in an episodic manner.

In this introductory chapter, we briefly describe the objective of the thesis and highlight the research and application domains from which we draw motivation. This chapter thus prepares the stage on which we will move during the entire exposition.

1.1. Problem Definition

The aim of this report is to investigate the effectiveness of RL agents on the process of portfolio management. We will be working with a finite set of financial instruments mostly in the form of stocks. Our agent will learn how to optimally allocate funds to the aforementioned assets starting with a finite budget and try to maximize its overall wealth. The agent is trained and tested on real market data. Then its performance is compared with the existing standard portfolio management approaches including “Follow the Winner”, “Follow the Loser” and “Uniformly Rebalanced”.

1.2 Motivations

RL can be applied to a gamut of domains including Robotics, Medicine and Finance. There have been many successful applications of RL to the above domains. Inspired by these successes, we wanted to experiment with the idea of creating an agent which will be smart enough to understand the dynamics of the stock market and help traders to maximize their profits with minimal supervision and inputs.

We select stock market to test our hypothesis because it lends itself perfectly to the RL problem setting of agent-environment setup. Our portfolio management agent interacts with the stock market environment, taking actions as new states are presented to it. And with each action take, the environment sends the feedback to the agent in terms of the amount of wealth made or lost.

We were motivated to use RL machine learning paradigm because the stock market price dynamics are difficult to predict using Supervised and Unsupervised Learning paradigms. But, RL has the ability to capture the temporal difference in the signals and learn the market dynamics over time. We also think that RL agents would be the perfect candidates for portfolio management tasks because of their ability to focus on long term goals instead of getting stuck in a greedy short term approach.

We have also learned about the abilities of Deep Learning networks as function approximators to learn highly complex features in a high dimensional space. Thus, we would be using Deep Learning networks as our portfolio management agents which would be allowed to play around in the stock market environment to learn the dynamics and invest in the best stocks to increase the overall portfolio wealth.

1.3 Report Structure

The report is divided into 2 parts: Background (Part I) and Portfolio Management as Deep RL Problem (Part II). A brief outline of the project structure and chapters is provided below:

Chapter 2: Financial Engineering The objective of this chapter is to introduce the essential financial terms and concepts for understanding the methods developed later in this thesis.

Chapter 3: Deep Learning The chapter introduces the reader to the deep learning concepts that helps get a clear understanding of the different methodologies available and leveraged during this thesis work.

Chapter 4: Reinforcement Learning The objective of this chapter is to introduce the key concepts and terminology behind the field of Reinforcement Learning. This chapter is quintessential in the understanding the inner working of the RL agents.

Chapter 5: Deep Reinforcement Learning This chapter gives an understanding of the latest field of Deep Reinforcement Learning and various algorithms that we intend to use.

Chapter 6: Reinforcement Learning Applied to Finance This chapter illustrates on the previous work done in this field and acts as a motivation for the work in this thesis.

Chapter 7: Setup This chapter gives the details of how we have setup the portfolio management problem in the deep RL framework.

Chapter 8: Results We talk about the various experiments we performed and the results we received.

Chapter 9: Conclusion We come to a certain set of conclusions from our results and we discuss those

Chapter 10: Future Work We talk about the improvements and future work we can do.

PART I

Background

Chapter 2

FINANCIAL ENGINEERING

Financial engineering is the application of mathematical methods to the solution of problems in finance. It is also known as financial mathematics, mathematical finance, and computational finance.

Financial engineering draws on tools from applied mathematics, computer science, statistics, and economic theory.

Investment banks, commercial banks, hedge funds, insurance companies, corporate treasuries, and regulatory agencies apply the methods of financial engineering to such problems as new product development, derivative securities valuation, portfolio structuring and management, risk management, and scenario simulation.

Quantitative analysis has brought innovation, efficiency and rigor to financial markets and to the investment process.

2.1. Financial Terms and Concepts

2.1.1. Asset

An **asset** is an item of economic value. Examples of assets are cash (in hand or in a bank), stocks, loans and advances, accrued incomes etc. Our main focus on this report is on cash and stocks, but general principles apply to all kinds of assets.

2.1.2. Stocks

A stock (also known as "shares" or "equity) is a type of security that signifies proportionate ownership in the issuing company. This entitles the stockholder to that proportion of the corporation's assets and earnings.

Stocks are bought and sold predominantly on stock exchanges, though there can be private sales as well, and are the foundation of nearly every portfolio. These transactions have to conform to government regulations which are meant to protect investors from fraudulent practices. Historically, they have outperformed most other investments over the long run. These investments can be purchased from most on-line stock brokers.

2.1.3. Portfolio

A portfolio is a grouping of financial assets such as stocks, bonds, commodities, currencies and cash equivalents, as well as their fund counterparts, including mutual, exchange-traded and closed funds.

A portfolio can also consist of non-publicly tradable securities, like real estate, art, and private investments. Portfolios are held directly by investors and/or managed by financial professionals and money managers. Investors should construct an investment portfolio in accordance with their risk tolerance and their investing objectives. Investors can also have multiple portfolios for various purposes. It all depends on one's objectives as an investor.

Portfolio can be represented as a collection of multiple financial assets, and it can be represented in a vector form:

- **Constituents**: ***M*** assets of which it contains
- **Portfolio Vector** w_t: its *i*-th component represents the ratio of the total budget invested vs the investment in the *i*-th asset, such that:

$$w_t = [w_{1,t}, \quad w_{2,t}, \quad \dots, \quad w_{M,t}]^T \in \mathbb{R}^M \text{ and } \sum_{i=1}^{M} w_{i,t} = 1 \tag{2.1}$$

2.1.4. Portfolio Optimization

Portfolio optimization is the process of selecting the best portfolio (asset distribution), out of the set of all portfolios being considered, according to some objective. The objective typically maximizes factors such as expected return, and minimizes costs like financial risk. Factors being considered may range from tangible (such as assets, liabilities, earnings or other fundamentals) to intangible (such as selective divestment).

2.2. Financial Time Series

The dynamic nature of the economy, as a result of the non-static supply and demand balance, causes prices to evolve over time. This encourages to treat market dynamics as time-series and employ technical methods and tools for analysis and modeling.

2.2.1. Prices

Let p_t belongs to $\mathbb{R}$ be the **price** of an asset at discrete time index *t*, then the sequence $p_1, p_2, \dots, p_T$ is a univariate time-series. We can use $p_{i,t}$ to represent the price of an asset *i* at time *t*. We can use $p_{\text{asset } i,t}$ to represent the prices of different assets. Prices can thus can be represented by price vectors of length T, where T is the time to which we want to explore.

The price time series of an asset *i*, is the column vector $\vec{p}_{i,1:T}$, such that:

$$\vec{p}_{i,1:T} = \begin{bmatrix} p_{i,1} \\ p_{i,2} \\ \vdots \\ p_{i,T} \end{bmatrix} \in \mathbb{R}_{+}^{T} \tag{2.2}$$

where the arrow highlights the fact that it is a time-series. For convenience of portfolio analysis, we define the price vector p_T, such that:

$$p_t = [p_{1,t}, \quad p_{2,t}, \ldots, p_{M,t}] \in \mathbb{R}_{+}^{M} \tag{2.3}$$

where the *i*-th element is the asset price of the *i*-th asset in the portfolio at time *t*. Extending the single-asset time-series notation to the multivariate case, we form the asset price matrix $\vec{P}_{1:T}$, by stacking column-wise the T – samples price time-series of the M assets of the portfolio, then:

$$\vec{P}_{1:T} = [\vec{p}_{1,1:T'}, \quad \vec{p}_{2,1:T'}, \ldots, \vec{p}_{M,1:T}] = \begin{bmatrix} p_{1,1} & p_{2,1} & \cdots & p_{M,1} \\ p_{1,2} & p_{2,2} & \cdots & p_{M,2} \\ \vdots & \vdots & \ddots & \vdots \\ p_{1,T} & p_{2,T} & \cdots & p_{M,T} \end{bmatrix} \in \mathbb{R}_{+}^{T \times M} \tag{2.4}$$

2.2.2. Types of Prices

2.2.2.1. High Price

This price represents the security's intraday highest traded price.

2.2.2.2. Low Price

This price represents the security's intraday lowest traded price.

2.2.2.3. Closing Price

This price represents the security's price at the end of the day's business.

2.2.2.4. Opening Price

This price represents the security's price at the start of the day's business.

2.2.3. Volume

Volume is the number of shares or contracts traded in a security or an entire market during a given period of time. For every buyer, there is a seller, and each transaction contributes to the count of total volume. That is, when buyers and sellers agree to make a transaction at a certain price, it is considered one transaction. If only five transactions occur in a day, the volume for the day is five.

2.2.4. Returns

Absolute asset prices are not directly useful for an investor. On the other

hand, prices changes over time are of great importance, since they reflect the investment profit and loss, or more compactly, its **return**.

2.2.4.1. Simple Return

The percentage change in asset price from time (*t*-1) to time *t* is called the simple return of the asset. The formula to calculate Simple Return is:

$$r_t \triangleq \frac{p_t - p_{t-1}}{p_{t-1}} = \frac{p_t}{p_{t-1}} - 1 = R_t - 1 \in \mathbb{R} \qquad (2.5)$$

2.2.4.2. Gross Return

The gross return R_t represents the scaling factor of an investment in the asset at time (t-1). For example, a B dollar investment in an asset at time (t - 1) will worth BR_t dollars at time t. It is given by the ratio of an assets prices at times t and (t - 1), such that:

$$R_t \triangleq \frac{p_t}{p_{t-1}} \in \mathbb{R} \qquad (2.6)$$

2.2.4.3. Log Return

The log return is the log of the gross return. Even though Simple Return is a simpler way to calculate returns, it is asymmetric. Log returns provide a symmetric way to calculate the future value of the asset.

$$\rho_t \triangleq \ln\left(\frac{p_t}{p_{t-1}}\right) \ln(R_t) \in \mathbb{R} \qquad (2.7)$$

2.2.5. Transaction Cost

Transaction costs are expenses incurred when buying or selling a good or service. Transaction costs represent the labor required to bring a good or service to market, giving rise to entire industries dedicated to facilitating exchanges. In a financial sense, transaction costs include brokers' commissions and spreads, which are the differences between the price the dealer paid for a security and the price the buyer pays.

2.2.6. Portfolio Value

The total monetary value of the portfolio obtained by multiplying the weights of the assets with the daily prices.

2.2.7. Portfolio Management Strategies

Portfolio management strategies are set of rules that are followed by portfolio managers for optimal allocation of assets in a portfolio.

2.2.7.1. Follow the Winner

Follow the Winner approach is characterized by transferring portfolio weights from the under-performing assets (experts) to the outperforming ones.

2.2.7.2. Follow the Loser

The Follow the Loser approach assumes that the under-performing assets will revert and outperform others in the subsequent periods. Thus, their common

behavior is to move portfolio weights from the outperforming assets to the under-performing assets.

2.2.7.3. Uniform Constant Rebalanced Portfolios

The UCRP approach suggests that the wealth be equally distributed between the chosen assets in a portfolio without making any kind of changes throughout the trading period. This helps avoid the transaction costs incurred by the trading agent.

2.3. Technical Indicators

Technical indicators are used by financial analysts and portfolio managers to gauge the performance of the assets that they are interested in. A multitude of indicators exist in the financial domain but we are only going to talk about the ones' mentioned below.

2.3.1. Average Daily Return

The average return is the simple mathematical average of a series of returns generated over a period of time. An average return is calculated the same way a simple average is calculated for any set of numbers. The numbers are added together into a single sum, and then the sum is divided by the count of the numbers in the set. The formula to calculate Average Daily Return is as:

$$Average\ Return = \frac{Sum\ of\ Returns}{Number\ of\ Returns} \tag{2.8}$$

2.3.2. Sharpe Ratio

The Sharpe Ratio [1] is calculated by subtracting the risk-free rate from the return of the portfolio and dividing that result by the standard deviation of the portfolio's excess return.
The ratio describes how much excess return you receive for the extra volatility you endure for holding a riskier asset.

As per Wikipedia, the Sharpe Ratio can be calculated as follows:

$$S_a = \frac{E[R_a - R_b]}{\sigma_a} = \frac{E[R_a - R_b]}{\sqrt{\text{var}[R_a - R_b]}} \tag{2.9}$$

where R_a is the asset return, R_b is the risk free rate, $E[R_a - R_b]$ is the expected value of the excess of the asset return over the benchmark return, and $\sqrt{\text{var}[R_a - R_b]}$ is the standard deviation of the asset excess return.

Chapter 3

DEEP LEARNING

Deep Learning is a machine learning method that takes in an input X, and uses it to predict an output Y. Before we understand what deep learning is, we need to understand the basic building blocks of deep learning.

3.1. Perceptron

[2] Given a finite set of *m* inputs, we multiply each input by a weight then we sum up the weighted combination of inputs, add a bias and finally pass them through a non-linear activation function to get the output $\hat{y}$.

The bias θ_0 allows to add another dimension to the input space. Thus, the activation function still provides an output in case of an input vector of all zeros. It is somehow the part of the output that is independent of the input.

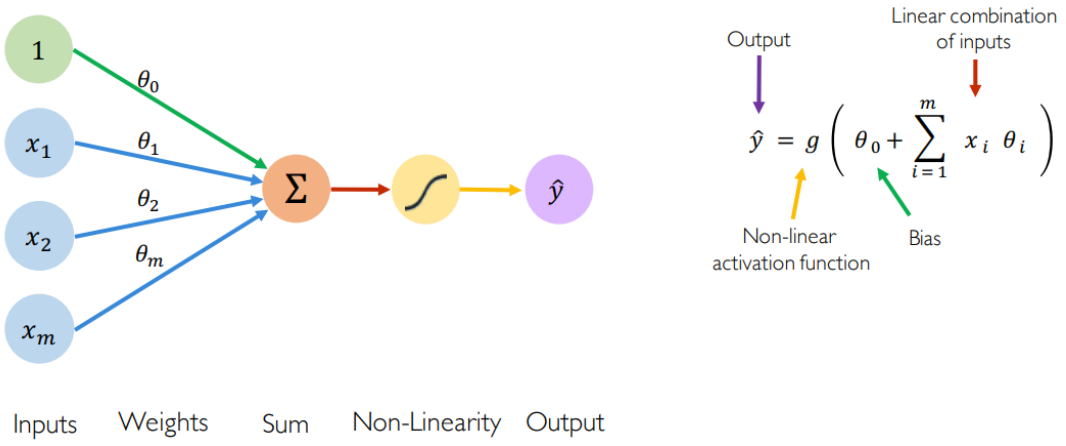


Figure 1: Architecture of Perceptron

3.2. Neural Network

Neural networks [3] are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. They interpret sensory data through a kind of machine perception, labeling or clustering raw input. The patterns they recognize are numerical, contained in vectors, into which all real-world data, be it images, sound, text or time series, must be translated.

Structurally, neural network is just a stacking of multiple perceptrons connected to the inputs and the outputs in different formations. A neural network with a hidden layer is often called a Multilayer Perceptron.

3.3. Activation Function

The purpose of the activation function [4] is to introduce non-linearities in the network. Linear activation functions produce linear decisions no matter the input distribution. Non-

linearities allow us to better approximate arbitrarily complex functions. Some activation functions are as below:

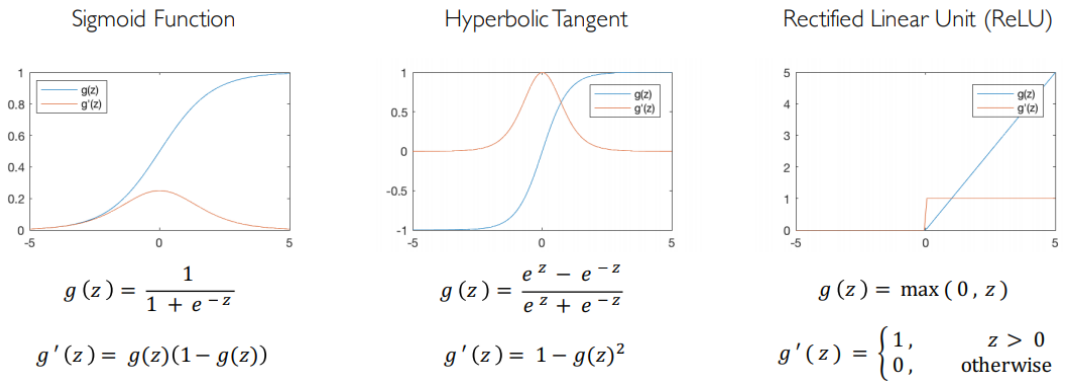


Figure 2: Different Activation Functions

3.4. Objective Function / Loss Function / Error Function / Cost Function

The Loss function (P.S also called Objective/Error/Cost function) [5] is a method of evaluating how well an algorithm models a dataset. If the predictions are totally off the target, the loss function will output a high number. If the predictions are good, the loss function will output a lower number. The loss function is the best indicator whether the algorithm is going in the right direction as changes are made to the model.

The loss function varies from measuring the absolute difference between a prediction and an actual value, to finding out the mean squared error, to calculating the cross entropy.

3.4.1. Different loss functions

3.4.1.1. Mean Absolute Error

Absolute Error calculates the absolute difference between the predicted value and the target value for each data point in a dataset.

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} |y^{(i)} - \hat{y}^{(i)}| \qquad (3.1)$$

3.4.1.2. Mean Squared Error

Mean Squared Error (MSE) is one of the most used basic loss functions. To calculate MSE, we take the difference between the predictions and the ground truth, square it and average it out across the whole dataset.

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2 \qquad (3.2)$$

3.4.1.3. Log Loss / Cross Entropy Loss

Log Loss is one of the most used loss functions for classification problems. It is a modification of the likelihood function with logarithms.

$$\mathcal{L} = -(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)) \quad (3.3)$$

3.5. Backpropagation

Backpropagation [6] is the method of propagating the information related to the error produced by the neural network in making a guess about the result, back through the network.
The parameters of the neural network have a relationship with the error the net produces, and when the parameters change, the error does, too. We change the parameters using an optimization algorithm.

3.6. Optimization Algorithms

Optimization algorithms [7] help minimize or maximize an Objective/Loss/Error/Cost function which is simply a mathematical function dependent on the model's internal learnable parameters which are used in computing the target values from the set of predictors used in the model. In the neural network models, the weights and bias are called the learnable parameters.

3.6.1. Different Optimization Algorithms

3.6.1.1. First Order Optimization Algorithms

These algorithms minimize or maximize a Loss function using its **gradient** values with respect to the model parameters. The First Order derivative tells us whether the function is decreasing or increasing at a particular point. First Order derivative basically gives us a line which is tangential to a point on the Loss function surface.

3.6.1.2. Second Order Optimization Algorithms

These algorithms use the second order derivative, also called the Hessian to minimize of maximize the Loss function. The Hessian is a matrix of the second order partial derivatives. Second Order derivative provides us with a quadratic surface which touches the curvature of the Loss function surface.

3.7. Gradient Descent Optimization Algorithms

3.7.1. Gradient Descent

Gradient Descent [8] is an algorithm used to calculate the gradient of an Error function with respect to the learnable parameters and then propagate this error back through the network updating the learnable parameters in the direction of the gradient to improve the overall learning of the model.

3.7.2. Variants of Gradient Descent

3.7.2.1. Batch Gradient Descent

This is a type of gradient descent which processes all the training examples for each iteration of gradient descent. But if the number of training examples is large, then batch gradient descent is computationally very expensive. Hence

if the number of training examples is large, then batch gradient descent is not preferred. Instead, we prefer to use stochastic gradient descent or mini-batch gradient descent.

3.7.2.2. Stochastic Gradient Descent

This is a type of gradient descent which processes 1 training example per iteration. Hence, the parameters are being updated even after one iteration in which only a single example has been processed. Hence this is quite faster than batch gradient descent. But again, when the number of training examples is large, even then it processes only one example which can be additional overhead for the system as the number of iterations will be quite large.

3.7.2.3. Mini Batch Gradient Descent

This is a type of gradient descent which works faster than both batch gradient descent and stochastic gradient descent. Here b examples where b<m are processed per iteration. So even if the number of training examples is large, it is processed in batches of b training examples in one go. Thus, it works for larger training examples and that too with lesser number of iterations.

3.7.3. Momentum

Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations. It does this by adding a fraction γ of the update vector of the past time step to the current update vector.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t \tag{3.4}$$

Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\gamma < 1$). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

3.7.4. Adam

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface. We compute the decaying averages of past and past squared gradients m_t and $v_t$respectively as follows:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{3.5}$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β1 and β2 are close to 1).
They counteract these biases by computing bias-corrected first and second moment estimates:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_t^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{3.6}$$

They then use these to update the parameters which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{3.7}$$

The authors propose default values of 0.9 for β1, 0.999 for β2, and 10^8 for ϵ. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

3.8. Deep Learning Architecture

A neural network architecture which has more than one hidden layer is called Deep Learning Architecture [9]. The addition of layers is what reflects the "Deep" in Deep Learning.

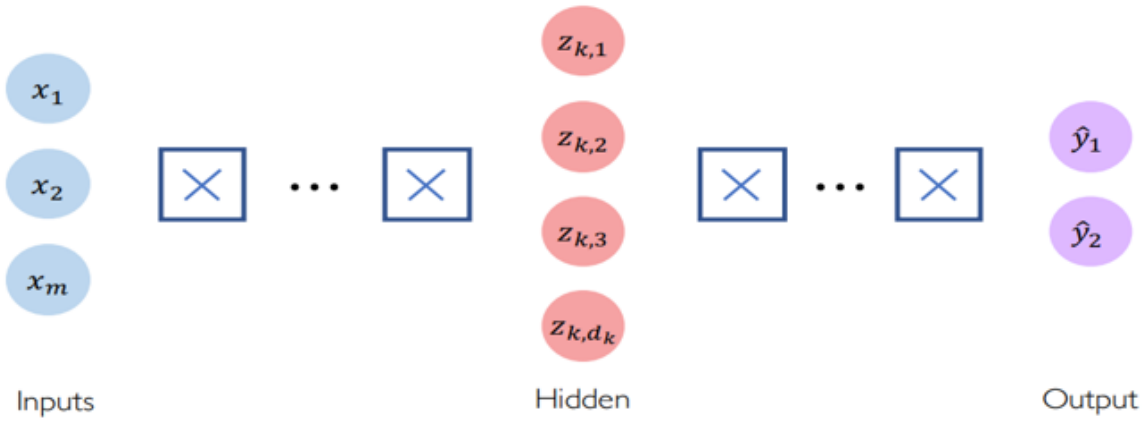


Figure 3: Multilayer Perceptron Architecture

3.9. Training

The outputs generated by Deep Neural Networks may not be the ones we expected. This means that the neural network has not learned the features which would help it to predict the correct results. In order to help the neural network, learn and get better at predicting results which are more close to the actual expected results, we need to train it with multiple samples of data.

In the training process, we will send inputs to the neural network and wait for it to predict an output. Once the neural network has generated an output, we will provide the correct target value to the network and calculate the amount of divergence in the results. The divergence is called **Error**.

This divergence is calculated by setting up an **Objective Function or Loss Function**. The motive of the neural network is to reduce the error in its predicted results. In terms of the Loss Function, it is said to find the global minimum.
During the training process, the weights of the neural networks adjust themselves in a way to improve the mapping generated to be very close to the outputs of the training data. Learning of neural network can be supervised in which correct choice of output is provided in training data or, unsupervised in which no output is provided.

The overall goal behind training the neural network on training data is to make sure that the network learns the features and is able to generalize it to testing samples. If the network performs well then this mean the neural network has achieved optimum learning. But, there are times when the network learns really well on the training dataset by reducing the error in its predictions to the minimum, but fails to generalize on the testing dataset. This situation is termed as **Overfitting** to the training dataset.

There are a multitude of ways to avoid overfitting. Some of them are below:

3.9.1. Regularization

Regularization [10] involves adding an extra element to the loss function, which punishes our model for being too complex or, in simple words, for using too high values in the weight matrix. This way we try to limit its flexibility, but also encourage it to build solutions based on multiple features.

Cost Function = Loss Function + Regularization Term

Two popular versions of this method are:

- L1 - Least Absolute Deviations (LAD)

$$Cost function = Loss + \frac{\lambda}{2m} * \sum \| w\| \tag{3.8}$$

- L2 - Least Square Errors (LS).

$$CostFunction = Loss + \frac{\lambda}{2m} * \sum \|w\|^2 \tag{3.9}$$

3.9.2. Regularization Rate

Regularization Rate λ, is the regularization parameter. It is the hyper-parameter whose value is optimized for better results.

3.9.3. Dropout

Dropout [11] is another widely used regularization technique used in Deep Learning to avoid overfitting. At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connection.

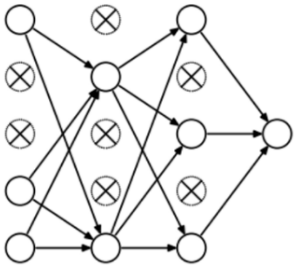


Figure 4:Dropout Regularization

As we can see in the figure, dropout randomly drops nodes and removes their incoming and outgoing connections. This way, the neural network doesn't over-fit to a certain set of features and weights. The dropout rate is a hyper-parameter that needs to be passed to the neural network.

Chapter 4

REINFORCEMENT LEARNING

Reinforcement learning (RL) [12] refers to both a learning problem and a sub-field of machine learning [13]. As a learning problem [14], it refers to learning to control a system (environment) so as to maximize some numerical value, which represents a long-term objective (discounted cumulative reward signal).

4.1. Key concepts and terminology

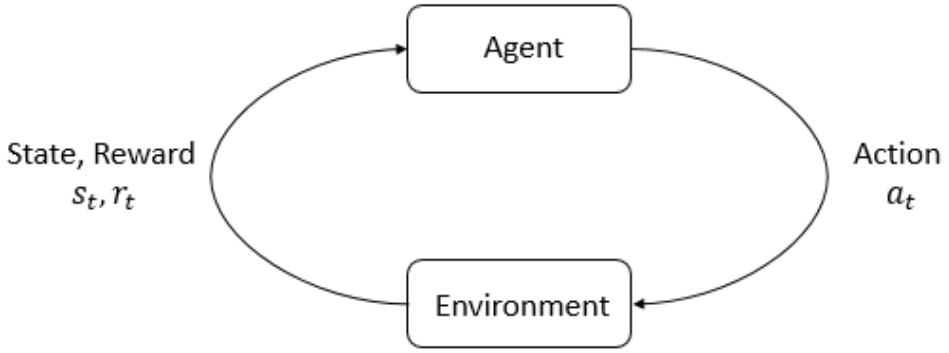


Figure 5: Reinforcement Learning Process

The main characters of RL are the **agent** and the **environment**. The environment is the world that the agent lives in and interacts with. At every step of interaction, the agent sees a (possibly partial) observation of the state of the world, and then decides on an action to take. The environment changes when the agent acts on it, but may also change on its own.

The agent also perceives a **reward** signal from the environment, a number that tells it how good or bad the current world state is. The goal of the agent is to maximize its cumulative reward, called **return**. Reinforcement learning methods are ways that the agent can learn behaviors to achieve its goal.

4.1.1. States and Observations

A state ***s*** is a complete description of the state of the world. There is no information about the world which is hidden from the state. An observation ***o*** is a partial description of a state, which may omit information.

When the agent is able to observe the complete state of the environment, we say that the environment is **fully observed**. When the agent can only see a partial observation, we say that the environment is **partially observed**.

4.1.2. Action Spaces

Different environments allow different kinds of actions. The set of all valid actions in a given environment is often called the **action space**. Some environments, like Atari and Go, have **discrete action spaces**, where only a finite number of moves are available to the agent. Other environments, like where the agent controls a robot in a physical world, have **continuous action spaces**. In continuous spaces, actions are real-valued vectors.

4.1.3. Policies

A policy is a rule used by an agent to decide what actions to take. It can be **deterministic**, in which case it is usually denoted by $\boldsymbol{\mu}$:

$$a_t = \mu(s_t) \tag{4.1}$$

or it may be **stochastic**, in which case it is usually denoted by π:

$$a_t \sim \pi(\cdot\,|s_t) \tag{4.2}$$

Because the policy is essentially the agent's brain, it's not uncommon to substitute the word "policy" for "agent", e.g. saying "The policy is trying to maximize reward."

4.1.4. Trajectory / Episodes / Rollouts

A trajectory $\boldsymbol{\tau}$ is a sequence of states and actions in the world,

$$\tau = (s_0, a_0, s_1, a_1, \ldots) \tag{4.3}$$

The very first state of the world, s_0, is randomly sampled from the start-state distribution, sometimes denoted by ρ_0:

$$s_0 \sim \rho_0(\cdot) \tag{4.4}$$

State transitions (what happens to the world between the state at time t, s_t, and the state at $t + 1$, s_{t+1}, are governed by the natural laws of the environment, and depend on only the most recent action, a_t. They can be either deterministic,

$$s_{t+1} = f(s_t, a_t) \tag{4.5}$$

or stochastic,

$$s_{t+1} \sim P(\cdot\,|s_t, a_t) \tag{4.6}$$

Actions come from an agent according to its policy.

4.1.5. Reward and Return

The reward function $\boldsymbol{R}$ is critically important in reinforcement learning. It depends on the current state of the world, the action just taken, and the next state of the world:

$$r_t = R(s_t, a_t, s_{t+1}) \tag{4.7}$$

although frequently this is simplified to just a dependence on the current state, $r_t = R(s_t)$, or state-action pair $r_t = R(s_t\,, a_t\,)$.

The goal of the agent is to maximize some notion of cumulative reward over a trajectory, but this actually can mean a few things. We'll notate all of these cases with $R(\tau)$, and it will either be clear from context which case we mean, or it won't matter (because the same equations will apply to all cases).

One kind of return is the **finite-horizon undiscounted return**, which is just the sum of rewards obtained in a fixed window of steps:

$$R(\tau) = \sum_{t=0}^{T} r_t \tag{4.8}$$

Another kind of return is the **infinite-horizon discounted return**, which is the sum of all rewards *ever* obtained by the agent, but discounted by how far off in the future they're obtained. This formulation of reward includes a discount factor $\gamma \in (0, 1)$:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{4.9}$$

4.1.6. The RL Problem

Whatever the choice of return measure (whether infinite-horizon discounted, or finite-horizon undiscounted), and whatever the choice of policy, the goal in RL is to select a policy which maximizes **expected return** when the agent acts according to it.

To talk about expected return, we first have to talk about probability distributions over trajectories.

Let's suppose that both the environment transitions and the policy are stochastic. In this case, the probability of a T -step trajectory is:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t) \tag{4.10}$$

The expected return (for whichever measure), denoted by $J(\pi)$, is then:

$$J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) = \underset{\tau\sim\pi}{\mathrm{E}}[R(\tau)] \tag{4.11}$$

The central optimization problem in RL can then be expressed by

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{4.12}$$

with π^* being the **optimal policy**.

4.1.7. Value Functions

It's often useful to know the **value** of a state, or state-action pair. By value, we mean the expected return if you start in that state or state-action pair, and then act according to a particular policy forever after. **Value functions** are used, one way or another, in almost every RL algorithm.

There are four main functions of note here.

1. The **On-Policy Value Function**, $V^{\pi}(s)$, which gives the expected return if you start in state *s* and always act according to policy π:

$$V^{\pi}(s) = \underset{\tau \sim \pi}{\mathrm{E}}[R(\tau)|s_0 = s] \tag{4.13}$$

2. The **On-Policy Action-Value Function**, $Q^{\pi}(s,a)$, which gives the expected return if you start in state *s*, take an arbitrary action *a* (which may not have come from the policy), and then forever after act according to policy π:

$$Q^{\pi}(s,a) = \underset{\tau \sim \pi}{\mathrm{E}}[R(\tau)|s_0 = s, a_0 = a] \tag{4.14}$$

3. The **Optimal Value Function**, $V^{*}(s)$, which gives the expected return if you start in state *s* and always act according to the *optimal* policy in the environment:

$$V^{*}(s) = \max_{\pi} \mathrm{E}_{\tau \sim \pi}[R(\tau)|s_0 = s] \tag{4.15}$$

4. The **Optimal Action-Value Function**, $Q^{*}(s,a)$, which gives the expected return if you start in state *s*, take an arbitrary action *a*, and then forever after act according to the *optimal* policy in the environment:

$$Q^{*}(s,a) = \max_{\pi} \underset{\tau \sim \pi}{\mathrm{E}}[R(\tau)|s_0 = s, a_0 = a] \tag{4.16}$$

There are two key connections between the value function and the action-value function that come up pretty often:

$$V^{\pi}(s) = \underset{a \sim \pi}{E}[Q^{\pi}(s,a)] \tag{4.17}$$

and

$$V^{*}(s) = \max_{a} Q^{*}(s,a) \tag{4.18}$$

4.1.8. The Optimal Q-Function and the Optimal Action

There is an important connection between the optimal action-value function $Q^{*}(s,a)$ and the action selected by the optimal policy. By definition, $Q^{*}(s,a)$ gives the expected return for starting in state *s*, taking (arbitrary) action *a*, and then acting according to the optimal policy forever after.

The optimal policy in *s* will select whichever action maximizes the expected return from starting in *s*. As a result, if we have Q^{*}, we can directly obtain the optimal action, $a^{*}(s)$, via

$$a^{*}(s) = \arg\max_{a} Q^{*}(s,a) \tag{4.19}$$

4.1.9. Bellman Equations

All four of the value functions obey special self-consistency equations called **Bellman equations** [15]. The basic idea behind the Bellman equations is this:

The value of your starting point is the reward you expect to get from being there, plus the value of wherever you land next.

The Bellman equations for the on-policy value functions are

$$V^{\pi}(s) = \mathrm{E}_{\substack{a\sim\pi \\ s'\sim P}}[r(s,a) + \gamma V^{\pi}(s')] \tag{4.20}$$

$$Q^{\pi}(s,a) = \mathrm{E}_{s'\sim P}[r(s,a) + \gamma \mathrm{E}_{a'\sim\pi}[Q^{\pi}(s',a')]] \tag{4.21}$$

where $s' \sim P$ is shorthand for $s' \sim P(.|s,a)$, indicating that the next state s' is sampled from the environment's transition rules; $a \sim \pi$ is shorthand for $a \sim \pi(.|s)$; and $a' \sim \pi$ is shorthand for $a' \sim \pi(.|s')$.

The Bellman equations for the optimal value functions are

$$V^{*}(s) = \max_{a} \mathrm{E}_{s'\sim P}[r(s,a) + \gamma V^{*}(s')] \tag{4.22}$$

$$Q^{*}(s,a) = \mathrm{E}_{s'\sim P}[r(s,a) + \gamma \max_{a'} Q^{*}(s',a')] \tag{4.23}$$

The crucial difference between the Bellman equations for the on-policy value functions and the optimal value functions, is the absence or presence of the max over actions. Its inclusion reflects the fact that whenever the agent gets to choose its action, in order to act optimally, it has to pick whichever action leads to the highest value.

The term "Bellman backup" comes up quite frequently in the RL literature. The Bellman backup for a state, or state-action pair, is the right-hand side of the Bellman equation: the reward-plus-next-value.

4.1.10. Advantage Functions

The advantage function corresponding to a policy π describes how much better it is to take a specific action *a* in state *s*, over randomly selecting an action according to $\pi(.|s)$, assuming you act according to π forever after. Mathematically, the advantage function is defined by

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{4.24}$$

4.1.11. Markov Decision Processes (MDPs)

An MDP is a 5-tuple, $\langle S, A, R, P, \rho_0 \rangle$, where

- *S* is the set of all valid states,
- *A* is the set of all valid actions,

- $R: S \times A \times S \to \mathbb{R}$ is the reward function, with $r_t = R(s_t, a_t, s_{t+1})$,
- $P: S \times A \to \mathcal{P}(S)$ is the transition probability function, with $P(s'|s, a)$ being the probability of transitioning into state s' if you start in state s and take action a,
- and ρ_0 is the starting state distribution.

The name Markov Decision Process [16] refers to the fact that the system obeys the **Markov property**: transitions only depend on the most recent state and action, and no prior history.

4.1.12. Actor Critic Methods

The Actor-Critic learning algorithm is used to represent the policy function independently of the value function. The policy function structure is known as the *actor*, and the value function structure is referred to as the *critic*. The actor produces an action given the current state of the environment, and the critic produces a TD (Temporal-Difference) error signal given the state and resultant reward. If the critic is estimating the action-value function *Q (s, a)*, it will also need the output of the actor. The output of the critic drives learning in both the actor and the critic. In Deep Reinforcement Learning, neural networks can be used to represent the actor and critic structures.

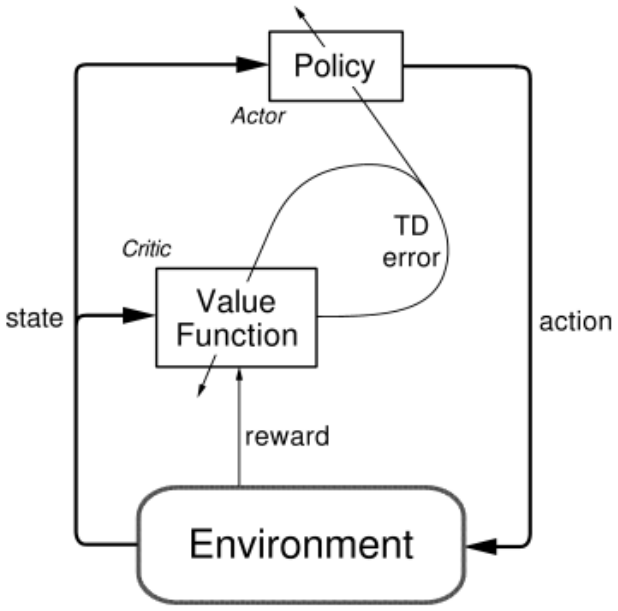


Figure 6: Actor Critic RL Architecture

4.2. Taxonomy of RL Algorithms

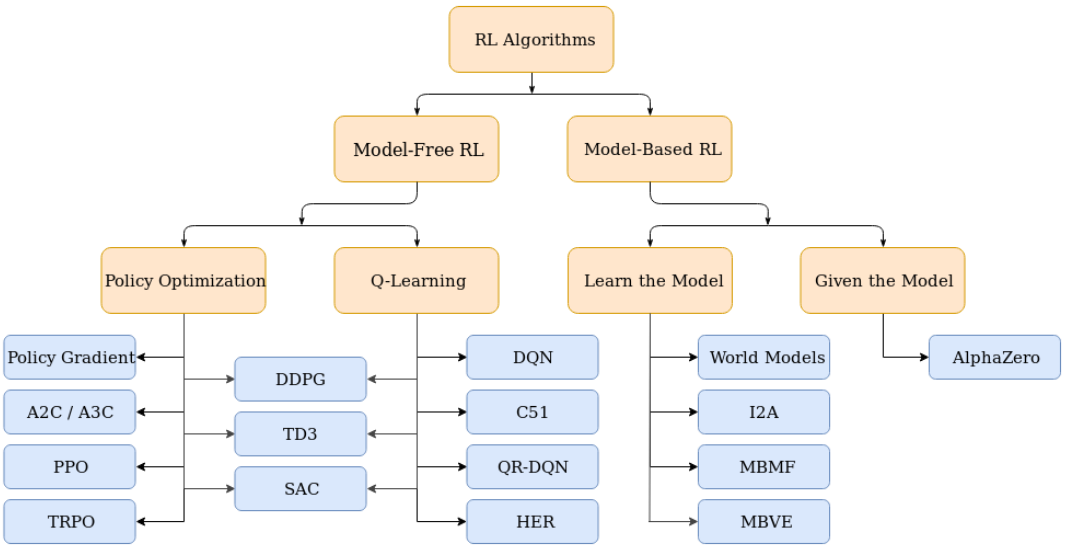


Figure 7: Taxonomy of RL Algorithms

4.2.1. Model Free Vs Model Based RL

RL algorithms are broadly divided based on the concept whether the agent has access to a model of the environment. By a model of the environment, we mean a function which predicts state transitions and rewards.

4.2.1.1. Model Based RL Algorithms

Model based algorithms allow the agent to plan by thinking ahead, seeing what would happen for a range of possible choices, and explicitly deciding between its options. Agents can then distill the results from planning ahead into a learned policy. A particularly famous example of this approach is AlphaZero.

The main downside is that a ground-truth model of the environment is usually not available to the agent. If an agent wants to use a model in this case, it has to learn the model purely from experience, which creates several challenges. The biggest challenge is that bias in the model can be exploited by the agent, resulting in an agent which performs well with respect to the learned model, but behaves sub-optimally in the real environment.

4.2.1.2. Model Free RL Algorithms

Algorithms which don't use a model of the environment are called Model Free RL methods. These algorithms do not make an effort to learn the underlying dynamics that govern how an agent interacts with the environment. Model Free methods forego the potential gains in sample

efficiency from using a model in order to be easier to implement and tune. Model-free algorithms directly estimate the optimal policy or value function through algorithms such as policy iteration or value iteration. This is much more computationally efficient. Model Free algorithms are explored much more widely compared to Model Based RL Algorithms.

There are two main approaches to representing and training agents with Model Free RL:

- **Policy Optimization**
- **Q Learning**

4.2.1.2.1. Policy Optimization

Methods in this family represent a policy explicitly as $\pi_\theta(a|s)$. They optimize the parameters θ either directly by gradient ascent on the performance objective $J(\pi_\theta)$, or indirectly, by maximizing local approximations of $J(\pi_\theta)$. This optimization is almost always performed **on-policy**, which means that each update only uses data collected while acting according to the most recent version of the policy. Policy optimization also usually involves learning an approximator $V_\phi(s)$ for the on-policy value function $V^\pi(s)$, which gets used in figuring out how to update the policy.

A couple of examples of policy optimization methods are:

- **Actor Critic Methods** (A2C/A3C), which performs gradient ascent to directly maximize performance.
- **Proximal Policy Optimization**, whose updates indirectly maximize performance, by instead maximizing a *surrogate objective* function which gives a conservative estimate for how much $J(\pi_\theta)$ will change as a result of the update.

4.2.1.2.2. Q-Learning

Methods in this family learn an approximator $Q_\theta(s,a)$ for the optimal action-value function, $Q^*(s,a)$. Typically, they use an objective function based on the Bellman equation. This optimization is almost always performed **off-policy**, which means that each update can use data collected at any point during training, regardless of how the agent was choosing to explore the environment when the data was obtained. The corresponding policy is obtained via the connection between Q^* and π^*: the actions taken by the Q-learning agent are given by

$$a(s) = \arg\max_{a} Q_\theta(s, a) \tag{4.25}$$

Example of Q-learning methods include

- **Deep Q Network**, a classic which substantially launched the field of deep RL.

4.2.1.3. Trade-offs Between Policy Optimization and Q-Learning

The primary strength of policy optimization methods is that they are principled, in the sense that you directly optimize for the thing you want. This tends to make them stable and reliable. By contrast, Q-learning methods only indirectly optimize for agent performance, by training Q_θ to satisfy a self-consistency equation. There are many failure modes for this kind of learning, so it tends to be less stable. But, Q-learning methods gain the advantage of being substantially more sample efficient when they do work, because they can reuse data more effectively than policy optimization techniques.

4.2.1.4. Interpolating Between Policy Optimization and Q-Learning

Policy optimization and Q-learning are not the only two available algorithms, there exist a range of algorithms that live in between the two extremes. Algorithms that live on this spectrum are able to carefully trade-off between the strengths and weaknesses of either side. Examples include

- Deep Deterministic Policy Gradient, an algorithm which concurrently learns a deterministic policy and a Q-function by using each to improve the other.

Chapter 5

DEEP REINFORCEMENT LEARNING

5.1. Vanilla Policy-Gradient (VPG)

Policy-Gradient (PG) [17] algorithms optimize a policy end-to-end by computing noisy estimates of the gradient of the expected reward of the policy and then updating the policy in the gradient direction. Traditionally, PG methods have assumed a stochastic policy $\mu\ (a \mid s)$, which gives a probability distribution over actions. Ideally, the algorithm sees lots of training examples of high rewards from good actions and negative rewards from bad actions. Then, it can increase the probability of the good actions. In practice, you tend to run into plenty of problems with vanilla-PG; for example, getting one reward signal at the end of a long episode of interaction with the environment makes it difficult to ascertain exactly which action was the good one. This is known as the **credit assignment problem**. For RL problems with continuous action spaces, VPG is all but useless. You can, however, get VPG to work with some RL domains that take in visual inputs and have discrete action spaces with a convolutional neural network representing your policy.

The key idea underlying policy gradients is to push up the probabilities of actions that lead to higher return, and push down the probabilities of actions that lead to lower return, until you arrive at the optimal policy. Key features to remember about VPG:

- VPG is an on-policy algorithm.
- VPG can be used for environments with either discrete or continuous action spaces.

Let π_θ denote a policy with parameters θ, and $J(\pi_\theta)$ denote the expected finite-horizon undiscounted return of the policy. The gradient of $J(\pi_\theta)$ is

$$\nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathrm{E}}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t)] \tag{5.1}$$

where τ is a trajectory and A^{π_θ} is the advantage function for the current policy.

The policy gradient algorithm works by updating policy parameters via stochastic gradient ascent on policy performance:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J\left(\pi_{\theta_k}\right) \tag{5.2}$$

Policy gradient implementations typically compute advantage function estimates based on the infinite-horizon discounted return, despite otherwise using the finite-horizon undiscounted policy gradient formula.

5.1.1. Exploration vs. Exploitation

VPG trains a stochastic policy in an on-policy way. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This may cause the policy to get trapped in local optima.

5.1.2. Vanilla Policy Gradient Algorithm Pseudocode

Algorithm 1 Vanilla Policy Gradient Algorithm

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

7: Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

or via another gradient ascent algorithm like Adam.
8: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
9: **end for**

Figure 8:Vanilla Policy Gradient Pseudocode

5.2. Deep Deterministic Policy Gradient (DDPG)

Google DeepMind devised a solid algorithm to tackle the continuous action space problem. They built off the work of David Silver et al. on Deterministic Policy Gradients [18] to come up with an off policy and model free policy gradient actor-critic algorithm called Deep Deterministic Policy Gradient (DDPG) [19].

DDPG is a policy gradient algorithm that uses a stochastic behavior policy for good exploration but estimates a deterministic target policy, which is much easier to learn. It is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.

This approach is closely connected to Q-learning, and is motivated the same way: if you know the optimal action-value function $Q^*(s,a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving

$$a^*(s) = \arg\max_a Q^*(s,a) \qquad (5.3)$$

DDPG interleaves learning an approximator to $Q^*(s,a)$, with learning an approximator to $a^*(s)$, and it does so in a way which is specifically adapted for environments with continuous action spaces. But what does it mean that DDPG is adapted specifically for environments with continuous action spaces? It relates to how we compute the max over actions in $\max_a Q^*(s,a)$.

When there are a finite number of discrete actions, the max poses no problem, because we can just compute the Q-values for each action separately and directly compare them. But when the action space is continuous, we can't exhaustively evaluate the space, and solving the optimization problem is highly non-trivial. Using a normal optimization algorithm would make calculating $\max_a Q^*(s,a)$ a painfully expensive subroutine. And since it would need to be run every time the agent wants to take an action in the environment, this is unacceptable.

Because the action space is continuous, the function $Q^*(s,a)$ is presumed to be differentiable with respect to the action argument. This allows us to set up an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then, instead of running an expensive optimization subroutine each time we wish to compute $\max_a Q(s,a)$, we can approximate it with $\max_a Q(s,a) \approx Q(s,\mu(s))$.

5.2.1. The Q-Learning Side of DDPG

The Bellman equation describing the optimal action-value function $Q^*(s,a)$ is given by,

$$Q^*(s,a) = \mathop{\mathrm{E}}_{s'\sim P}[r(s,a) + \gamma \max_{a'} Q^*(s',a')] \qquad (5.4)$$

where $s' \sim P$ is shorthand for saying that the next state, s', is sampled by the environment from a distribution P(. | *s*, *a*). This Bellman equation is the starting point for learning an approximator to $Q^*(s,a)$. Suppose the approximator is a neural network $Q_\phi(s,a)$, with parameters ϕ, and that we have collected a set $\mathcal{D}$ of transitions (s,a,r,s',d) (where d indicates whether state $s'$is terminal). We can set up a **mean-squared Bellman error (MSBE)** function, which tells us roughly how closely Q_ϕ comes to satisfying the Bellman equation:

$$L(\phi,\mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}}[(Q_\phi(s,a) - (r + \gamma(1-d)\underset{a'}{max}Q_\phi(s',a')))^2] \quad (5.5)$$

Here, in evaluating (1 – *d*), we've used a Python convention of evaluating True to 1 and False to zero. Thus, when *d* == True, which is to say, when s' is a terminal state, the Q-function should show that the agent gets no additional rewards after the current state.

Q-learning algorithms for function approximators, such as DQN (and all its variants) and DDPG, are largely based on minimizing this MSBE loss function. There are two main tricks employed by all of them which are worth describing, and then a specific detail for DDPG.

5.2.1.1. Replay Buffers

All standard algorithms for training a deep neural network to approximate $Q^*(s,a)$ make use of an experience replay buffer. This is the set $\mathcal{D}$ of previous experiences. In order for the algorithm to have stable behavior, the replay buffer should be large enough to contain a wide range of experiences, but it may not always be good to keep everything. If you only use the very-most recent data, you will over-fit to that and things will break; if you use too much experience, you may slow down your learning. This may take some tuning to get right.

5.2.1.2. Target Networks

Q-learning algorithms make use of **target networks**. The term

$$r + \gamma(1-d)\underset{a'}{max}Q_\phi(s',a') \quad (5.6)$$

is called the **target**, because when we minimize the MSBE loss, we are trying to make the Q-function be more like this target. Problematically, the target depends on the same parameters we are trying to train: ϕ. This makes MSBE minimization unstable. The solution is to use a set of parameters which comes close to ϕ, but with a time delay—that is to say, a second network, called the target network, which lags the first. The parameters of the target network are denoted ϕ_{targ} .

In DQN-based algorithms, the target network is just copied over from the main network every some-fixed-number of steps. In DDPG-style algorithms, the target network is updated once per main network update by polyak averaging:

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi \qquad (5.7)$$

where ρ is a hyperparameter between 0 and 1 (usually close to 1).

5.2.1.3. Calculating the Max Over Actions in the Target

As mentioned earlier: computing the maximum over actions in the target is a challenge in continuous action spaces. DDPG deals with this by using a **target policy network** to compute an action which approximately maximizes $Q_{\phi_{\text{targ}}}$. The target policy network is found the same way as the target Q-function: by polyak averaging the policy parameters over the course of training.

Putting it all together, Q-learning in DDPG is performed by minimizing the following MSBE loss with stochastic gradient descent:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\text{E}}[(Q_\phi(s,a) - (r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))))^2] \quad (5.8)$$

where $\mu_{\theta_{\text{targ}}}$ is the target policy.

5.2.2. The Policy Learning Side of DDPG

Policy learning in DDPG is fairly simple. We want to learn a deterministic policy $\mu_\theta(s)$ which gives the action that maximizes $Q_\phi(s,a)$ Because the action space is continuous, and we assume the Q-function is differentiable with respect to action, we can just perform gradient ascent (with respect to policy parameters only) to solve

$$\max_\theta \underset{s\sim\mathcal{D}}{\text{E}}[Q_\phi(s, \mu_\theta(s))] \qquad (5.9)$$

Note that the Q-function parameters are treated as constants here.

5.2.2.1. Exploration vs. Exploitation

DDPG trains a deterministic policy in an off-policy way. Because the policy is deterministic, if the agent were to explore on-policy, in the beginning it would probably not try a wide enough variety of actions to find useful learning signals. To make DDPG policies explore better, we add noise to their actions at training time. The authors of the original DDPG paper recommended time-correlated OU noise, but more recent results suggest that uncorrelated, mean-zero Gaussian noise works perfectly well. Since the latter is simpler, it is preferred.

5.2.2.2. Deep Deterministic Policy Gradient Pseudocode

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

Figure 9: Deep Deterministic Policy Gradient Pseudocode

Chapter 6

REINFORCEMENT LEARNING APPLIED TO FINANCE

There are a multitude of papers which have already used Reinforcement Learning in trading stock, portfolio management and portfolio optimization.

Moody et al. were the pioneers in applying the RL paradigm to the problem of stock trading and portfolio optimization. In [20], [21] and [22] they proposed the idea of Recurrent Reinforcement Learning (RRL) for Direct Reinforcement. RRL is an adaptive policy search algorithm that can learn an investment strategy on-line. Direct Reinforcement was a term coined to show algorithms that don't need to learn a value function in order to derive a policy. In other words, policy gradient algorithms in a Markov Decision Process framework are generally referred to as Direct Reinforcement. In [23] Moody et al. showed that a differential form of the Sharpe Ratio and Downside Deviation Ratio can be formulated to enable efficient on-line learning with Direct Reinforcement.

In [24] David W. Lu used the idea of Direct Reinforcement with an LSTM learning agent to learn how to trade in a Forex and commodity futures market. Du et al. in [25] use value function based algorithm Q Learning for algorithmic trading. They use different forms of value functions like interval profit, Sharp Ratio and derivative Sharp Ratio to evaluate the performance of the approach.

In [26] Tang et al. used an actor-critic based portfolio investment method taking into consideration the risks involved in asset investment. The paper uses approximate dynamic programming to setup a Markov Decision model for the multi-time segment portfolio with transaction cost.

Jiang et al. in [27] is one of the first papers which provides a detailed Deep Reinforcement Learning framework which can be used in the task of Portfolio Management in a cryptocurrency market exchange. They used the concept of a Portfolio Vector Memory to help train the network, which they call the Ensemble of Identical Independent Evaluators (EIIE). They take into consideration market risks and the transaction costs associated with buying and selling assets in a stock exchange.

We use the work by Jiang et al. [27] as a reference while modeling the replay buffer and neural network architectures.

PART II

Portfolio Management as Deep RL Problem

Chapter 7

SETUP

7.1. Data Preparation

We have used American Stocks data from Yahoo Finance. We have the stocks data for the following 10 companies: Boeing, Coca Cola, Ford, IBM, GE, JP Morgan, Microsoft, Nike, Walmart, and Exxon Mobil. Out of these ten, we decide to work with **M** stocks which are chosen randomly.

We have downloaded the data from March 24th, 1986 till March 22nd, 2019. This means we have 12054 days' worth of data for each stock. The downloaded data contains the Date, Open Price, High Price, Low Price, Close Price, Adjusted Close Price, Volume and Stock Ticker. All the prices are in Dollars, while the volume is defined in the number of shares traded for that ticker on that day.

We divide the data as per the following:
Total Days = 12,054 days
Train Days = 8030 days
Validate Days = 868 days
Test Days = 3152 days

For days, where we don't have the stock data available, we maintain the time series by filling the empty price data with the close price of the previous day and we also set the volume to 0 to indicate that the market is closed on that day.

In order to come up with a general agent which is robust with different stocks, we will normalize the price data. We will divide the opening price, closing price, high price, and low price by the highest closing price of the total period.

The network works on an input tensor of shape [(M+1) x L x N], where M is the number of stocks we select, L is the length of the window and N is the number of features. We add a 1 to M to represent liquid cash that we initially start with.

The assets under consideration are liquid, hence they can be converted into cash quickly, with little or no loss in value. Moreover, the selected assets have available historical data in order to enable analysis.

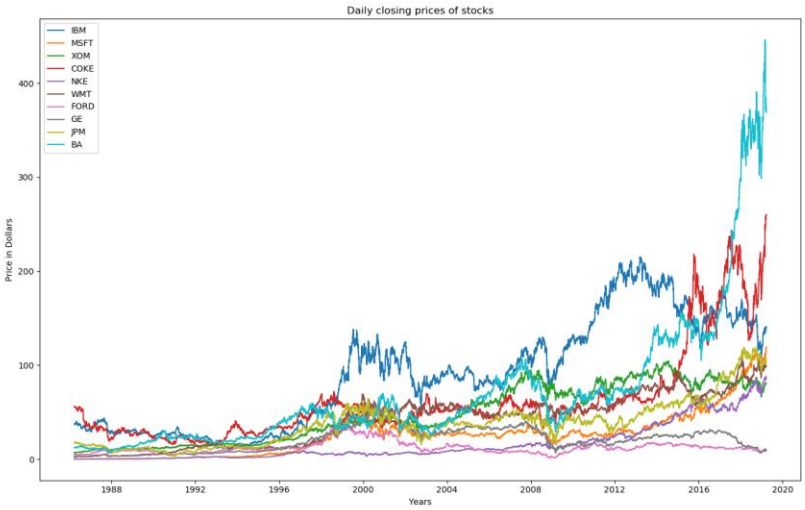


Figure 10: Daily Stock Closing Prices for 10 stocks

7.2. Action

The action represents the weights of the stocks in a portfolio at any given time (*t*-1). It is represented as $\boldsymbol{a}_{t-1}= (a_{0,t}, 1, a_{1,t}, 1, \dots, a_{m,t}, 1)^{T}$. It is subject to the constraint that the sum of all the portfolio weights should be 1.

7.3. Reward Function

The immediate reward r_t, at time (*t* -1) is given by:

$$r_t(s_{t-1}, a_{t-1}) = \log(\mathbf{a_{t-1}} \cdot \mathbf{y_{t-1}} - \mu \sum_{i=1}^{m} |a_{i,t-1} - w_{i,t-1}|) \qquad (7.1)$$

where,

s_{t-1} is the state of the environment which means it reflects the price tensor for the M assets of window length L and N features.

a_{t-1} is the action taken by the agent at time (*t*-1), it reflects the portfolio weights vector.

y_{t-1}, is the closing price change vector represented by

$\mathrm{y}_t = \frac{\mathbf{v}_t}{\mathbf{v}_{t-1}} = (1, \frac{v_{1,t}}{v_{1,t-1}}, \dots, \frac{v_{m,t}}{v_{m,t-1}})^{T}$. It represents the fluctuation in the closing price of the selected stocks.

μ, represents the transaction cost which we have set to 0.0025.

$\mu \sum_{i=1}^{m} |a_{i,t-1} - w_{i,t-1}|$ represents the transaction cost involved in changing the portfolio weights.

7.4. Objective Function

The objective function is an accumulative portfolio value given by,

$$P_T = \prod_{t=1}^{T} P_0 r_t \qquad (7.2)$$

where, the reward r_t, is given above and the P_0 is the initial investment value which we take as $10,000. Our agent wants to maximize this objective function.

7.5. Network Structure

Our network structure is motivated by Jiang et. al., where they use the concept of Identical Independent Evaluators (IIE). IIE means that the networks flow independently for the assets while network parameters are shared among these streams. The network evaluates one stock at a time and evaluates its preference to invest in this stock. The total (**M+1**) stocks are then normalized by the Softmax function and compressed into a weight vector as the next periods action.

We will be working with Deep Feed Forward Neural Networks and Convolutional Neural Networks, which will act as our portfolio management agents.

7.5.1. Deep Feed Forward Neural Network

The Deep Feed Forward Neural Network (DFNN) agent comprises of the input layer, 2 hidden layers and the output layer. The input layer flattens the input tensor into the shape of M * L * N.

The first hidden layer has the weights in the shape of (M * L * N, 784) and uses RELU as an activation function.

The second hidden layer has the weights in the shape of (784, 500) and uses RELU as the activation function. The second hidden layer feeds the results into an output layer which has the shape of (500, M) and uses Softmax activation to output the action.
We us the output of the last layer to calculate the portfolio value. We take the log of the portfolio value and try to maximize the value which also becomes the loss function to continue stochastic gradient descent. We use Adam as the algorithm for calculating gradient descent.

Input	[-1 * (M+1) x L x N]	
Hidden Layer 1	[-1 * 784]	Weights = [(M+1) x L x N, 784] bias = [784]
Activation Function	ReLU	
Hidden Layer 2	[-1 * 500]	Weights = [784, 500] bias = [500]
Activation Function (Conv1)	ReLU	
Dense	[-1 * (M+1)]	Weights = [500, (M+1)] bias = [(M+1)]
Activation Function (Conv2)	Softmax	

Table 1: Deep Feed Forward Neural Network Architecture

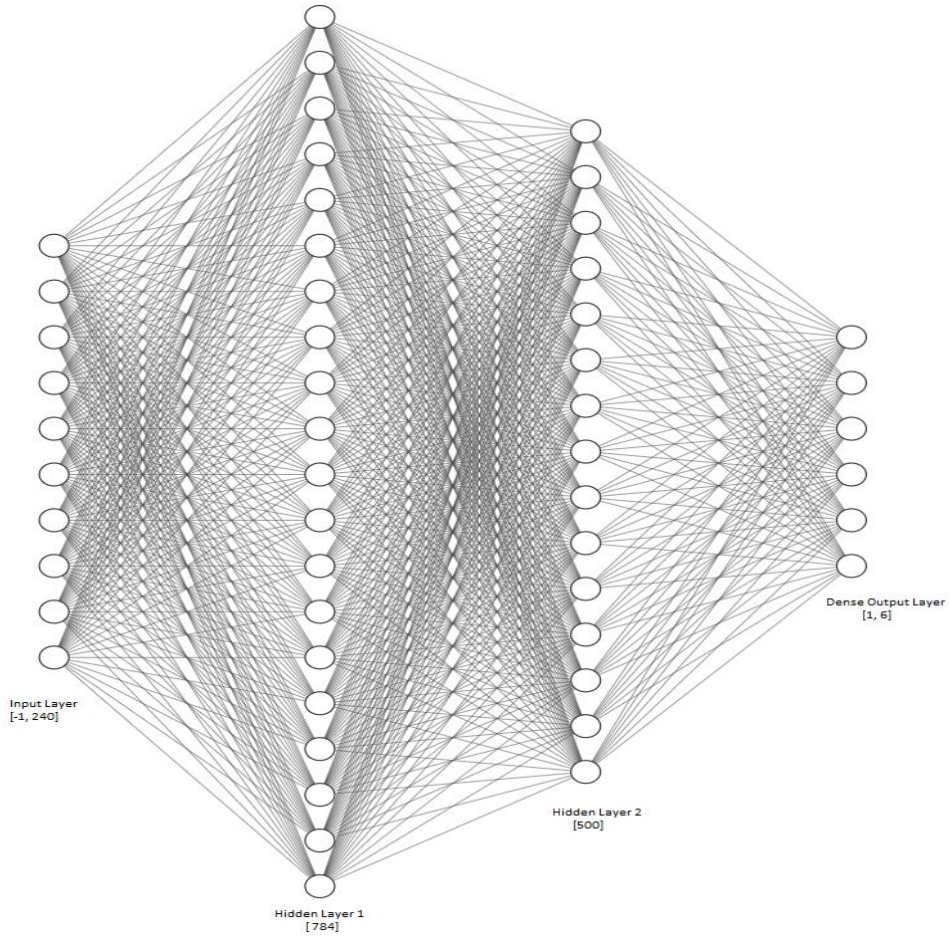


Figure 11: Deep Feed Forward Neural Network

7.5.2. Deep Convolutional Neural Network

The Deep Convolutional Neural Network (DCNN) agent comprises of the input layer, 3 convolution layers, and a dense layer which generates the output. The input layer works with an input of shape [M, L, N]. The first convolution layer works with 2 convolution filters of shape [1, 2] and a stride of size 1. We do not use any padding and use the RELU activation function.

The next convolution layer takes the output of the first layer and uses it as input. We use 48 convolution filters. The width of the output of the previous layer is used to define the shape of the convolution filter which is [1, width]. We take a stride of size 1. We do not use any padding and use the RELU activation function. We also perform L2 regularization in this layer.

Before the third convolution, we concatenate previous portfolio weights with the output of the second convolution layer and use it as our input. We use 1 convolution filter in this layer with the shape [1, width]. We do not use any padding and use RELU activation function. We also perform L2 regularization in this layer as well.

The output the previous convolution layer is flattened and sent to a dense layer which use softmax activation to produce M probabilities as the final output.

We us the output of the last layer to calculate the portfolio value. We take the log of the portfolio value and try to maximize the value which also becomes the loss function to continue stochastic gradient descent. We use Adam as the algorithm for calculating gradient descent.

Input	[-1 * (M+1) * L * N]	
Conv1	[-1 * (M+1) * (L-1) * 2]	Filter = 2, Kernel =[1,2], stride=1, padding='valid'
Batch Normalization		
Activation Function (Conv1)	ReLU	
Conv2	[-1 * (M+1) * 1 * 48]	Filter = 48, Kernel =[1, L-1], Stride=1, Padding='valid', Regularizer = L2
Activation Function (Conv2)	ReLU	
Concatenation	[-1 * (M+1) * 1 * 49]	[-1 * M * 1 * 1]
Conv3	[-1 * (M+1) * 1 * 1]	Filter = 1, Kernel =[1, 1], Stride=1, Padding='valid', Regularizer = L2
Activation Function (Conv3)	ReLU	
Flatten	[-1* (M+1)]	
Dense	[-1* (M+1)]	
Activation Function (Dense)	Softmax	

Table 2: Deep Convolutional Neural Network Architecture

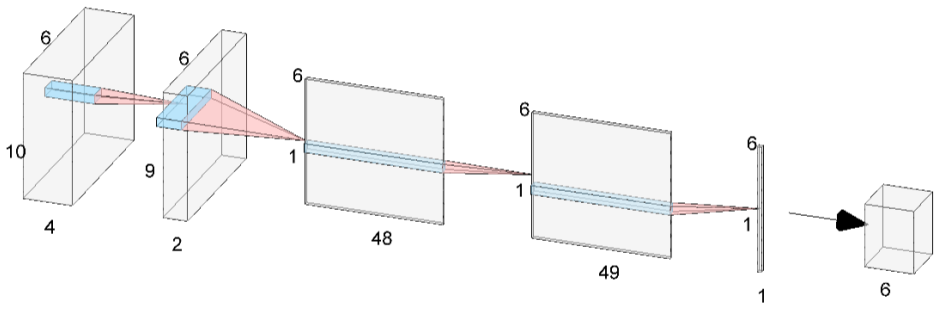


Figure 12:Deep Convolutional Neural Network

Chapter 8

RESULTS

We performed a number of experiments involving different set of network architectures, learning rates and features of data.

8.1. Learning Rate

Learning Rate plays an important role in neural network training. However, learning rate changes can have a big impact on the training speed of the network. A high learning rate will make training loss decrease fast but may drop the learning into local minimum instead of finding the global minimum, or it may cause the gradient to fluctuate around the global minimum instead of letting it reach the actual global minimum. A low learning rate on the other hand will make the training loss decrease very slowly even after a large number of epochs. Only an optimum learning rate will help the network achieve the satisfactory results.

In order to make sure our networks learn the optimum way, we play around with different learning rates and notice the results.

Algorithm	PG		DDPG			
Network Architecture	Feed Forward Neural Network	Convolutional Neural Network	Feed Forward Neural Network		Convolutional Neural Network	
			Actor	Critic	Actor	Critic
Optimizer	Adam	Adam	Adam	Adam	Adam	Adam
Learning Rates	0.1	0.1	0.01	0.001	0.01	0.001
	0.01	0.01	0.01	0.0001	0.01	0.0001
	0.001	0.001	0.01	1e-5	0.01	1e-5
	0.0001	0.0001	0.001	0.001	0.001	0.001
	-	-	0.001	0.0001	0.001	0.0001
	-	-	0.001	1e-5	0.001	1e-5

Table 3: Different Learning Rates applied to networks

8.1.1. Feed Forward Neural Network for Policy Gradients

Loss and Reward values in training with respect to different learning rates for 100 epochs

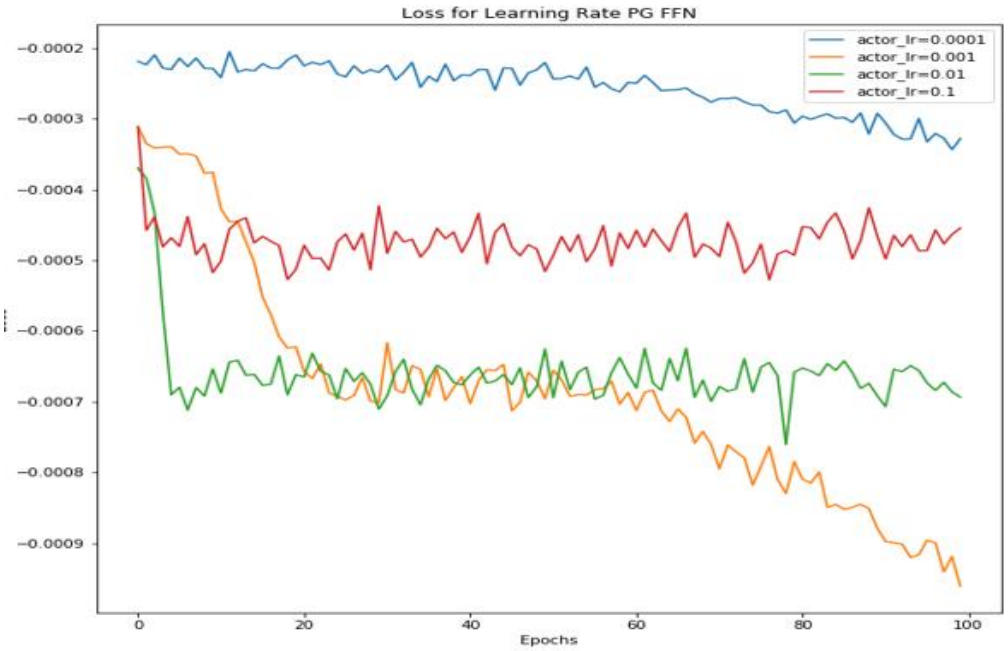


Figure 13: Loss vs Epochs for Policy Gradient Feed Forward Neural Network

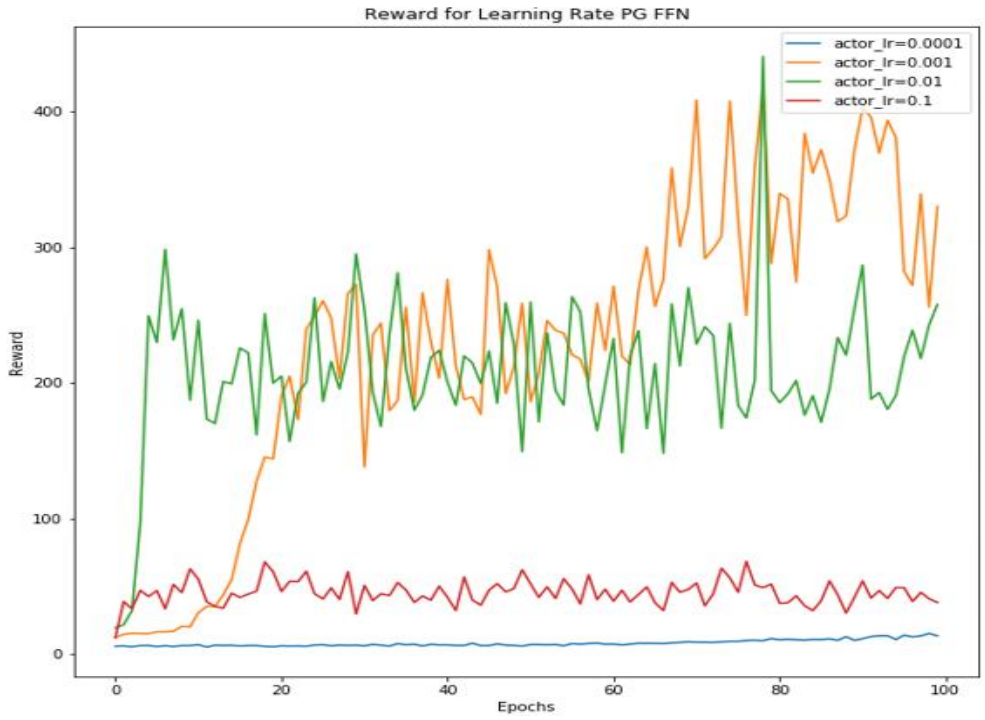


Figure 14: Reward vs Epochs for Policy Gradient Feed Forward Neural Network

8.1.2. Convolutional Neural Network for Policy Gradients

Loss and Reward values in training with respect to different learning rates for 100 epochs

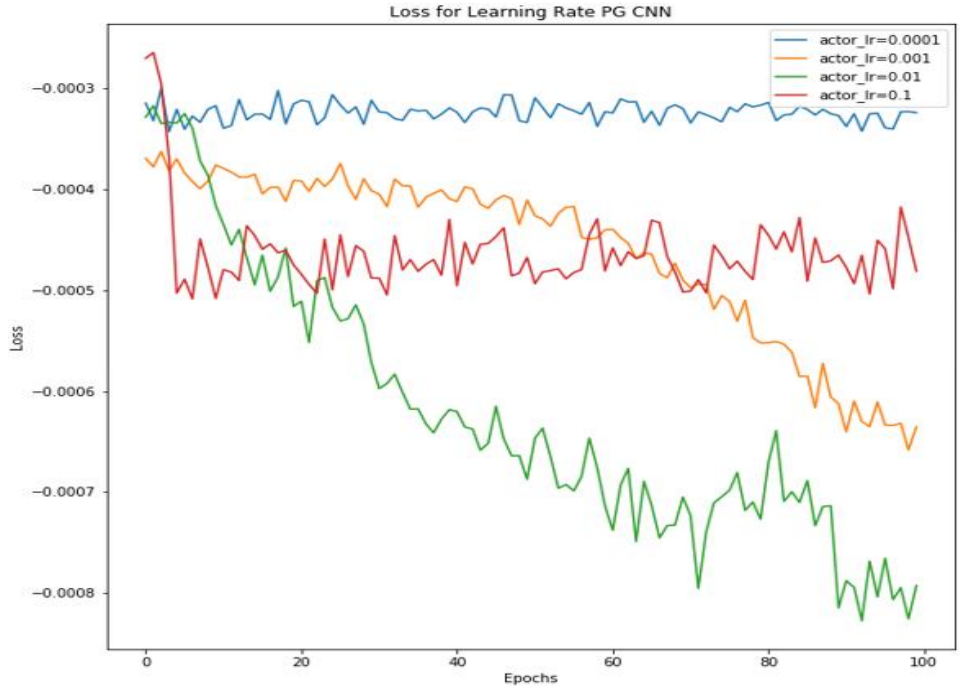


Figure 15: Loss vs Epochs for Policy Gradient Convolutional Neural Network

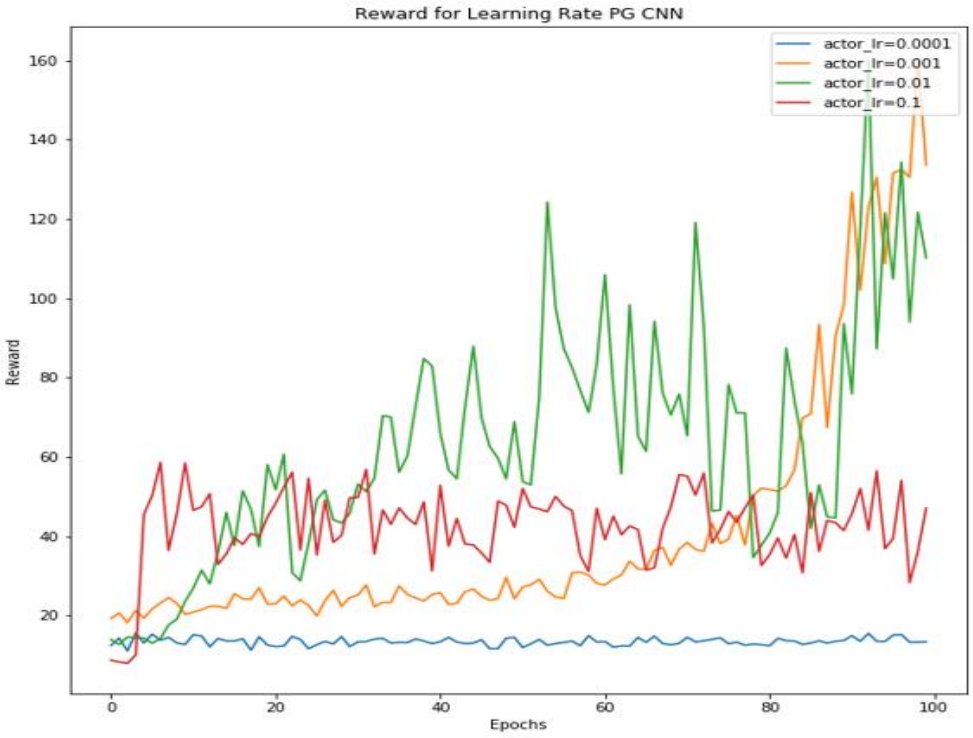


Figure 16: Reward vs Epochs for Policy Gradient Convolutional Neural Network

8.1.3. Wealth created by FFN agent trained over 100 epochs using different learning rates on test data

8.1.3.1. Learning Rate = 0.1

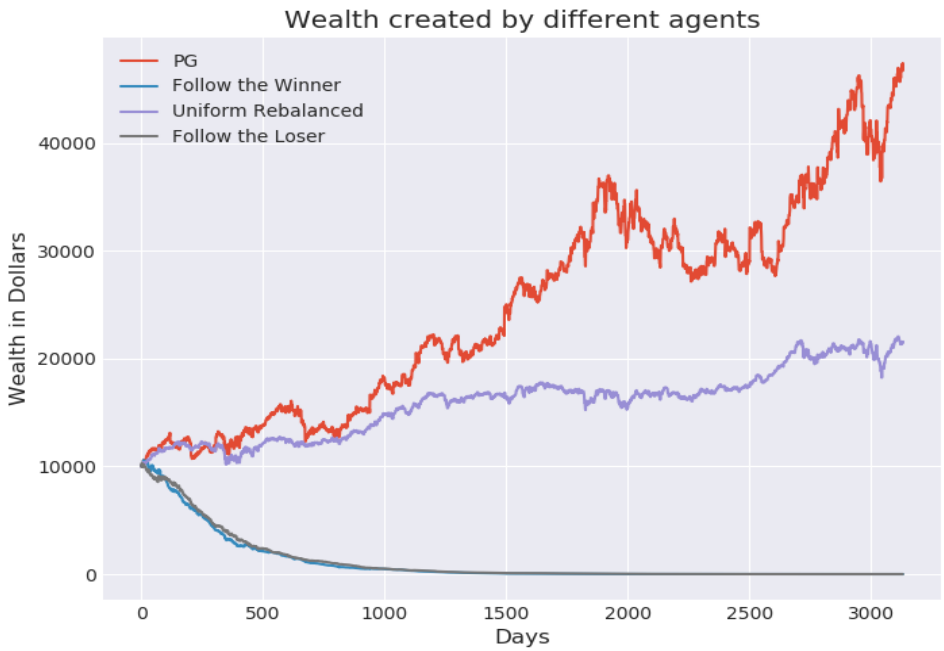


Figure 17: Wealth created by FFN for PG LR = 0.1

Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.303	-2.35
Follow the Loser	-0.283	-2.2
Uniform Rebalanced	0.059	0.488
Policy Gradient - FFN	0.089	0.651

Table 4: Technical Indicators for FFN PG LR = 0.1

8.1.3.2. Learning Rate = 0.01

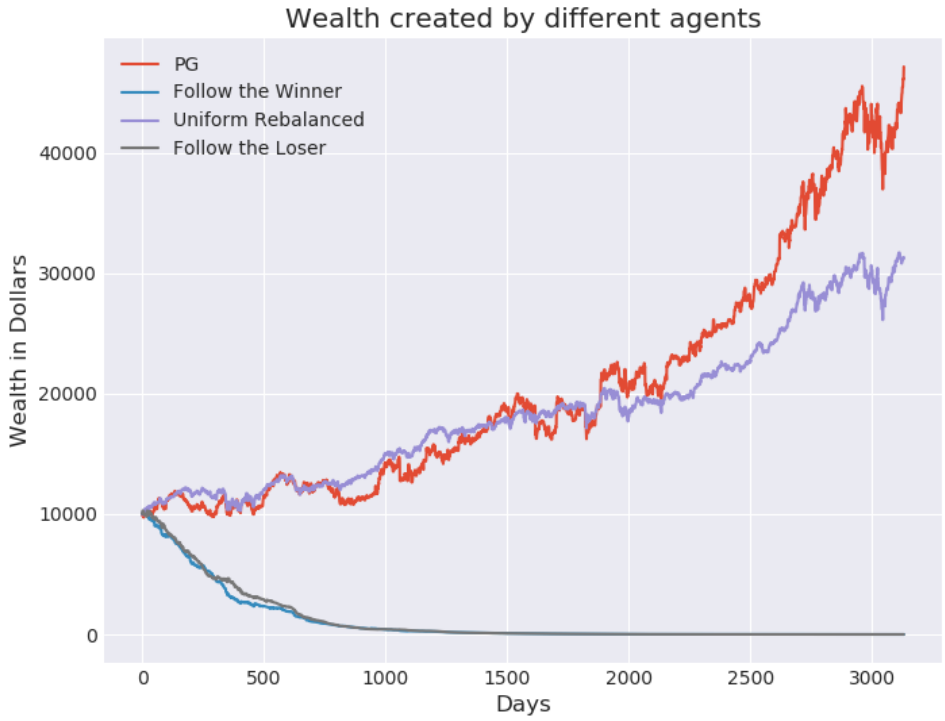


Figure 18: Wealth created by FFN for PG LR = 0.01

Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.276	-2.128
Follow the Loser	-0.295	-2.33
Uniform Rebalanced	0.071	0.584
Policy Gradient - FFN	0.089	0.654

Table 5: Technical Indicators for FFN PG LR = 0.01

8.1.3.3. Learning Rate = 0.001

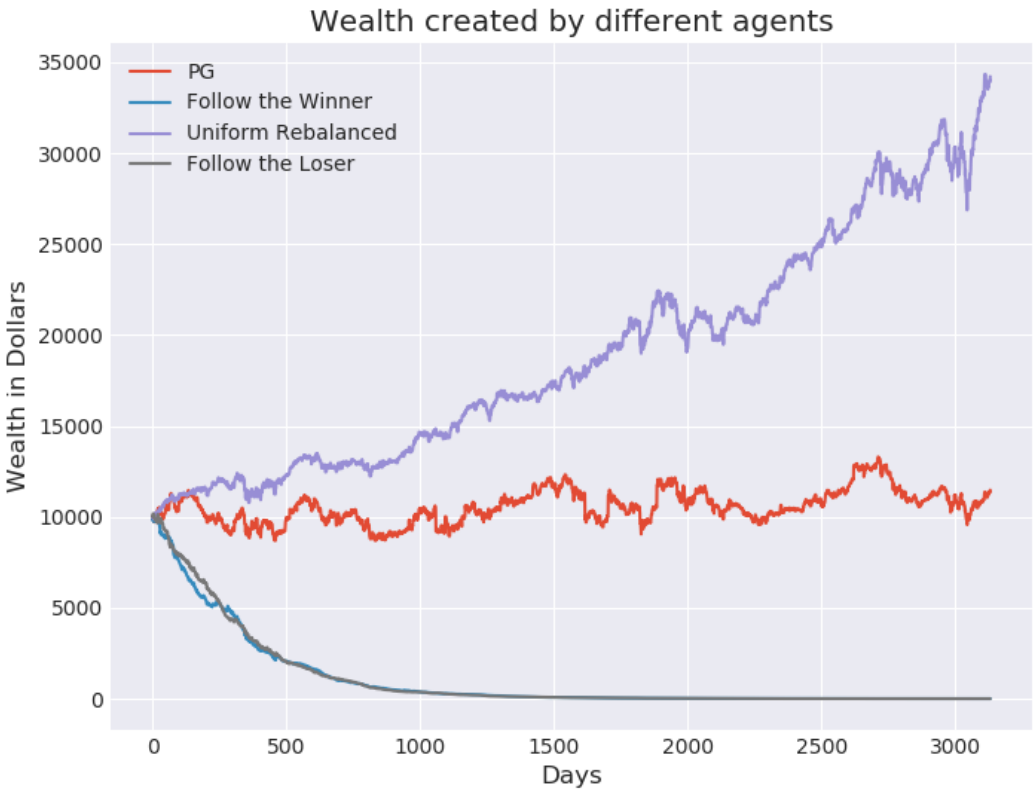


Figure 19: Wealth created by FFN for PG LR = 0.001

Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.274	-2.009
Follow the Loser	-0.302	-2.314
Uniform Rebalanced	0.074	0.609
Policy Gradient - FFN	0.042	0.321

Table 6: Technical Indicators for FFN PG LR = 0.001

8.1.3.4. Learning Rate = 0.0001

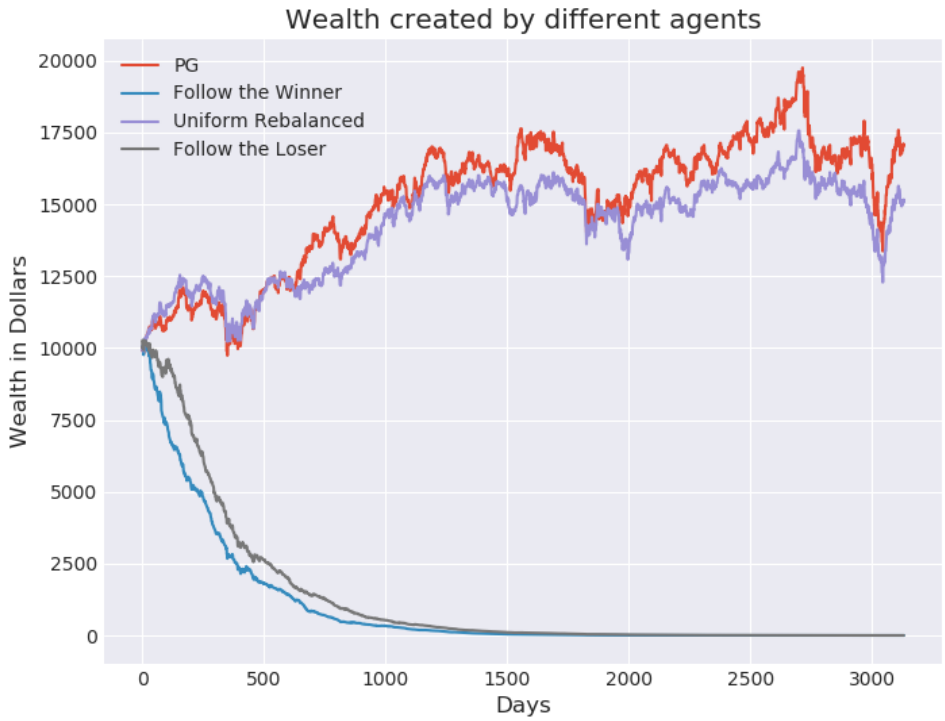


Figure 20: Wealth created by FFN for PG LR = 0.0001

Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.317	-2.438
Follow the Loser	-0.28	-2.171
Uniform Rebalanced	0.048	0.395
Policy Gradient - FFN	0.052	0.424

Table 7: Technical Indicators for FFN PG LR = 0.0001

8.1.4. Wealth created by CNN agent trained over 100 epochs using different learning rates on test data

8.1.4.1. Learning Rate = 0.1

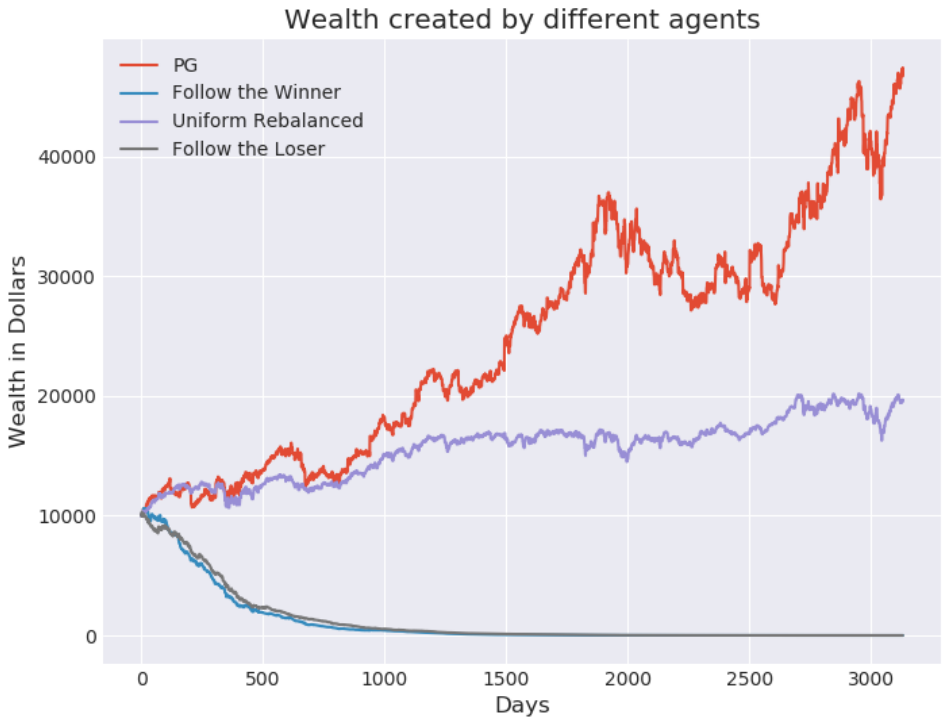


Figure 20: Wealth created by CNN for PG LR = 0.1

Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.298	-2.291
Follow the Loser	-0.284	-2.196
Uniform Rebalanced	0.056	0.463
Policy Gradient - CNN	0.089	0.651

Table 8: Technical Indicators for CNN PG LR = 0.1

8.1.4.2. Learning Rate = 0.01

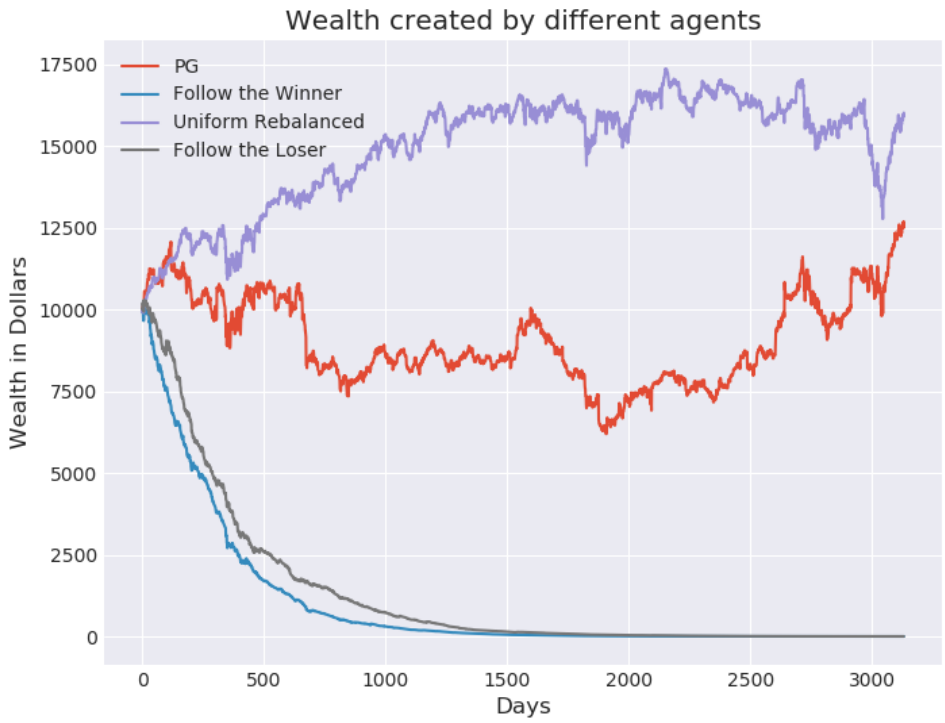


Figure 21: Wealth created by CNN for PG LR = 0.01

Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.305	-2.36
Follow the Loser	-0.271	-2.13
Uniform Rebalanced	0.049	0.41
Policy Gradient - CNN	0.045	0.345

Table 9: Technical Indicators for CNN PG LR = 0.01

8.1.4.3. Learning Rate = 0.001

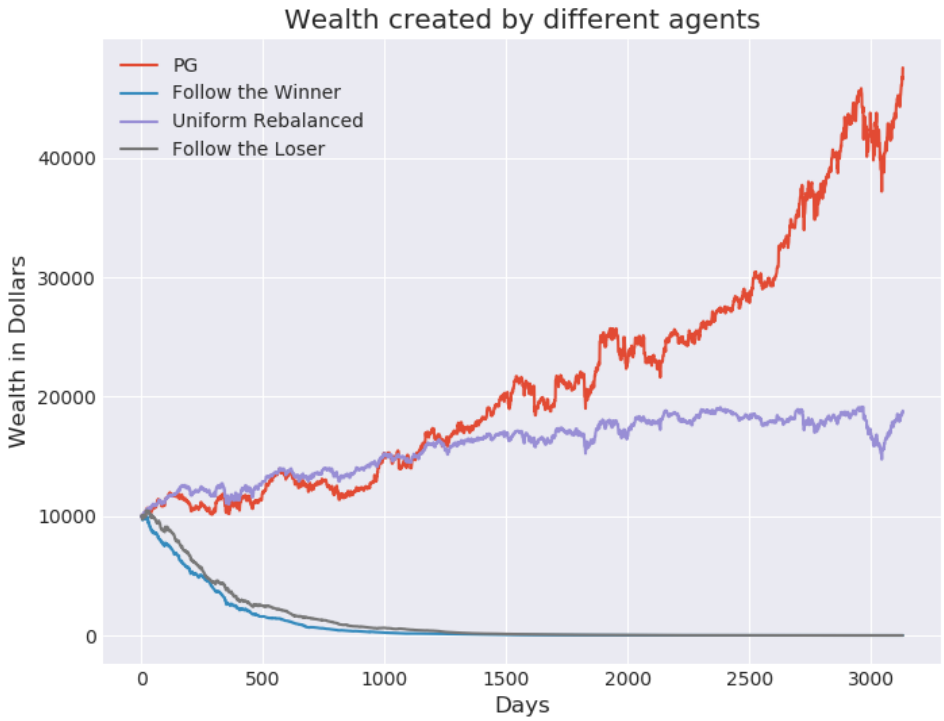


Figure 21: Wealth created by CNN for PG LR = 0.001

Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.295	-2.236
Follow the Loser	-0.278	-2.166
Uniform Rebalanced	0.055	0.452
Policy Gradient - CNN	0.087	0.668

Table 10: Technical Indicators for CNN PG LR = 0.001

8.1.4.4. Learning Rate = 0.0001

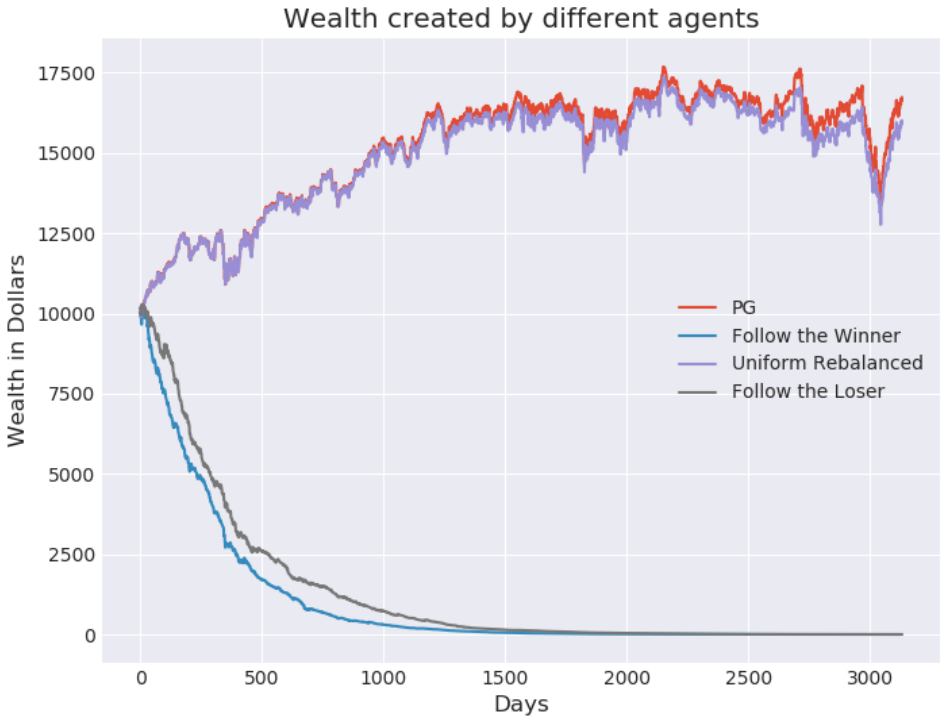


Figure 21: Wealth created by CNN for PG LR = 0.0001

Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.305	-2.36
Follow the Loser	-0.271	-2.13
Uniform Rebalanced	0.0549	0.41
Policy Gradient - CNN	0.05	0.422

Table 11: Technical Indicators for CNN PG LR = 0.0001

8.1.5. Feed Forward Neural Network for Deep Deterministic Policy Gradients

Loss vs Reward values in training with respect to different learning rates for 100 epochs

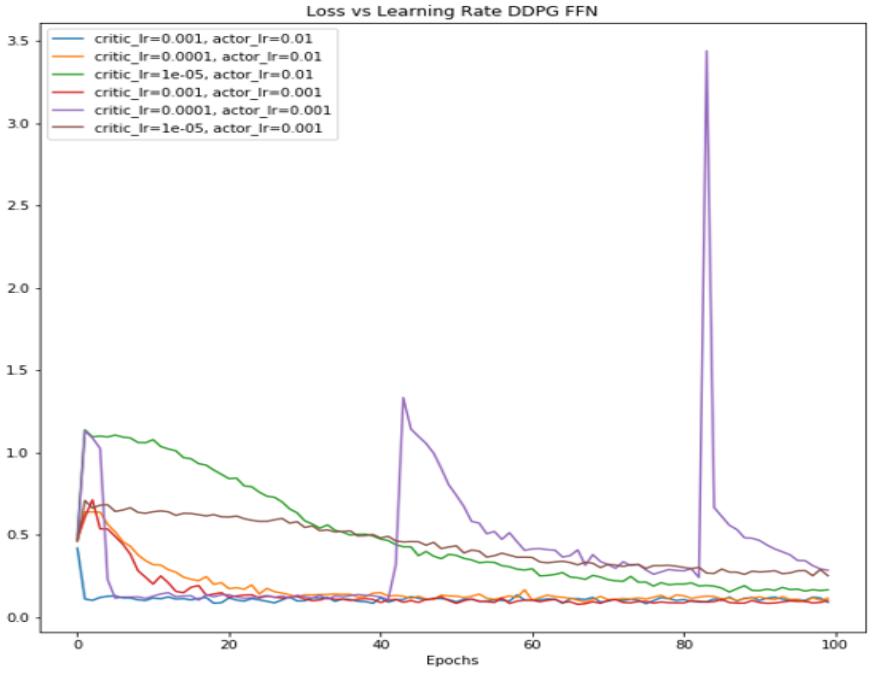


Figure 21: Loss vs Epochs for Deep Deterministic Policy Gradient Feed Forward Neural Network

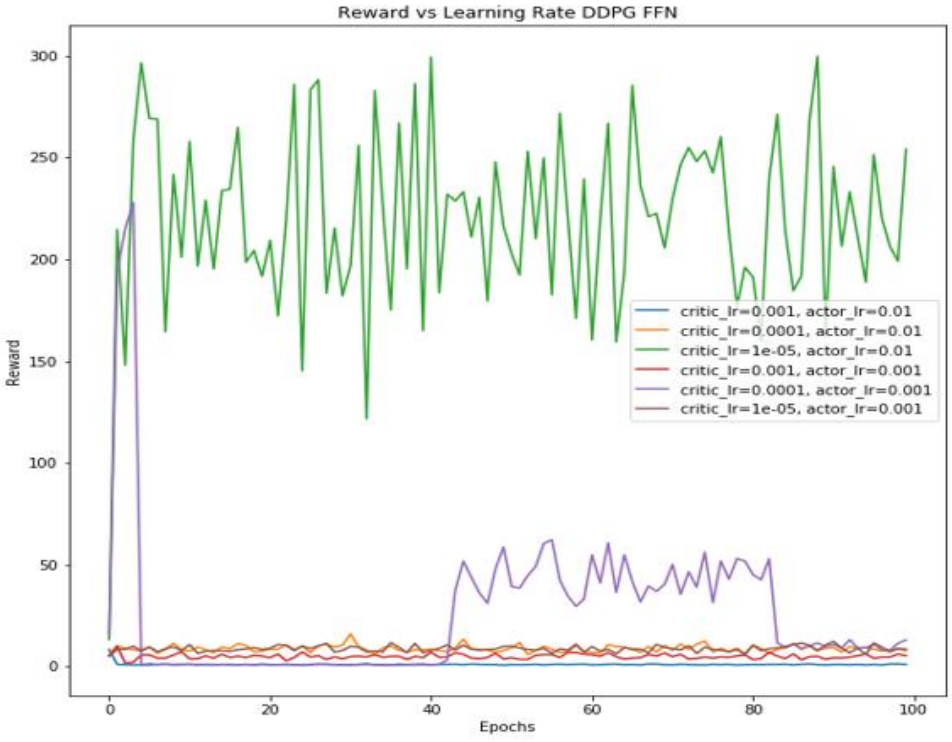


Figure 22: Reward vs Epochs for Deep Deterministic Policy Gradient Feed Forward Neural Network

8.1.6. Convolutional Neural Network for Deep Deterministic Policy Gradients

Loss and Reward values in training with respect to different learning rates for 100 epochs

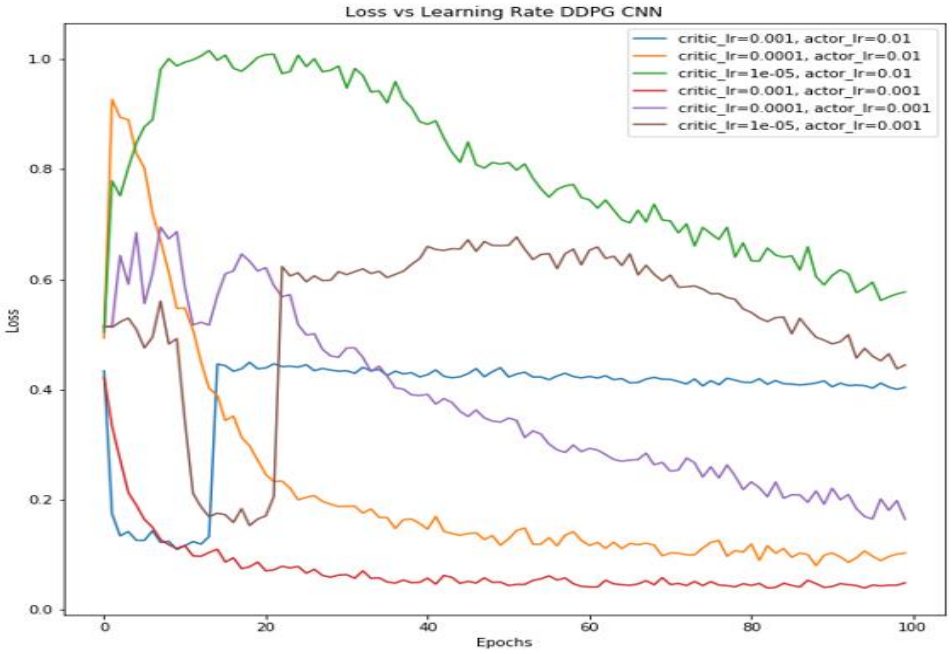


Figure 23: Loss vs Epochs for Deep Deterministic Policy Gradient Convolutional Neural Network

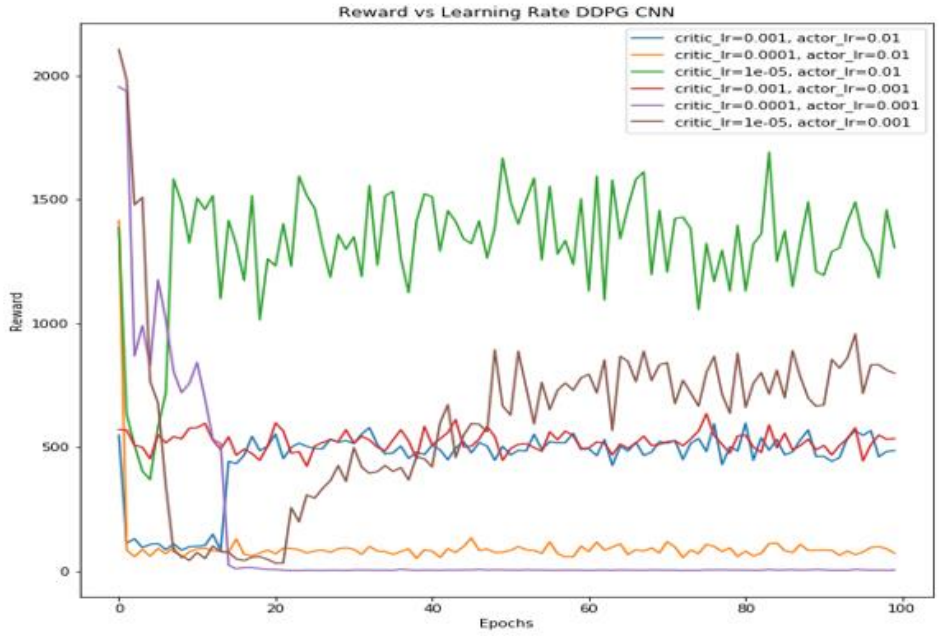


Figure 24: Reward vs Epochs for Deep Deterministic Policy Gradient Convolutional Neural Network

8.1.7. Wealth created by FFN agent trained over 100 epochs using different learning rates on test data

8.1.7.1. Actor Learning Rate=0.01 Critic Learning Rate= 0.001

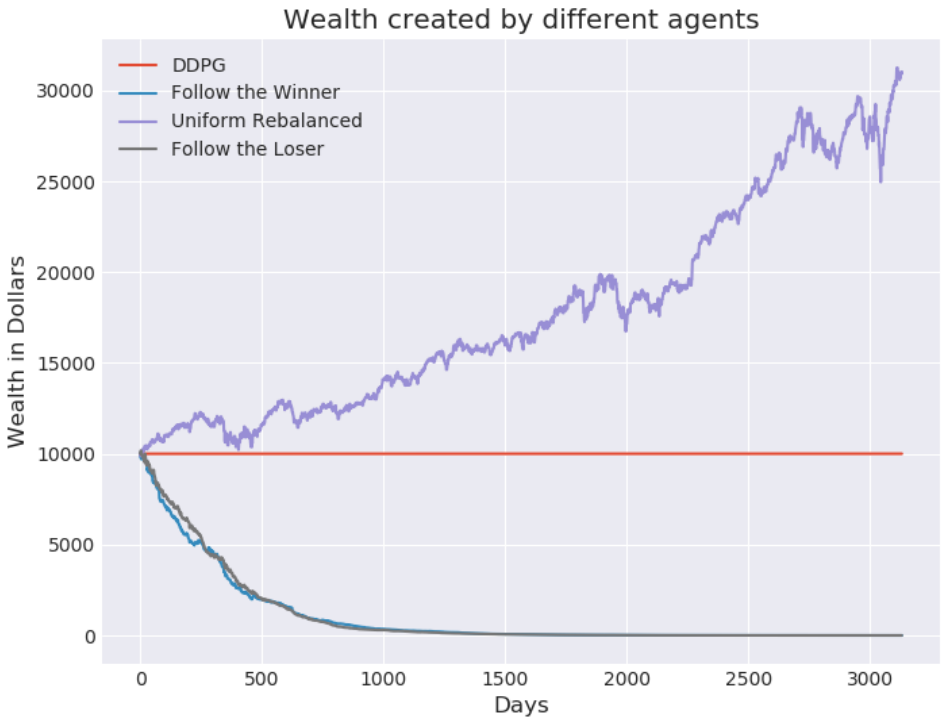


Figure 25: Wealth created by FFN for DDPG ALR = 0.01, CLR = 0.001

Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.286	-2.118
Follow the Loser	-0.304	-2.318
Uniform Rebalanced	0.071	0.582
DDPG - FFN	0.032	0.284

Table 12: Technical Indicators FFN for DDPG ALR = 0.01, CLR = 0.001

8.1.7.2. Actor Learning Rate = 0.01 Critic Learning Rate = 0.0001

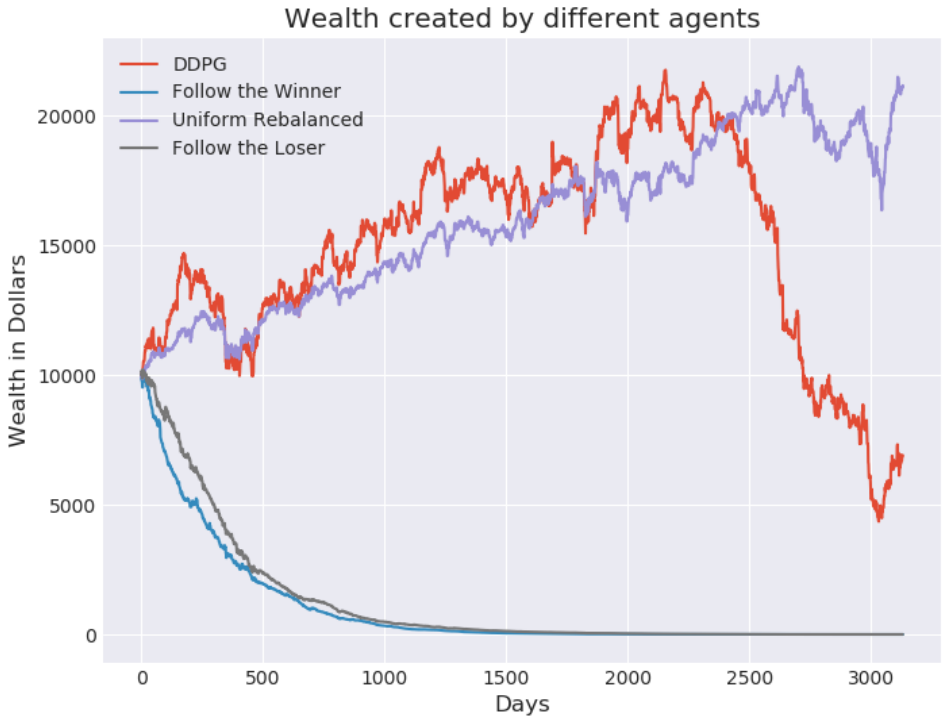


Figure 26: Wealth created by FFN for DDPG ALR = 0.01, CLR = 0.0001

Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.3	-2.207
Follow the Loser	-0.287	-2.188
Uniform Rebalanced	0.058	0.484
DDPG - FFN	0.029	0.205

Table 13: Technical Indicators FFN for DDPG ALR = 0.01, CLR = 0.001

8.1.7.3. Actor Learning Rate = 0.01, Critic Learning Rate = 1e-05

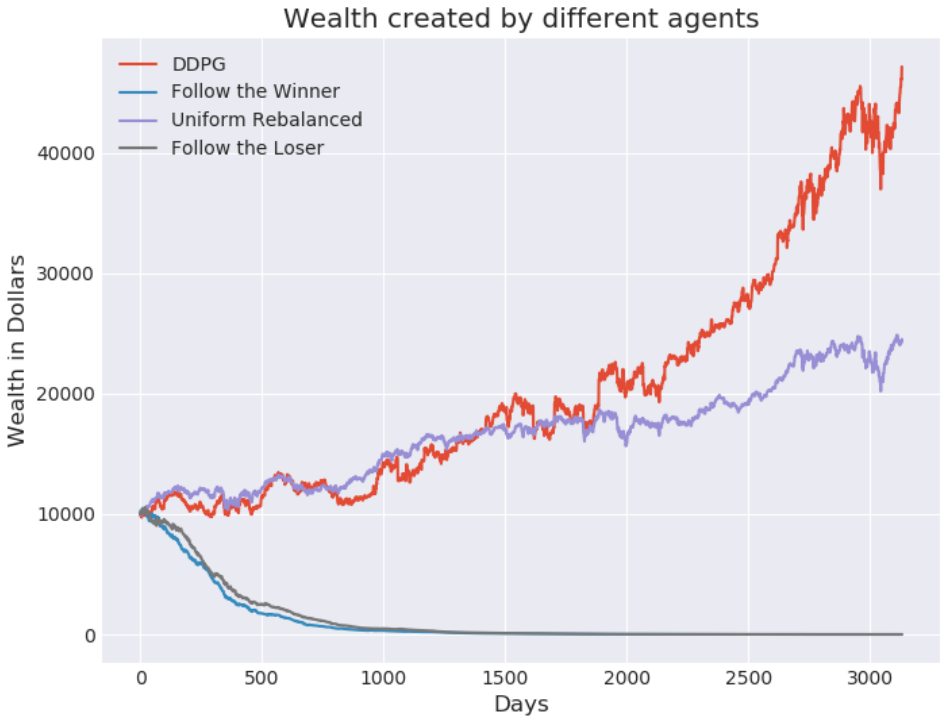


Figure 27: Wealth created by FFN for DDPG ALR = 0.01, CLR = 1e-5

Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.288	-2.192
Follow the Loser	-0.284	-2.195
Uniform Rebalanced	0.063	0.52
DDPG - FFN	0.089	0.654

Table 14: Technical Indicators FFN for DDPG ALR = 0.01, CLR = 1e-5

8.1.7.4. Actor Learning Rate = 0.001
Critic Learning Rate = 0.001

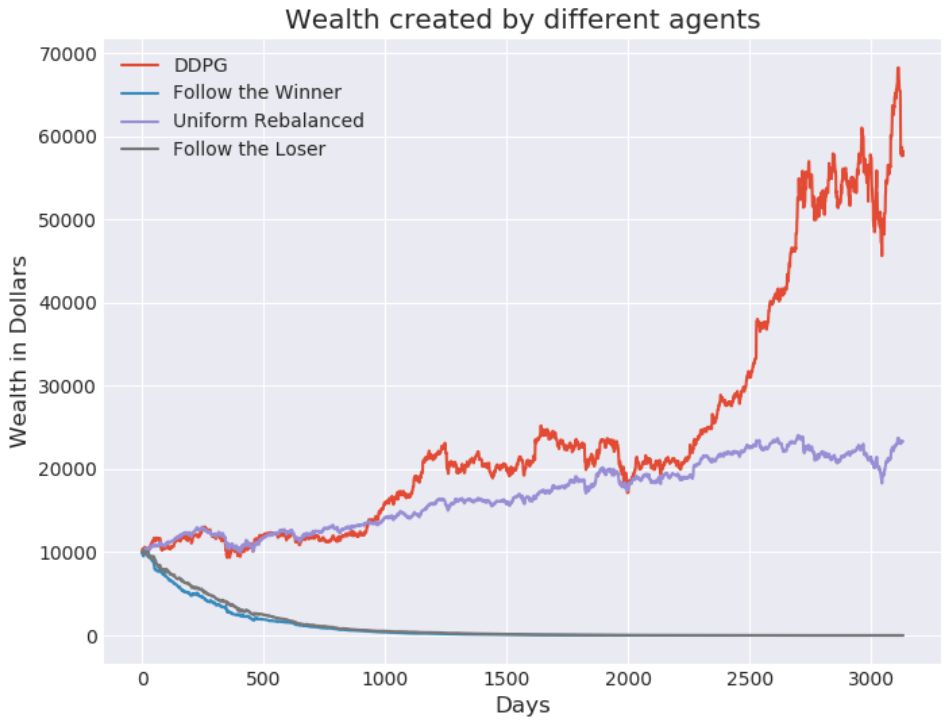


Figure 28: Wealth created by FFN for DDPG ALR = 0.001, CLR = 0.001

Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.297	-2.156
Follow the Loser	-0.286	-2.145
Uniform Rebalanced	0.062	0.507
DDPG - FFN	0.096	0.697

Table 15: Technical Indicators FFN for DDPG ALR = 0.001, CLR = 0.001

8.1.7.5. Actor Learning Rate = 0.001
Critic Learning Rate = 0.0001

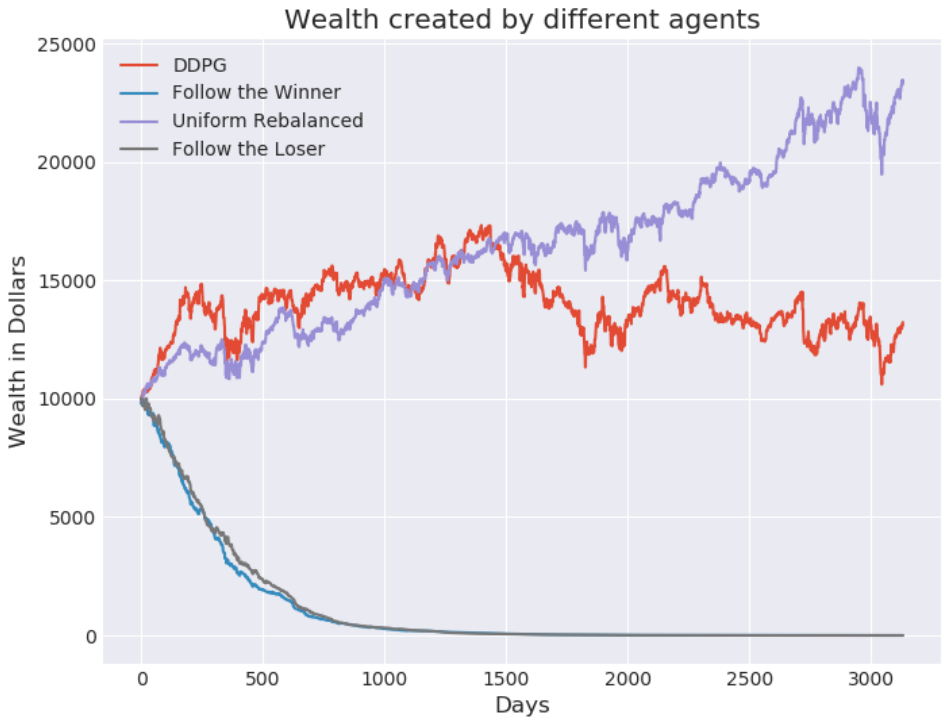


Figure 29: Wealth created by FFN for DDPG ALR = 0.001, CLR = 0.0001

Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.295	-2.268
Follow the Loser	-0.31	-2.456
Uniform Rebalanced	0.062	0.51
DDPG - FFN	0.046	0.356

Table 16: Technical Indicators FFN for DDPG ALR = 0.001, CLR = 0.0001

8.1.7.6. Actor Learning Rate = 0.001 Critic Learning Rate = 1e-5

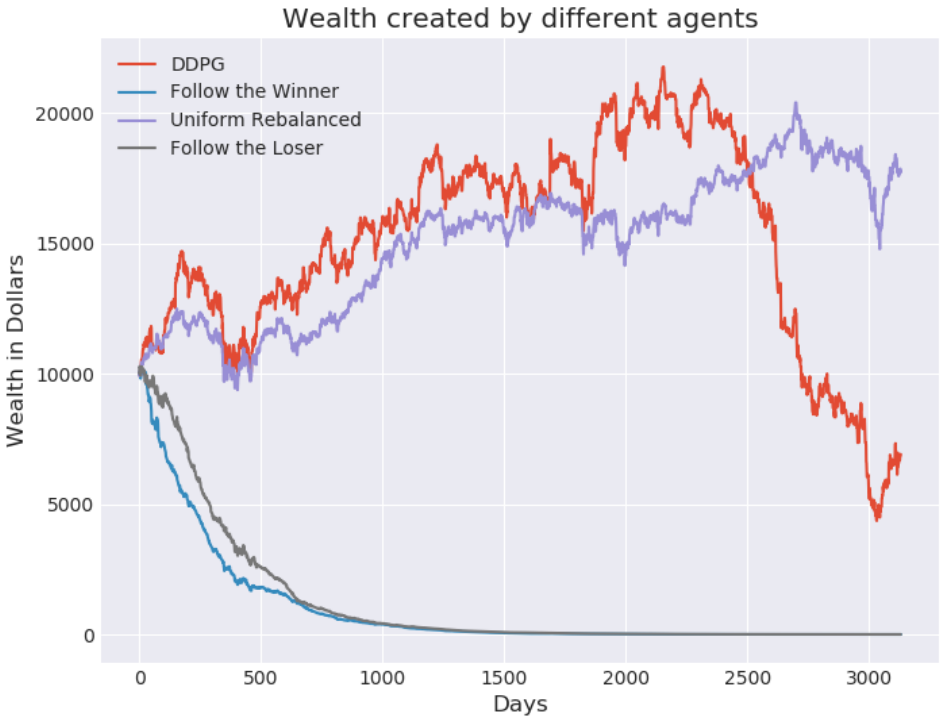


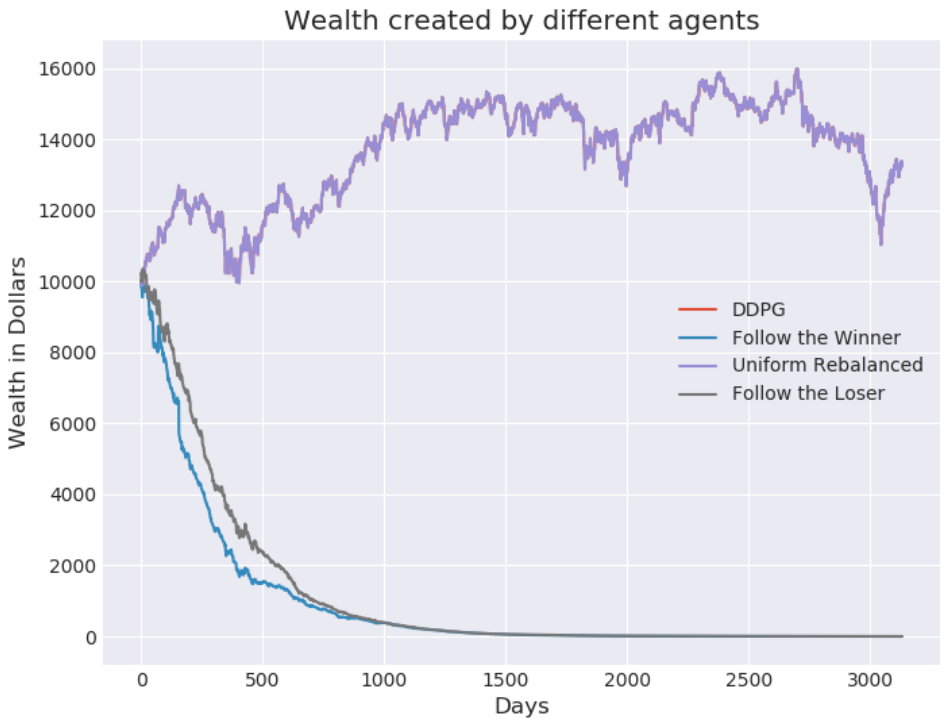
Figure 30: Wealth created by FFN for DDPG ALR = 0.001, CLR = 1e-5

Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.31	-2.359
Follow the Loser	-0.286	-2.196
Uniform Rebalanced	0.053	0.435
DDPG - FFN	0.029	0.205

Table 17: Technical Indicators FFN for DDPG ALR = 0.001, CLR = 1e-5

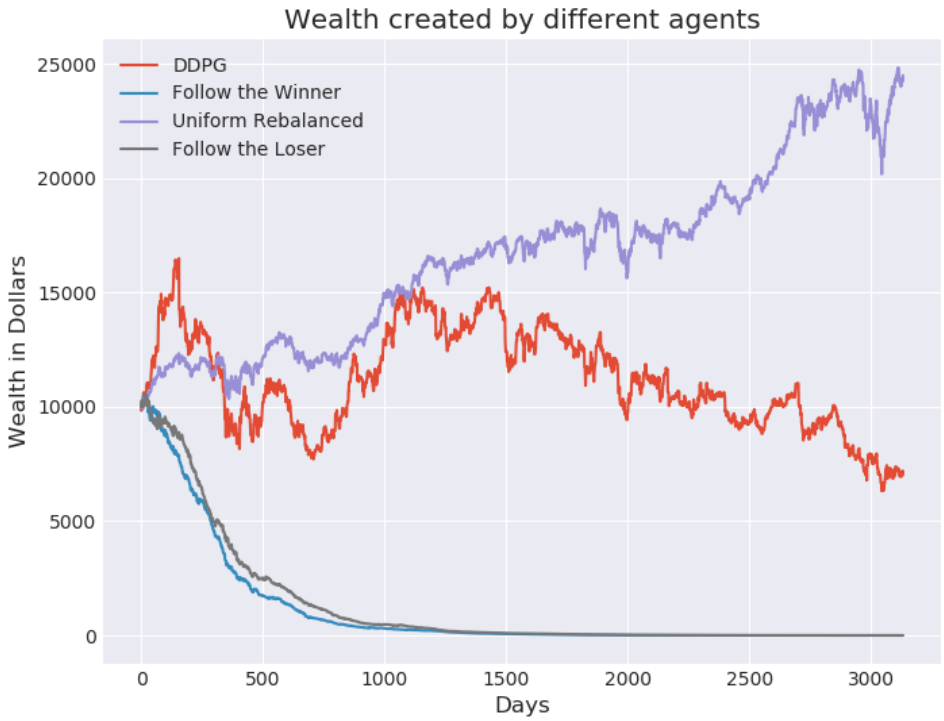
8.1.8. Wealth created by CNN agent trained over 100 epochs using different learning rates on test data

8.1.8.1. Actor Learning Rate = 0.01
Critic Learning Rate = 0.001



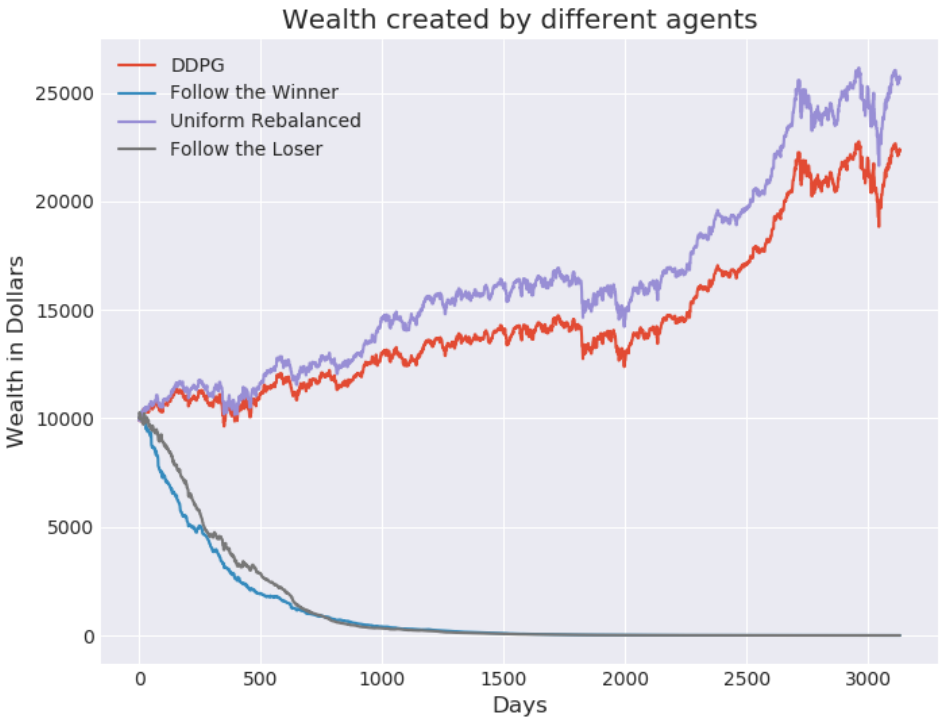
Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.311	-2.363
Follow the Loser	-0.299	-2.32
Uniform Rebalanced	0.044	0.36
DDPG - CNN	0.044	0.36

8.1.8.2. Actor Learning Rate = 0.01
Critic Learning Rate = 0.0001



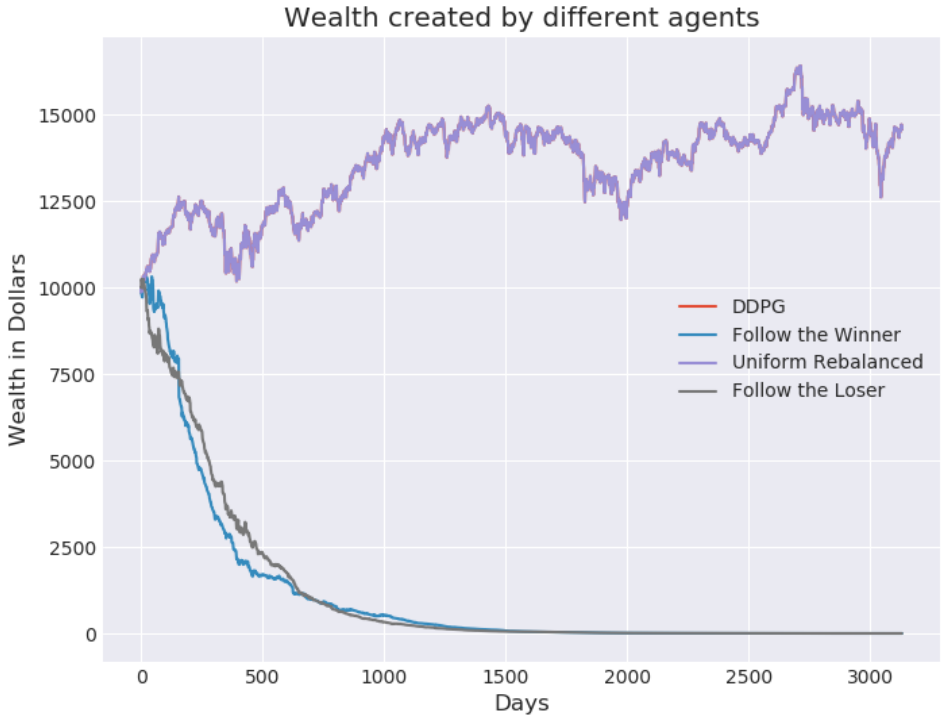
Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.288	-2.192
Follow the Loser	-0.284	-2.2
Uniform Rebalanced	0.063	0.52
DDPG - CNN	0.031	0.218

8.1.8.3. Actor Learning Rate = 0.01
Critic Learning Rate = 1e-05



Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.286	-2.242
Follow the Loser	-0.293	-2.29
Uniform Rebalanced	0.064	0.534
DDPG - CNN	0.06	0.497

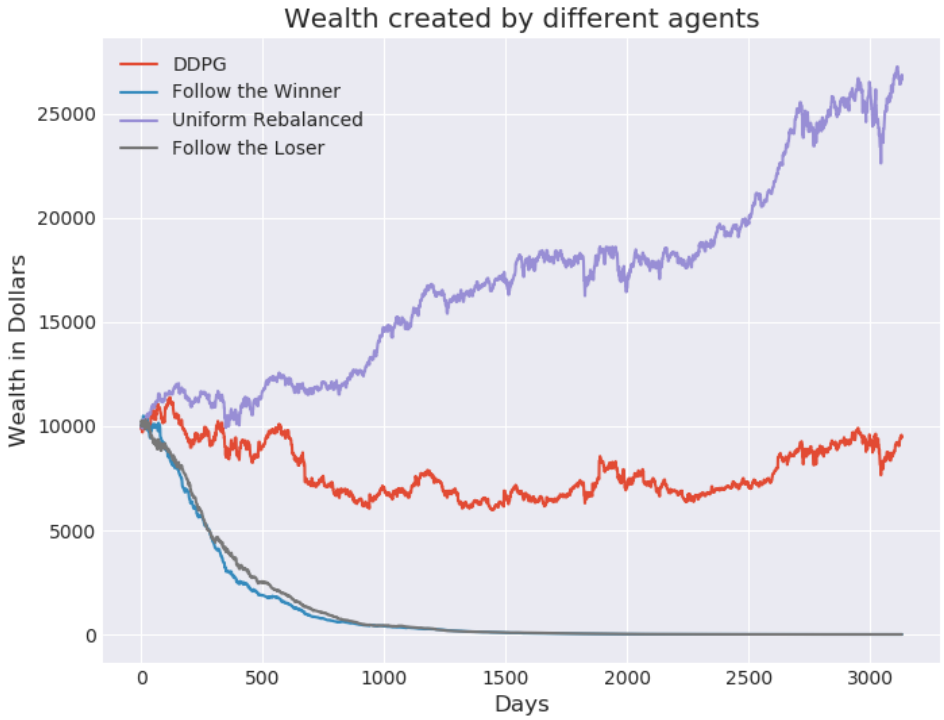
8.1.8.4. Actor Learning Rate = 0.001
Critic Learning Rate = 0.001



Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.311	-2.401
Follow the Loser	-0.307	-2.39
Uniform Rebalanced	0.046	0.386
DDPG - CNN	0.046	0.385

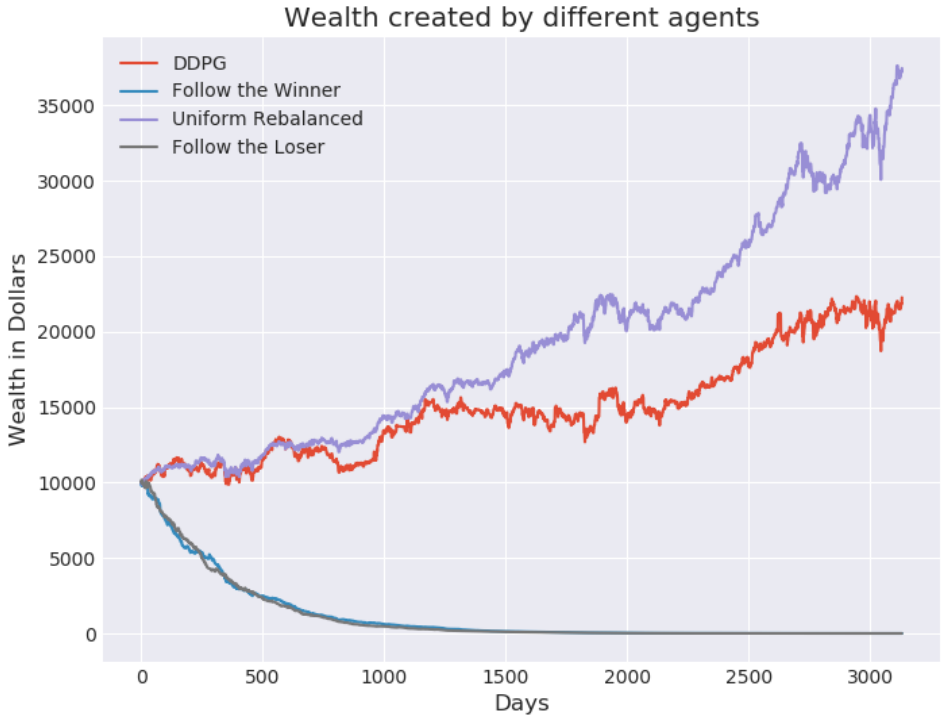
8.1.8.5. Actor Learning Rate = 0.001

Critic Learning Rate = 0.0001



Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.283	-2.181
Follow the Loser	-0.282	-2.18
Uniform Rebalanced	0.066	0.546
DDPG - CNN	0.037	0.276

8.1.8.6. Actor Learning Rate = 0.001
Critic Learning Rate = 1e-05



Strategy	Average Daily Return	Sharp Ratio
Follow the Winner	-0.257	-1.908
Follow the Loser	-0.294	-2.28
Uniform Rebalanced	0.076	0.635
DDPG - CNN	0.062	0.486

8.2. Number of Features

We experiment with different combinations of features to understand which ones work the best for the agent to make informed decisions. We test the following set of combinations:

- Only Closing Price
- Closing and High Price
- Closing and Open Price
- Closing and Low Price
- All Prices

- All Prices and Volume

We find that the Closing and High Price combination gives us the best results.

8.3. Number of Epochs

Experiments were run for different number of epochs. We found the best results for 100 epochs. For more number of epochs, the algorithms did well on the training data but on the actual test scenarios they performed poorly.

Chapter 9

CONCLUSION

This thesis applies Deep Reinforcement Learning (DRL) algorithms in a continuous action space for asset allocation in a portfolio. We compare the performances of model free algorithms namely VPG and DDPG using different Deep Neural Network architectures. Compared with other works of portfolio management using RL, we test our agents with accumulative portfolio value as the objective function with different learning rates and different set of feature combinations as inputs. We notice that VPG algorithms outperform the DDPG algorithms, which is a surprise because the DDPG algorithm is a more advanced algorithm. We also notice that VPG algorithm in

combination with Feed Forward Neural Networks as the learning agent works more consistently when compared to other portfolio management strategies. We hypothesize that this may be because of the complexity of the CNN, causing the vanishing gradient problem. We need to perform more experiments to understand this.

We find that DRL can somehow capture the patterns of market movement in a limited set of data and features, while also improving its performance as it keeps on training from experience.

In spite of all the things said above, however, we do not see a stable performance across the board when it comes to the stock market. In some cases, the agents perform very well, they learn the market dynamics and increase the overall wealth but in some cases they perform very poor. They sometimes work worse than uniform distribution of stocks in the portfolio.

We have also found that DRL can perform very well on the gaming environments but its performance is relatively unstable when compared to highly dynamic stock market. One hypothesis that we put forward is the complexity of the reward functions. Gaming environments have simple reward functions when compared to Financial reward functions like the one we have used in this thesis. With a better and simplified reward function, we believe the DRL agents can achieve far greater results. We hope of proving this hypothesis by experimenting further.

Chapter 10

FUTURE WORK

This thesis is in no way a complete or thorough analysis of the different research directions which we can go with, based on the applications of deep reinforcement learning in the portfolio management domain.

In this research, we have not considered the inherent risk associated with the portfolio management process. For future research, we can try to use indicators that measure the risk involved with the way the agent performs asset allocation. Based on the risk appetite of the investing agent, there could be a multitude of changes in the results of the agent.

In this thesis, we have focused on the model free reinforcement learning paradigm while considering the portfolio management problem. The next area of research could be the using model based reinforcement learning algorithms to help the agent learn the optimal way to invest in a set of stocks to increase the overall final reward. In model based reinforcement learning, a model of the dynamics of the environment is used to make predictions.

One more direction could be to adapt a different objective function. We could work with risk adjusted return. We could also try and simplify the objective function to something which is more intuitive like the scoring system in games. With a simplified objective, would make it easier for the agent to achieve better performance compared to a complex one.

This thesis focused on the act of portfolio management solely based on the stock price movements, without trying to understanding the various market factors that help human portfolio managers judge the direction of the market. Factors like economic fluctuations in the market, federal laws, natural and man-made disasters, past history of the company, ethical practices of the executive team and human emotions play a key role when decisions about investment in a company are made. In order to take these into consideration, we can enhance this thesis to add the ability of natural language processing in order to take into account the various social media and news media platforms. The agent will theoretically have the ability to choose the assets to invest in based on the various multitude of factors just mentioned above after analyzing the information it gathers by churning through the news and social media outlets.

REFERENCES

1. Sharpe, W. F. (1994). The sharpe ratio. Journal of portfolio management, 21(1), 49-58.
2. Haykin, S. S., Haykin, S. S., Haykin, S. S., Elektroingenieur, K., & Haykin, S. S. (2009). Neural networks and learning machines (Vol. 3). Upper Saddle River: Pearson education.
3. Hagan, Martin T., et al. Neural network design. Vol. 20. Boston: Pws Pub., 1996.
4. Jain, Anil K., Jianchang Mao, and K. M. Mohiuddin. "Artificial neural networks: A tutorial." Computer 3 (1996): 31-44.
5. Törn, Aimo, and Antanas Žilinskas. Global optimization. Vol. 350. Berlin: Springer-Verlag, 1989.
6. LeCun, Yann, et al. "A theoretical framework for back-propagation." Proceedings of the 1988 connectionist models summer school. Vol. 1. CMU, Pittsburgh, Pa: Morgan Kaufmann, 1988.
7. Bertsekas, Dimitri P., and Athena Scientific. Convex optimization algorithms. Belmont: Athena Scientific, 2015.
8. Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747(2016).
9. LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." nature 521.7553 (2015): 436.
10. Kukačka, J., Golkov, V., & Cremers, D. (2017). Regularization for deep learning: A taxonomy. arXiv preprint arXiv:1710.10686.
11. Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." The Journal of Machine Learning Research 15.1 (2014): 1929-1958.
12. Sutton, Richard S., and Andrew G. Barto. Introduction to reinforcement learning. Vol. 135. Cambridge: MIT press, 1998.
13. Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016.
14. Szepesvári, Csaba. "Algorithms for reinforcement learning." Synthesis lectures on artificial intelligence and machine learning 4.1 (2010): 1-103.
15. Bellman, Richard. "The theory of dynamic programming." Bulletin of the American Mathematical Society 60.6 (1954): 503-515.
16. Thomas, Philip S., and Billy Okal. "A notation for Markov decision processes." arXiv preprint arXiv:1512.09075 (2015).
17. Sutton, Richard S., et al. "Policy gradient methods for reinforcement learning with function approximation." Advances in neural information processing systems. 2000.
18. Silver D, Lever G, Heess N, Degris T, Wierstra D, Riedmiller M. Deterministic policy gradient algorithms. InICML 2014 Jun 21.
19. Lillicrap TP, Hunt JJ, Pritzel A, Heess N, Erez T, Tassa Y, Silver D, Wierstra D. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971. 2015 Sep 9.
20. Moody, John, and Lizhong Wu. "Optimization of trading systems and portfolios." Proceedings of the IEEE/IAFE 1997 Computational Intelligence for Financial Engineering (CIFEr). IEEE, 1997.
21. Moody J, Wu L, Liao Y, Saffell M. Performance functions and reinforcement learning for trading systems and portfolios. Journal of Forecasting. 1998 Sep;17(5-6):441-70.

22. Moody JE, Saffell M. Reinforcement learning for trading. InAdvances in Neural Information Processing Systems 1999 (pp. 917-923).
23. Moody J, Saffell M. Learning to trade via direct reinforcement. IEEE transactions on neural Networks. 2001 Jul;12(4):875-89.
24. Lu, David W. "Agent Inspired Trading Using Recurrent Reinforcement Learning and LSTM Neural Networks." arXiv preprint arXiv:1707.07338 (2017).
25. Du, X., Zhai, J., & Lv, K. (2016). Algorithm trading using q-learning and recurrent reinforcement learning. positions, 1, 1.
26. Tang L. An actor-critic-based portfolio investment method inspired by benefit-risk optimization. Journal of Algorithms & Computational Technology. 2018 Dec;12(4):351-60.
27. Jiang Z, Xu D, Liang J. A deep reinforcement learning framework for the financial portfolio management problem. arXiv preprint arXiv:1706.10059. 2017 Jun 30.