香　港　大　學

THE UNIVERSITY OF HONG KONG

Department of Electrical and Electronic Engineering

Final Report 2017-2018

An Investigation on Quantitative Techniques for Computational Finance

Student Name: Cheung Chun Lok

Supervisor: Dr. Vincent W.L.Tam

University number: 3035187974

Curriculum: BEng(EE)

Date of submission: 9th of April, 2018

Abstract

With the development of the AI technologies, the use of machine learning algorithms have achieved remarkable performance in many fields, including computational finance. Most of the research focuses on the fundamental or technical indicators prediction which often achieved 55% - 65% directional accuracy. Although some studies had attempted financial portfolio management by reinforcement learning, it was mainly based on discrete action spaces, which leads to inflexibility of portfolio construction. In this research, continuous control with deep reinforcement learning and deterministic policy gradient was adopted. This algorithm is able to provide sufficient flexibility of portfolio construction without growing too much complexity of neural networks, even with a large number of assets. The result was compared with different types of algorithm for online portfolio selection, including Uniform Constant Rebalanced Portfolio (UCRP) and Uniform Buy and Hold Portfolio (UBHP). Based on the back-testing, at present stage, it was demonstrated that the proposed model was able to generate a portfolio value which is comparable with some of the existing algorithms.

Keywords: Machine Learning, Reinforcement Learning, Deterministic Policy Gradient, Continuous Control, Convolutional Neural Network, Portfolio management

Acknowledgment

I am honored to express my thanks of gratitude to my supervisor Dr. Vincent W. L. Tam, a responsible, patient and respectable scholar for his patient guidance, enthusiastic encouragement and expert advice of this project. He shared his recent research papers which inspired me a lot in the perspective of both computer science and computation finance.

Secondly, I heartily thank my groupmate Chow Hui Zhuo Garrett, a hard-working and motivated student from computer engineering. He helped me in finalizing my project within the limited time frame and contributed a lot for the project.

Table of Content

List of Figures

List of Tables

List of Abbreviations

Anticor	Anti-correlation
API	Application Programming Interface
CNN	Convolution Neural Network
DPG	Deterministic Policy Gradient
DQN	Deep Q Network
EIIE	Ensemble of Identical Independent Evaluators
FNN	Feedforward neural networks
LSTM	Long short-term memory
NYSE	New York Stock Exchange
OHLC	Open High Low Close
OLMAR	On-line portfolio selection with moving average reversion
PAMR	Passive aggressive mean reversion
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RMR	Robust Median Reversion
UBHP	Uniform Buy and Hold Portfolio
UCRP	Uniform Constant Rebalanced Portfolio

1. Introduction

Traditionally, to forecast the movement of financial markets, fundamental, technical and macroeconomic indicators are analyzed manually before making a decision. With the rise of computing power, algorithmic trading is widely adopted such that financial market can be simulated by a complex mathematics model. However, due to the complexity of markets, these algorithms are non-universal, or model-based, it always depends on the properties of market. Therefore, over the past years, much research was conducted to forecast the market performance by supervised learning, expecting the hidden patterns in financial data can be recognized. Nonetheless, its performance was not satisfied when comparing with the same application in other fields, for instance, image recognition usually achieves ninety percent or above accuracy. It is mainly because market is driven by a group of emotion and events which are hardly predicted by only figures. Recently, with the application of deep reinforcement learning, another challenging task, portfolio management is tried out apart from prediction, it optimizes the risk and return of the portfolio, which is the goal of this research.

The concept of utilizing neural network to approximate Q-value was firstly introduced by Google Deepmind [1], such that Atari 2600 games were tested by this algorithm. Raw pixel and score are firstly fed into neural networks, then the model selects an optimized action based on generated Q-value by ε-greedy policy. Eventually, it had been shown that it was able to master a wide range of games to above-human level. After the paper was published, DQN (Deep Q Network) is applied on variety of tasks, most of them can achieve a satisfactory performance. When applying DQN to trade a single security, a typical output should consist of 3 actions: BUY, HOLD, SELL. If the number of security increases to n, the number of output increases to 3^n as well, and usually DQN cannot handle with such massive output effectively. An alternative approach to overcome this problem is to divide the portfolio into low-beta security and high-beta security, a set of predefined numbers represents the action which sells p% of low-beta security and buys p% of high-beta security in each trading interval [2]. However, these action numbers are fixed, again, it suffers from discrete action-space which leads to the inflexibility of portfolio. The same problem was illustrated by Google Deepmind at 2016 [3] [4], it introduced a new algorithm called Deterministic Policy Gradient (DPG) to produce a continuous action space. This method can address some special tasks that require a continuous output, like controlling a robot arm, and eventually outperforms the traditionally DQN.

1.1. Objective

This research attempted to implement a reinforcement learning framework and utilize Deterministic Policy Gradient algorithm to generate a portfolio selection vector, each number in the vector represents an exact percentage of total capital allocated to an individual stock in the portfolio during next trading interval. Different reward functions and models were implemented in this research, and the result was compared with different trading algorithms as benchmarks

such that effectiveness of models could be evaluated. Additionally, this research model attempted to integrate with another price prediction model developed by my groupmate Chow Hui Zhuo Garrett, its daily price prediction result was designed to interrupt intermediate layer within this research model such that it further enhances profitability of model.

1.2. Outline

This essay firstly introduces the basic concepts of machine learning, deep learning and reinforcement learning, which include all relevant techniques and algorithms which were adopted in this research. Then, it explains the methodology of modeling a portfolio management problem mathematically by reinforcement learning and the setting of trading simulation. In addition, configuration of proposed models is clarified. The performance of different experimental settings is compared by figures and graphs, its results are further discussed and summarized.

2. Background

This section will briefly describe the background knowledge of machine learning, and the corresponding algorithms for updating a neural network.

2.1. Machine Learning

Machine learning is regarded as one of the subset of artificial intelligence, it is defined as the algorithm that can learn and make predictions on data without hard programming. The machine learning tasks can be classified into three categories:

Supervised Learning: During the training process, the predictions are compared to labels, its differences are used to update trainable parameters of the model. The typical applications are classification (K Nearest Neighbor or Support Vector Machine) and regression. Many financial forecast models are built by regression model.

Unsupervised Learning: No labels are presented in the model, the algorithms are able to explore and group hidden patterns in the features. The common algorithms are hierarchical clustering and k-means clustering.

Reinforcement Learning: Rewards and punishments are given as feedback to the model which is updated to maximize the reward. More details will be discussed in later section.

2.2. Deep Learning

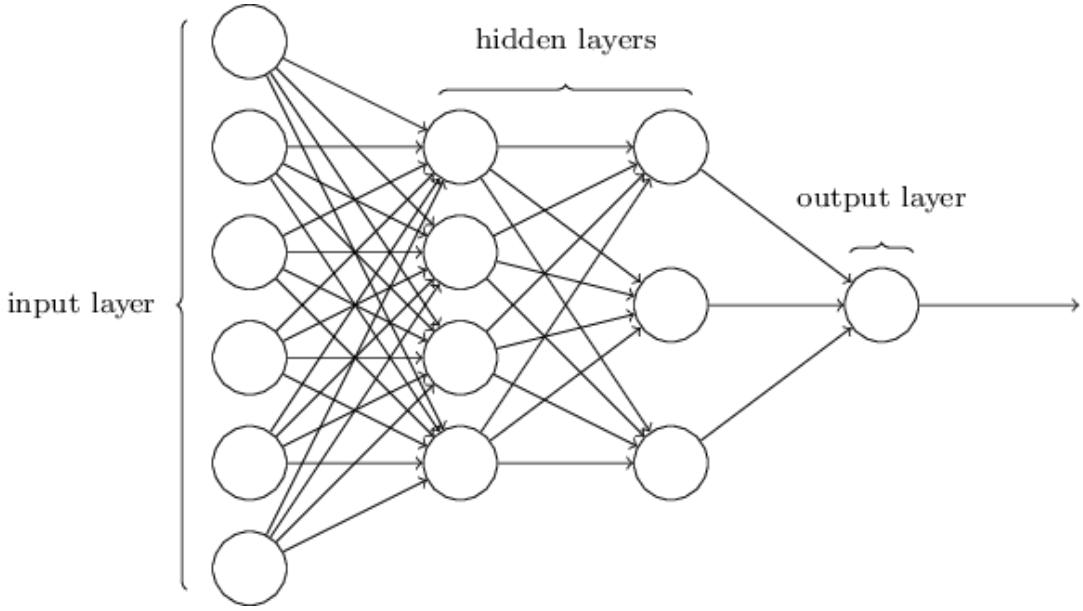

Figure 1 – Implementation of artificial neural network

Deep learning is regarded as subset of machine learning algorithms that (1) input features are processed through a chain of multiple layers, each layer is supposed to generate a non-linear output; (2) Output and input of successive layers are connected together; (3) Be able to learn in supervised or unsupervised manner; (4) The parameters of the network are updated through backpropagation by gradient.

2.3. Artificial neuron

Each neuron inside the layer receives inputs from previous layers, and produces a single output by a simple formula:

$$y = g\left(\sum_i w_i x_i + b\right), \quad (1)$$

The number of inputs depends on the previous layer, the weight w_i and constant bias b are updated by backpropagation from the preceding layers. The function g(x) is an activation function, which plays an important role in the network. Since it provides non-linearity property which enables the network to learn and model a complicated task. Otherwise, the overall transfer function of the network is just a polynomial with degree of one, and simply becomes a linear regression model.

2.4. Activation function

Artificial neural network is considered as Universal Function Approximator which is expected to learn and model any kinds of function. An activation function is one of the main keys to increase the complexity of the network. The activation must be differentiable, such that gradients of output of an individual neuron with respect to weights can be calculated in order to update the trainable parameters by backpropagation.

There are many activations which are available. Long short-term memory (LSTM) models

usually use sigmoid or hyperbolic tangent functions, while rectified linear unit (ReLU) is more popular in Convolutional neural networks (CNNs) and Feedforward neural networks (FNNs) [5]. The adopted activation functions in this research model were as follows:

2.4.1. Rectified Linear Unit (ReLU)

Since the proposed model in this research made use of convolution layers and fully connected layers, rectified linear unit was adopted. It can be expressed as a simple function:

$$g(x) = \max(0, x) \tag{2}$$

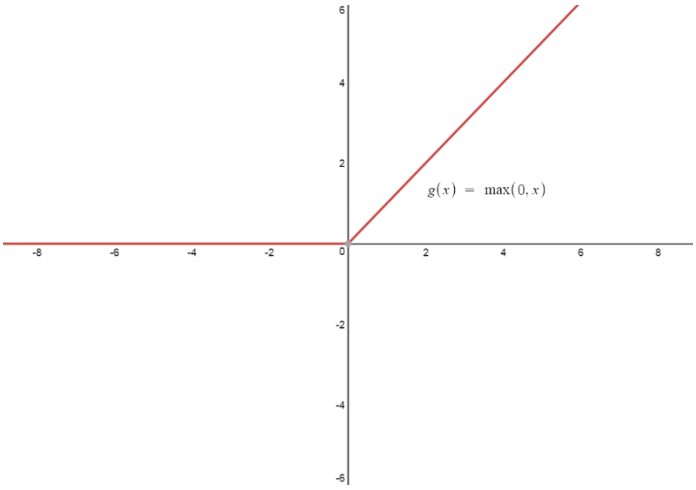

Figure 2 – Graph of Rectified Linear Unit

All values greater than zero are passed, otherwise, become zero. It is suggested that this function can induces the sparsity in the hidden units, it does not suffer from vanishing problem for convolution layer. However, as the function restrains negative values, it may lead to "dead neuron" which implies the outputs of a neuron are all zero regardless of inputs. Therefore, some improved versions like leaky ReLU or parametric ReLU are aimed to solve this problem by introducing a small value for negative region.

2.4.2. Hyperbolic Tangent

Hyperbolic tangent function was used as an activation function of LSTM layer in proposed model. It is suggested that tanh function helps to prevent the gradient vanishing especially for LSTM layer, since second derivative of tanh can sustain for a long range before going to zero. [6] The output also has a range of -1 and 1 which is suitable for internal gate operation inside LSTM layers. The function can be expressed as:

$$g(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3}$$

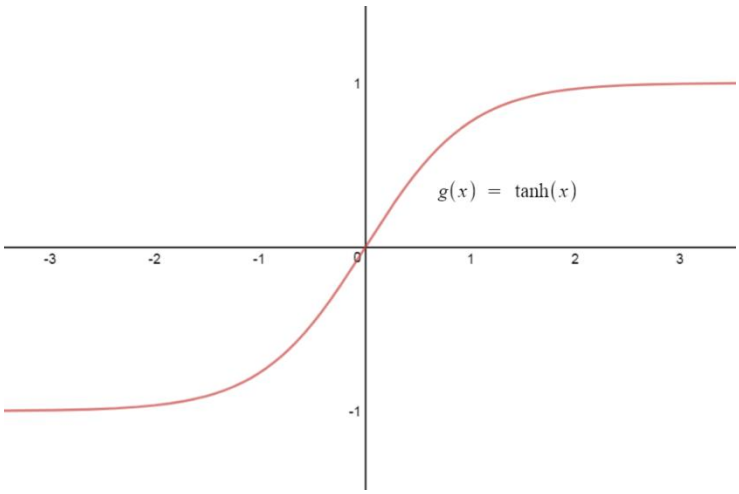

Figure 3 – Graph of tanh function

2.4.3. Softmax function

Another adopted activation function was softmax function, it can be expressed as:

$$g(x_j) = \frac{e^{x_j}}{\sum_i e^{x^i}} \tag{4}$$

$$\sum_i g(x_i) = 1 \tag{5}$$

This function represents the probability distribution over the output vector, therefore, it is often used for classification task. The function was aimed convert a voting score vector to a portfolio selection vector in this research model. Although the portfolio selection vector is not strongly related to concept of probability, it simply takes the advantages of its non-linearity and sum of outputs being zero to generate a reasonable output. So this function was arranged after the last layer of every proposed model in this research.

2.5. Convolution Layer

Convolution layer applies convolution operation to an input vector which is usually two or three-dimensions, each filter of the layer generates one single feature map respectively. The filter contains a set of trainable weights which are updated by backpropagation like a normal neuron. Theoretically, the filters help to detect a pattern from the input vector.

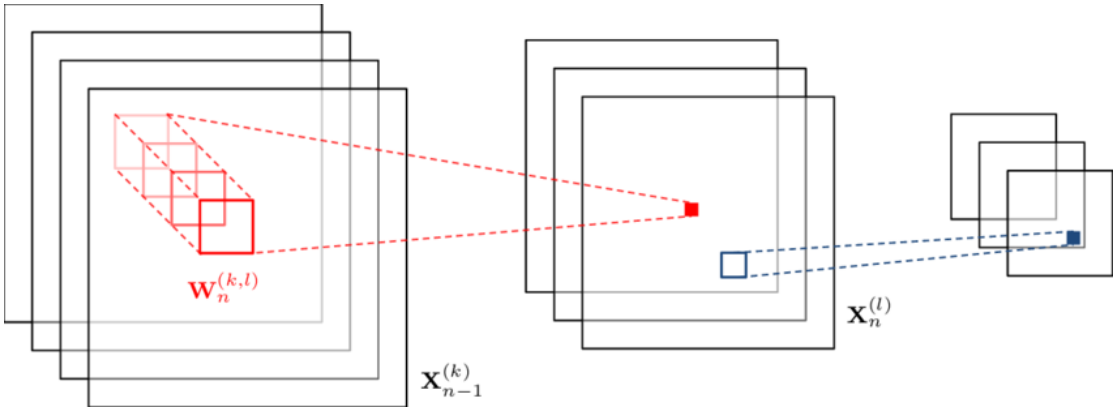

Figure 4 – Implementation of convolution layer

The Convolutional Neural Network (CNN) has shown excellent performance in many computer vision and machine learning problems which consists of high dimensional input features [7]. Normally, for stock prediction task, the input features are usually two-dimensional, one of the dimensions is financial indicator, another is time series which depends on the observed interval. In contrast, for portfolio management task, input data usually consists of three dimensions which are financial indicators, time series and the corresponding security. Although Feed-Forward neural network (FNN) or Recurrent neural network (RNN) can still handle such input vector with three-dimensional, CNN offers benefits of ease of implementation and spatial information. For fully connected layer, outputs from preceding layer are being connected to all input neurons in next layer, in other words, all output neurons from preceding layer are equally important to next layer.

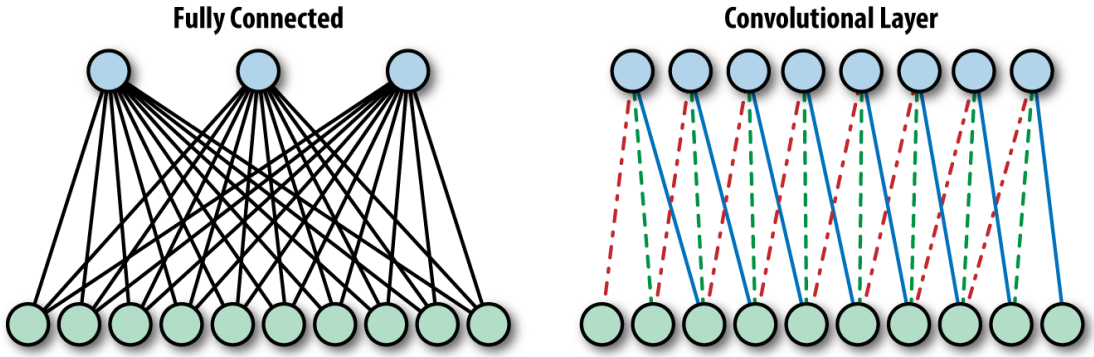

Figure 5 – Comparison between Convolution and fully-connected layer

On the other hand, filters in convolution layer can extract a partial pattern from the input effectively. For example, if there is a set of stock price with 60 days, literally, stock price of first day and last day are not much correlated, but strongly correlated to their adjacent days. Therefore, a filter size of 5x1 can be applied to extract every 5-days movement within 60 days which might strengthen the learning ability of the network.

Much recent research had already concluded that CNN can actually produce a comparable performance to that of FNN and SVM in stock price prediction [8] [9]. Therefore, convolution layers were selected as the hidden layer in one of proposed network.

2.6. Long Short-Term Memory

Long short term memory (LSTM) is a building block of recurrent neural layer which is a type of layer aimed to deal with sequential data. LSTM is explicitly invented to solve the long-term dependency problem which is mainly due to vanishing gradient problem. It is a unit composed of a cell, an input gate, an output gate and a forget gate. It enables the cell to "remember" a value over arbitrary time intervals. Each individual gate is similar to conventional artificial neuron.

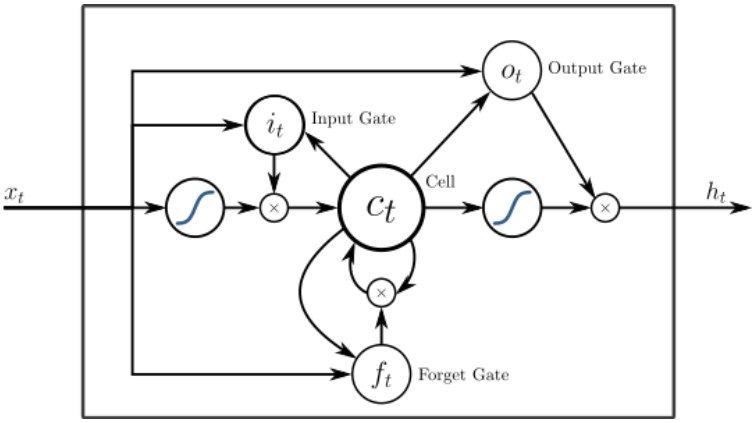

Figure 6 – Implementation of LSTM cell

In this research, effectiveness of LSTM layer was tested by two approaches. The first approach was to test the performance of LSTM solely, the second approach was to evaluate the performance when LSTM layer worked together with CNN. It is expected that LSTM will achieve a satisfactory performance on analyzing financial data, as LSTM was originally designed to handle a set of sequential data.

2.7. Reinforcement Learning

Reinforcement learning enables the model to address a complex problem, learn and take actions to maximize the reward through interaction with the environment. This environment is usually modelled by a mathematical framework called Markov decision processes (MDP) which defines five major components:

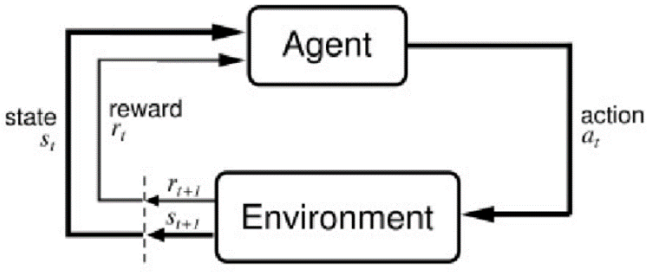

Figure 7 – The typical framing of a Reinforcement Learning (RL) scenario

1. A set of (possibly infinite) states s_t
2. A set of actions a_t
3. Transition probability distribution that action A_t in state S_t leads to new state s_{t+1}

$$\mathrm{P}(s_{t+1} = s' \mid s_t = s \cap a_t = \mathrm{a})$$

4. Reward function $r_a(\mathrm{s}, s_{t+1})$ after the transition by action A
5. Discount factor $\gamma \in [0,1]$ to penalize the future reward

Markov decision processes also provide one important property that state S_{t+1} entirely depends on S_t rather than any states before time t. The agent can establish a policy

$$\pi_\theta(\mathrm{s}) : \mathrm{S} \rightarrow \mathrm{A} \tag{6}$$

which determines the action to be chosen in current state. The policy can be further differentiated into two types:

Stochastic policy: it is modelled by a probability distribution such that randomness is introduced to the function, the exact action cannot be decided in a given state.

$$\pi_\theta(\mathrm{a} \mid \mathrm{s}) = \mathrm{P}(\mathrm{a} \mid \mathrm{s}) \tag{7}$$

Deterministic policy: it is a deterministic function, when policy network is fed by the same state, it always produces the same action, transits to the same next state and receives the same reward.

$$\mathrm{a} = \pi_\theta(\mathrm{s}) \tag{8}$$

In order to evaluate the performance, state value function V(s) and state-action value function Q(s, a) are introduced:

$$V_\pi(s) = E_\pi\{r_t \mid s_t = s\} = E_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right) \tag{9}$$

$$V_\pi(s) = E_\pi\{r_t \mid s_t = s\} = E_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right) \tag{10}$$

The concepts of V(s) and Q(s, a) are simple. V(s) is the expected reward in current state considering all possible actions while Q(s, a) is the expected reward of taking an individual action under current state. The state value and state-action value are the main key to establish most of reinforcement learning algorithms

2.7.1. Q-Learning

Q learning is a model-free, value-based reinforcement learning algorithm. It directly approximates and learns action-value function Q(s, a), the goal of the agent is to choose an action which can obtain a maximum action-value, therefore, this greedy policy can be interpreted as:

$$\pi(\mathrm{S}) = argmax_a Q_t(s, a) \tag{11}$$

The action-value function Q(s, a) can be further expanded by Bellman Equation

$$Q_t(s, a) = r_t(s, a) + \gamma * max_a Q_{t+1}(s_{t+1}, a_{t+1}) \tag{12}$$

Since function approximator like Deep Q network only generates Q value for certain action, the action space must be discrete, which may not be suitable for portfolio management task.

2.7.2. Deterministic Policy Gradient

The deterministic policy gradient (DPG) algorithm is similar to stochastic policy gradient

algorithm, the only difference is that action from stochastic policy is based on a probability distribution, which might be more difficult to compute the gradient, it implies that DPG can be estimated much more efficiently than the usual stochastic policy gradient [4].

The concept of DPG is to directly maximize the Q-value by gradient ascending method. Firstly, by considering a deterministic policy μ_θ with trainable model parameters θ which are weights of neural network in this case, a performance objective can be defined as,

$$J(\mu_\theta) = Q(s, \mu_\theta) \tag{13}$$

This is simply an action-value function with respect to network parameters (or policy). In other words, the performance objective maps an output from policy network to an action value. Next, the gradient of performance objective can be estimated by a mini-batch which comprises N observed states,

$$\nabla_\theta J(\mu_\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \mu_\theta(s_i) * \nabla_a Q(s_i, a_i)|_{a=\mu_\theta} \tag{14}$$

Therefore, the trainable model parameters are updated by

$$\theta' = \theta + \alpha \nabla_\theta J(\mu_\theta) \tag{15}$$

where α is the learning rate and the plus sign indicates that these parameters are updated toward maxima of performance objective $J(\mu_\theta)$.

This algorithm enables continuous action control, and the implementation is as simple as Q-learning with neural network. Hence, it was adopted in this research model.

3. Problem definition

Portfolio management is defined as making decisions about investment mix and policy, continuous asset reallocation of number of financial assets. The final goal is to optimize the overall return at a given appetite for risk. For an automated trading machine, financial assets in portfolio are reallocated regularly within a defined period, and it should be able to react to market trends. This section defines the portfolio management problems in a mathematics manner.

3.1. Hypotheses

In this research, the experiment is only carried out by back-testing, all trading is based on historical data. However, back-testing is never same as real-time trading, some hypotheses should be made.

1 The liquidity of all traded asset or security is high enough that the exact price can be bought and sold instantly. That is the spread between bid and ask prices is extremely small.

2 Volume of each trading made by this experiment is small enough that there is no significant impact on the overall markets.

3 Continuity of Stock Unit: The agent can trade at any number of stock (e.g. 5.5 stocks).

In the reality, if the overall trading volume in a market is high enough, it is possible that first two assumptions stay true. [9] Additionally, if the initial capital is large, the third assumption also stays true.

3.2. Trading period

Under proposed model, the trading interval was defined on a daily basis. One of the reason is that it provides a large amount of data to be trained and tested comparing on weekly and monthly basis. For training a neural network, feature sets should be as many as possible, it prevents the model from overfitting to a small set of data, but learns from different patterns, especially the period of finance crisis 2008. Since much conducted research in stock price forecast generally have poor performance in downtrend periods. Therefore, data in year of 2008 should be included in the trading period to evaluate how the model reacts to economic downturn. In addition, the daily financial data which starts from 1997 can be easily accessed by python API, including Yahoo Finance API and Quandl API. The trading period can be further divided into training, validating and testing period.

3.2.1. Training period

Training period enables model to learn and update the model's parameters in order to maximize a specific reward. For portfolio management task, the training period was defined as 7 years, which contains around 1700 data points per episode. Iterating the training period once is defined as one episode. For training approach in proposed strategy, the model was trained by 200 to 300 episodes which might vary according to different models. Additionally, the feature set to be trained in each episode was randomly constructed by different stocks within the same GICS sector, this approach was aimed to prevent the model from overfitting the same feature set.

3.2.2. Validating period

Validating period evaluates the performance of trained model with unseen data. The best episode was chosen according to its performance in validating period. The length was defined as 1 year when serval tests can be carried out in order to obtain a set of best model hyper-parameters. Learning rate, sample bias, rolling step and cash bias were being tuned in this period. Detail definition of each parameters will be discussed in later section.

3.2.3. Testing period

Testing period evaluates how the trained model with a set of validated model hyper-parameter reacts to unseen data, it reflects the true performance of the model in a more realistic manner. In this case, the length of testing period was defined as 2 years right after validating period. Some pretest was conducted which shown that the performance dropped when time went far away from training or validating period.

Generally, in supervised learning, the model's parameter does not update during testing period. In the proposed model, the concept of online learning was applied. That is the model was continuously receiving a reward and being updated during testing period, it enables the model to optimize a new policy towards the unseen data. Since there is no guarantee that all patterns and strategies learned in training or validating period were useful and effective in testing period.

3.3. Data Treatments

The portfolio contains m assets. The input price tensor is a three-dimensional tensor, each of the dimension represents assets, time and financial indicator respectively.

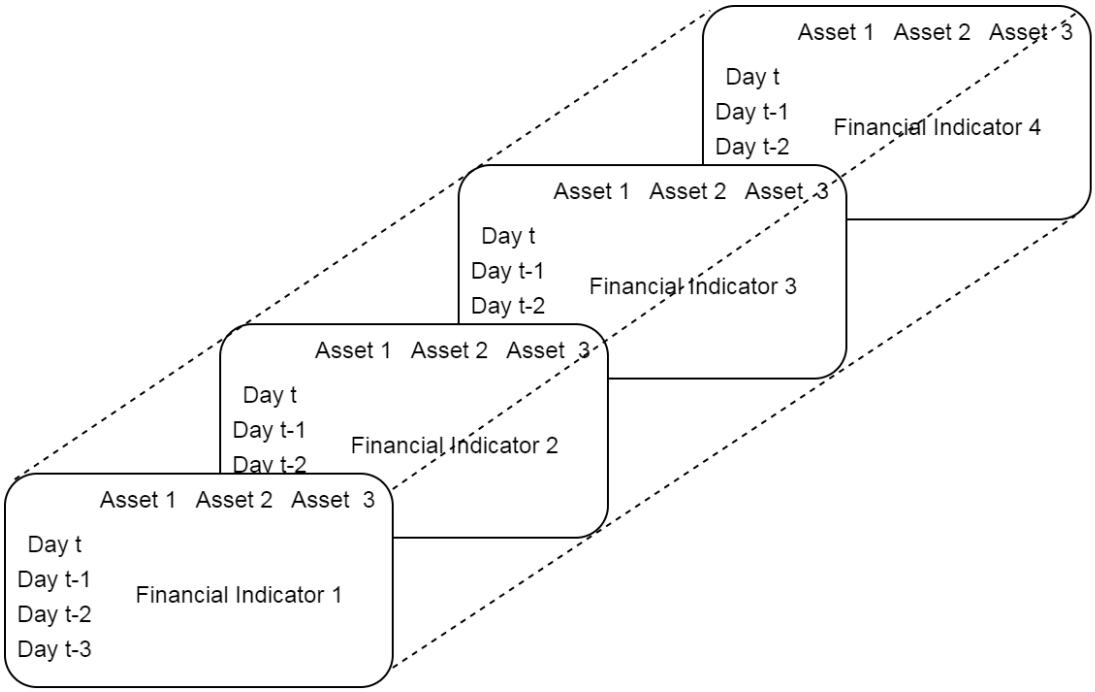

Figure 8 – Illustration of input price matrices X_t

3.3.1. Asset Selection

Stocks in S&P500 were selected to be tested in the experiment and the number of assets in portfolio was set to 15 in this stage. Although a portfolio of 20 to 30 stocks is considered to be ideal in U.S. market, where stocks are less correlated to the overall market [10]. Yet, the result will only compare to different trading algorithms, absolute profit is not major consideration in this experiment. For stock pre-selection, it was mainly based on beta β which indicates the volatility of individual asset compared to market, beta β greater than one is considered to be more volatile than the market, and vice versa.

Portfolio	Beta	GICS sector	Company
1	High	Information Technology	'AAPL', 'AMAT', 'AMD', 'CSCO', 'EBAY', 'GLW', 'HPQ', 'IBM', 'INTC', 'KLAC', 'MSFT', 'MU', 'NVDA', 'QCOM', 'TXN'
2	Low	Consumer Discretionary	'AZO', 'BBY', 'DHI', 'F', 'GPS', 'GRMN', 'HOG', 'JWN', 'MAT', 'MCD', 'NKE', 'SBUX', 'TJX', 'TWX', 'YUM'
3	Hybrid	Industrials	'BA', 'CAT', 'CTAS', 'EMR', 'FDX', 'GD', 'GE', 'LLL', 'LUV', 'MAS', 'MMM', 'NOC', 'RSG', 'UNP', 'WM'

Table 1 – Description of pre-selected assets

Three groups of stocks combination were pre-selected to construct portfolios for experiment.

Portfolio 1 consisted of high-beta stocks only within Information Technology Sector. Portfolio 2 consisted of low-beta stocks only within Consumer Discretionary Sector. Portfolio 3 consisted of composition of high-beta stocks and low-beta stocks with a ratio of 50:50 within Industrials Sector.

The sector is defined by Global Industry Classification Standard (GICS) which is for use by the global financial community. The correlation between portfolio return by this research model and beta was one of interests in this research.

During training phase, stocks within the corresponding sector were chosen randomly to construct a portfolio for each episode. For example, every stock within Information Technology Sector has equal chance to be selected to train an episode if test of Portfolio 1 is going to be carried out. This approach can prevent model from overfitting a same set of financial data and enable the model to learn overall market movement or specific correlation within a sector effectively. Normally, each sector contains around 70 companies, the total combination of a portfolio consisting of 15 stocks is ${}_{70}C_{15} = 7x10^{14}$, hence, the input price matrix is almost impossible to be repeated during training.

3.3.2. Financial indicator

It is expected that the model is able to recognize a specific pattern within financial indicators which leads to positive reward (return). For example, the model may learn that increasing price with stable volume is a bullish sign and voting score for that stock should be enlarged.

Name	Definition / Implication
Fundamental Indicators	
Close	Daily adjusted close price
Open	Daily adjusted open price
High	Daily adjusted high price
Low	Daily adjusted low price
Volume	Daily trading volume
Technical Indicators	
ROC	Price rate of change: shows speed of stocks price changing
MACD	Moving average convergence divergence: shows trend following and momentum of stocks
MA5	5-day simple moving average: identifies short-term price trends
MA10	10-day simple moving average: identifies short-term price trends
EMA20	20-day Exponential Moving Average: identifies weighted short-term price trends
SP20	20-day Sharpe ratio: measures short-term risk adjusted ratio

Table 2 – Description of the input variables

For fundamental indicators:

Adjusted close, adjusted open, adjusted high, adjusted low and volume were selected as

financial indicators. Adjusted price includes effect of stock splits and dividends, which allows the simulation to be more realistic and the model to learn a correct trend. For example, APPLE had a 7 for 1 stock split at June 09, 2014, the model may misclassify this as huge downtrend if it is fed by actual price directly.

For technical indicators:

Technical indicators are often used extensively for future price level prediction by active traders in the market, as these indicators are mainly designed for analyzing short-term price movements. Since the proposed model executed trading order on a daily basis, it is expected that the model is able to establish a technical analysis strategy to maximize portfolio return. As a result, some popular technical indicators used by algorithm trading were selected as part of input price matrix.

3.3.3. Time series

A sliding window was defined as a set of delayed prices plus today price. As mentioned in section 2.7, reinforcement learning problem is usually modeled as Markov Decision Process (MDP) which has the property of next state depending on current state only. The received information from current state should be sufficient and informative enough that a profitable trading strategy can be made by current state only but not any previous states.

In the experiment, today price plus last 19 business day prices which are the daily price of one month were proposed. During pre-testing, a sliding window sizes of 10, 20, 40, 60 were tested, it turned out window sizes of 20, 40, 60 had the similar performance which was better than size of 10 significantly. Yet, a sliding window size of 20 contains less data points which can fasten the training process, therefore, 20 business days was defined as time length of input price matrix.

3.3.4. Data normalization

For a neural network, data must be pre-processed before feeding into the network. In the experiment, all financial indicators which has the unit of price were nominalized with its latest value, the mathematical expression is:

$$\boldsymbol{X_{price}} = \left(1\,,\ \frac{\vec{X}_{price,\ t-1}}{\vec{X}_{price,\ t}}\,,\ \frac{\vec{X}_{price,\ t-2}}{\vec{X}_{price,\ t}}, \dots \dots\right)^{T} \tag{16}$$

By this approach, each element within the input matrices now depends on the latest value. Literally, the latest price, for instance, latest adjusted close is the closest price before next trading (asset reallocation) taking place, it is the most critical element within the price matrices X_{price}. As a result, under this normalization approach, the whole matrix is an alternative representation of latest price to a certain extent.

Yet, ROC and SP20 are the indicators that processed without normalization. This is because there are no unit for these two indicators technically, and its range normally lies between 0 and 2 which is good enough to be processed in neural network without normalization.

3.3.5. Portfolio selection vector

The output vector is a one-dimensional vector with a size of m+1 (number of assets + 1). Each element represents the percentage of each asset or cash to be held in the portfolio at t day.

$$\vec{w}_t = (w_1, w_2, w_3 \dots \dots)^T \tag{17}$$

$$\sum_{i=0}^{m} w_i = 1 \tag{18}$$

The action of short selling was not considered in this experiment, so all weights are positive number. Although cash holdings are often criticized for reducing profitability in bullish market, it is also suggested that it can reinforce the liquidity of portfolio due to economic uncertainties [11] and suppress portfolio volatility. For example, if trading agent predicts that all holding stocks will have a poor performance in next trading interval, it is reasonable to redistribute some capital to cash holdings. By adjusting parameter of cash bias manually, tendency of holding cash and aggressiveness of portfolio can be controlled.

3.4. Portfolio Return

In a real situation, it is impossible to access market close price and trade it on the same day, as the market has already closed once received OHLC price. One possible solution is to trade it by Market-On-Close Order and use the closing price of 15 minutes before end of market day (NYSE) to represent close price in input price matrix, however, it requires minute by minute data which is not suitable for this experiment.

Another solution which was adopted is to define the actual trading (assets reallocation) which takes place on the next business day. The trading simulation procedures were as follows: Firstly, input vector X_{t-1} which includes a set of delayed OHLC prices + technical indicators at time t-1 was retrieved from markets. Then a portfolio selection vector w_t was generated by the policy network. Lastly, the asset reallocation was executed at market open of time t. By hypotheses 1 and 3, it is assumed that order could be made on any exact open price.

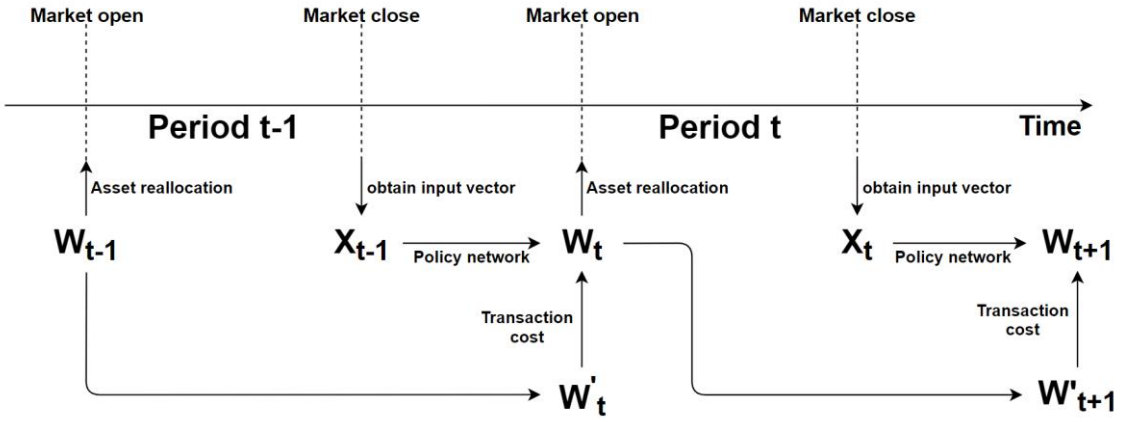

Figure 9 – Illustration of trading simulation

Since all the orders were executed at open price, the day return should be determined by the

difference of two successive adjusted open price. Let y_t denotes adjusted open price vector, each element represents adjusted open price of respective assets in the portfolio at time t. Portfolio day return can be derived as:

$$r_t = \vec{w}_{t-1} \cdot \frac{\vec{y}_t}{\vec{y}_{t-1}} \tag{19}$$

$$\text{cumulative return} = \prod_{t=0}^{n} \vec{w}_{t-1} \cdot \frac{\vec{y}_t}{\vec{y}_{t-1}} \tag{20}$$

which excludes the transaction cost. The cash component (in U.S dollar) of y_t was defined as 1 at any time, such that

$$\frac{y_{cash,t}}{y_{cast,t-1}} = 1 \tag{21}$$

It implies that there is no profit and loss for holding cash component. It is assumed that all trading orders were executed in U.S. dollars, the value of U.S. dollar remains the same with respect to stock price regardless of inflation. Remind that the definition of w_t is the percentage of each asset or cash to be held in the portfolio at t day.

3.4.1. Transaction cost

In reality, investors make order through brokerage which charges commission fee per share. Since the trading machine rebalances the portfolio every stock market open, the transaction cost becomes significant. There are three main types of brokerage, the first one is traditional "full-service" brokerage. These brokers tend to provide a wide range of services and products, and charge higher commission fee. Another one is discount brokerage which does not provide any investment advice but charges a lower commission fee, additionally, these trading orders are executed through a computerized system. The third one is online brokerage which charges the least expensive transaction fees. They only provide online trading service to investor. To minimize the transaction cost, it is assumed that trading decisions made by proposed models are executed through online brokerage. The transaction cost (in percentage) at time t can be formulated as:

$$\text{transaction cost} = c_s * \sum_{i=0}^{m-1} |w_{t,i} - w'_{t,i}| \tag{22}$$

c_s is the transaction factor which is 0.25% of trade value. This is an approximation of trading fee, since online brokers charge a per-trade flat fee which ranges between $5 to $30 per trade for each transaction, and a monthly account maintenance fees which are usually around 0.5%. On the other hand, w'_t is the weights redistribution from w_{t-1} due to market movement. For instance, if an individual stock price rose while others remained the same, the percentage of total capital allocated to that stock would increase, thus, the weight for that individual stock would slightly increase. However, for ease of computation, the difference between w_{t-1} and w'_t was ignored because it is considered to be very small on a daily basis. Therefore,

$$\vec{w}_t' \approx \vec{w}_{t-1} \tag{23}$$

$$\text{transaction cost} \approx c_s * \sum_{i=0}^{m-1} |w_{t,i} - w_{t-1,i}| \tag{24}$$

Daily return including transaction cost can be rewritten as:

$$r_t = \vec{w}_{t-1} \cdot \frac{\vec{y}_t}{\vec{y}_{t-1}} * (1 - c_s) * \sum_{i=0}^{m-1} |w_{t,i} - w_{t-1,i}| \tag{25}$$

4. Proposed Model

This section introduces the design of the policy network, presents a reinforcement learning framework for problem defined at section 3.

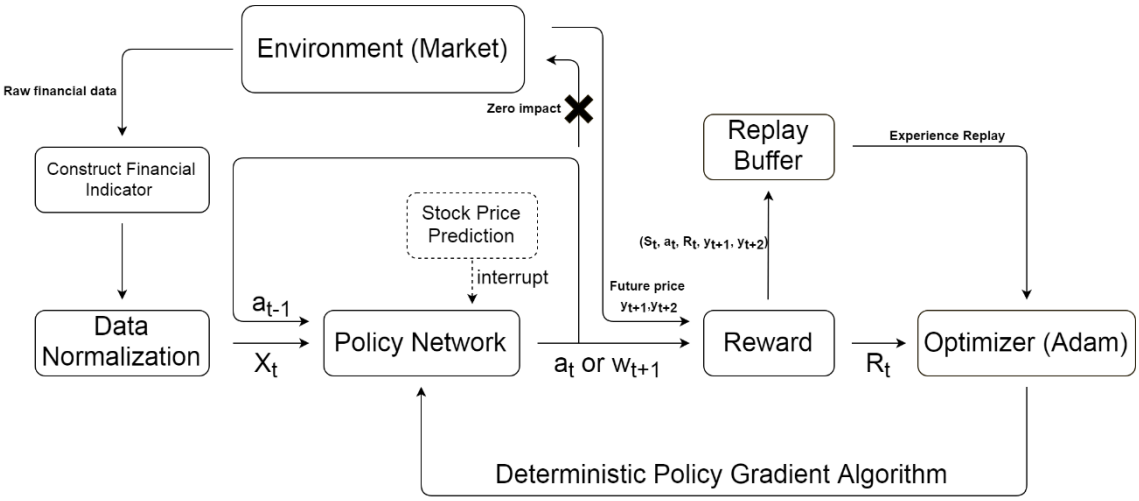

Figure 10 – Flowchart of Proposed Model

Figure 4 demonstrates the framework of proposed model. The details of each procedures and sub-model will be discussed one by one below.

4.1. Environment and agent

Environment represents market movements. For every time step, agent receives information from environment as state and computes an action which might influence the environment, then the agent receives a reward which indicates how well the agent did.

However, it is impossible to obtain every single information from a complex market environment, only relevant information should be considered. Hence, the input price matrices X_t which contains a set of fundamental and technical indicators of every asset in portfolio was adopted. This is regarded as external state because RL agent cannot influence overall market movement, in other words, the future price is not affected by any previous action based on hypothesis 2.

Furthermore, the portfolio selection vector from previous time step was also considered as

state, since the difference between w_t and w_{t+1} is proportional to transaction cost and day return. It is expected that the RL agent can learn an optimized strategy without spending too much transaction cost. This is regarded as internal state, as the last action was decided by RL agent itself. Combining both internal and external, the state at time t can be interpreted as:

$$s_t = (X_t, w_t) = (X_t, a_{t-1}) \tag{26}$$

The action a_t generated from the policy network redistributes the wealth among assets in open market of t+1,

$$a_t = w_{t+1} = \pi_\theta(s_t) \tag{27}$$

where π_θ denotes the policy network given trainable parameters θ

4.2. Policy network

In this research, four policy networks with different combination of convolution layers, fully connected layers and LSTM layers were proposed. All proposed networks follow the concept of Ensemble of Identical Independent Evaluators (EIIE) which was introduced by Jiang 2016 [9]. This concept is to evaluate the performance of each asset independently, but sharing the same parameters between columns in EIIE configuration. Therefore, the network does not bias to one or two specific investment combinations, since all data are passed independently and a voting score is submitted to softmax layer to determine an action. For example, if the input price matrix of policy network is constructed in the order of [AAPL, GOOGL, MSFT], the policy network should produce an exact same voting score of each equity even the input order is reversed to [MSFT, GOOGL, AAPL].

The aims of testing four policy networks are to evaluate the effectiveness of CNN and LSTM in portfolio selection and explore the best combination.

4.2.1. CNN

This network consists of three convolution layers only to produce a set of voting score for each equity.

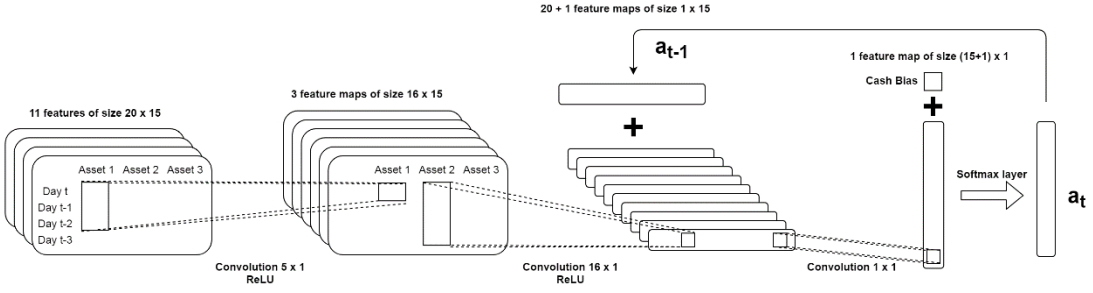

Figure 11 – Implementation of CNN network

All kernel sizes are nx1 which ensures all assets are being processed separately. The first convolution layer has a kernel size of 5x1, which was designed to extract a 5-day pattern from the normalized input. Alternatively, it can be changed to 3x1 to extract a 3-day pattern or even shorter period. The second layer has a kernel size of 16x1, which was designed to convert all feature maps to a one-dimensional tensor. The third layer is to combine all one-dimensional feature maps

to a single one-dimensional feature map, which is voting score of each individual asset. Then the voting score plus a cash bias are passed by a softmax layer to generate a set of valid portfolio selection weights such that sum of output equals to one.

As mentioned in section 4.1, each state included previous action in order to minimize transaction cost, the last action was inserted between second and third convolution layer. Consequently, the size of action vector is same as feature maps after second convolution layer, and hence the action vector are convoluting simultaneously with other feature maps.

4.2.2. CNN + Dense

This model consists of two convolution layers and two fully connected layers.

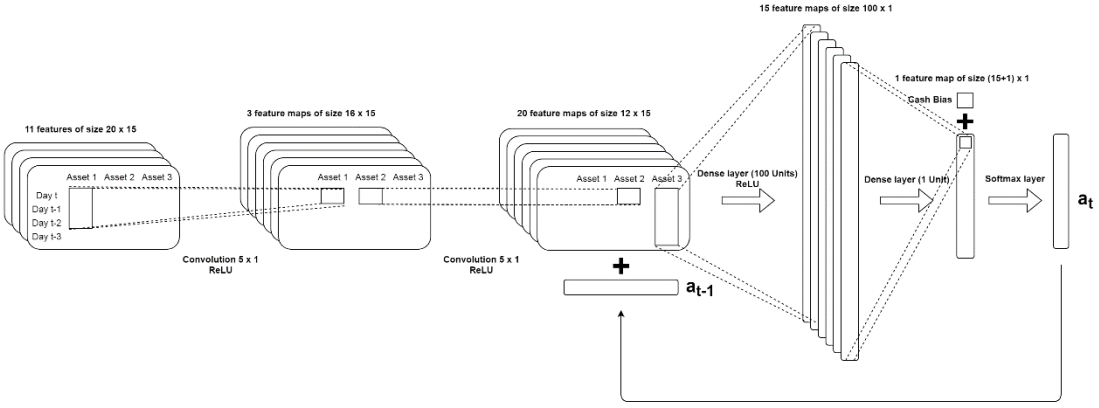

Figure 12 – Implementation of CNN + Dense network

The first convolution layer which extracts a 5-day pattern is same as CNN model defined in section 4.2.1. The second convolution layer with a filter of 5x1 further convolutes the data. After that, each asset has its own set of convoluted values with size of 20x(12x1), these values are flattened to a size of 1x240 and passed through two fully connected layers one by one such that a voting score of each individual asset is obtained each time. That is, there are 15 one-dimensional tensors with size of 1x240 each representing an asset in portfolio, passing to fully connected layers separately. It can be achieved by TimeDistributed layer which enables fully connected layers to pass serval 1-dimensional tensors independently. Therefore, it follows the concept of Ensemble of Identical Independent Evaluators (EIIE) which data of all assets are independently processed within all layers.

4.2.3. CNN + LSTM

This model consists of two convolution layers and two LSTM layer.

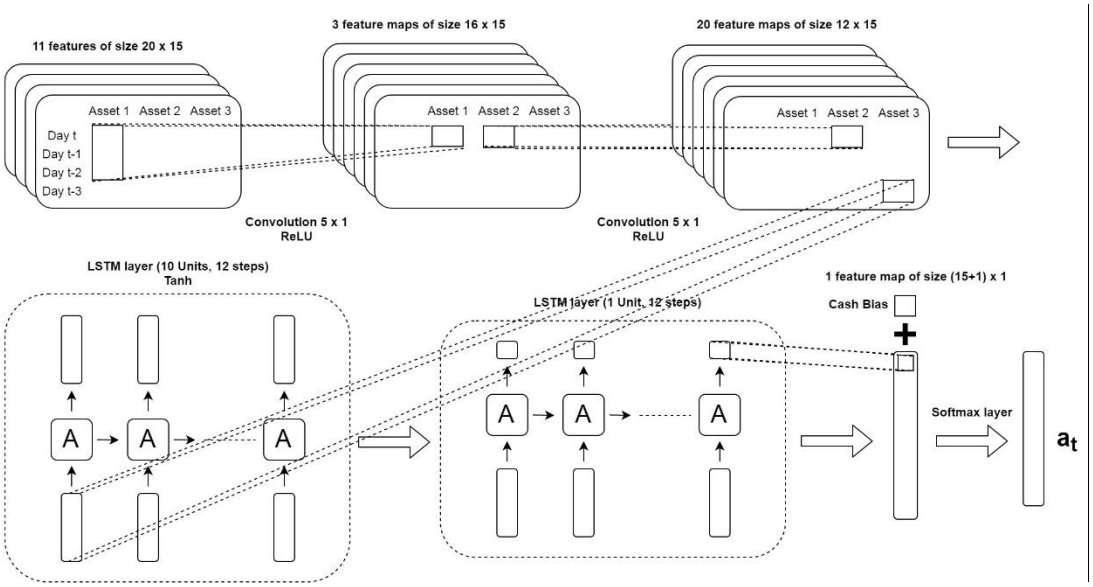

Figure 13 – Implementation of CNN + LSTM network

The first two convolution layers are exactly same as CNN + Dense model defined in section 4.2.2. Then, a set of convoluted values are reshaped to a size of 15x12x20 which can be regarded as each individual asset having a 2-dimensional feature map of size 12x20, 12 is the timesteps and 20 is the number of features. Consequently, feature map of each asset is passed through LSTM layer one by one in sequential order and it generates a voting score respectively. Similarly, it can be achieved by TimeDistributed layer which enables LSTM layer to pass serval 2-dimensional tensors separately. This model does not introduce the last action as input to network like CNN or CNN + Dense does, because the shape of a single last action does not match with the sequential order of LSTM input. The aim of this network is to explore the possibility of connecting CNN and LSTM in series to process and analyze a set of financial data. Noted that there is no activation function at the second LSTM layer, since an activation like tanh or sigmoid might sustain the voting score in a range of -1 and 1, passing these values to softmax layer normally generate a set of uniform weights and hence the model cannot establish an aggressive strategy eventually.

4.2.4. LSTM

This model consists of two LSTM layers only.

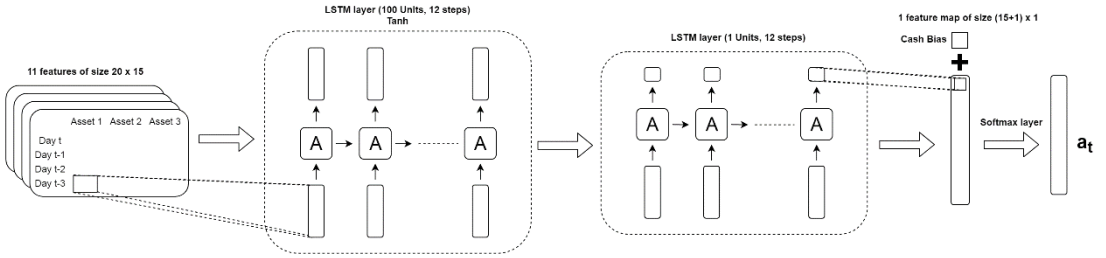

Figure 14 – Implementation of LSTM network

The input data is firstly reshaped to size of 15x20x11 which can be regarded as each asset in portfolio having an input of 20x11. 20 is the timesteps which is sliding window size of input

and 11 is the number of features which contains both fundamental and technical indicators defined in section 3.3.2. The configuration of LSTM layers is same as CNN + LSTM model. Nonetheless, because of absence of convolution layer, the number of LSTM cell should be increased to strengthen the computational ability of network which might proportional to the number of trainable parameters in the network. This model was designed to test the effectiveness of solely using LSTM layers to analyze and process a set of financial data.

4.2.5. Cash Bias

Cash bias is a constant number associated with voting score layer. It determines the amount of cash holdings in next trading interval. For example, if the cash bias was set to 80 while other equites obtained a voting score of 50-60, after passing to the softmax layer, it will generate an action of reallocating most of the capital to cash holdings and vice versa. Therefore, the activeness of the trading strategy can be controlled by adjusting cash bias. In different versions of trained model, the voting score of stocks has a different range, thus, the cash bias should also be modified within a similar range. The cash bias was tuned in validation phase according to its performance and range of voting score.

4.3. Reward function

The main objective of policy network is to maximize the reward or the action-state value Q by computing gradient of reward with respect to network's trainable parameter. Under hypothesis 2, action from policy network would not affect any future state and price, thus, it was boldly proposed that the discount factor of Q-value was zero. It implies the model does not care about future reward other than reward made by current action. It is reasonable because this research model reallocated all capital on a daily basis, the reward in next trading day is not important as long as the model is able to optimize a current reward, though, a current action affects future reward due to transaction cost to a certain extent.

Generally, it is necessary to have a function approximator to map an action to reward, which is what critic network does in Deterministic Deep Policy Gradient Algorithms (DDPG). For example, for a self-driving task, there is no formulated relationship between action and reward. However, for portfolio management task, a relationship between reward and action can be established by a simple formula, then the model can get rid of critic network and reduce its complexity.

Under the proposed model, the reward was directly calculated by dataflow graph. This concept is same as neural network except there is no trainable parameter inside the graph. In other words, it is an extension from policy network to generate a reward. Since the policy network is implemented by Keras (Tensorflow backend) and the reward function is a dataflow graph implemented by Tensorflow, the entire process from state to reward can also be regarded as a single Tensorflow network. As a result, the gradient can be back-propagated from reward to model's weight layer by layer.

In this experiment, four reward functions were created. By mini-batch updating, the reward function was fed by a set of time-ordered actions and future prices. Let t_b and n_b denote a specific point of time and size of each mini-batch respectively.

4.3.1. Average logarithmic cumulative return

$$R_1 = \frac{1}{n_b} \sum_{t=t_b}^{t_b+n_b-1} \ln(r_t) \tag{28}$$

where r_t is the day return defined in section 3.4.

The goal of the model is to generate a portfolio value as high as possible, therefore, it is reasonable to utilize an average logarithmic cumulative return within a specific period as reward function. By logarithm operation, cumulative return can be expressed as summation instead of multiplication. In addition, it provides a linear relationship of each element and the gradient of reward function is easier be computed.

4.3.2. Relative average logarithmic cumulative return

$$R_2 = \frac{1}{n_b} \sum_{t=t_b}^{t_b+n_b-1} \ln(r_t) - ln\left(\frac{1}{m} sum\left\{\frac{\vec{y}_t}{\vec{y}_{t-1}}\right\}\right) \tag{29}$$

It is simply logarithmic difference between return generated by policy network and Uniform Constant Rebalanced Strategy (UCRP) which is uniformly holding each equity throughout the period. Uniform Constant Rebalanced Portfolio (UCRP) is one of the important benchmark in this experiment, the main expectation of the model is not only obtaining a considerable profit, but also outperforming the benchmark. It gives a positive reward when model outperforms the average return even it suffers a loss. In contrast, it gives a punishment to model when it underperforms the average return even it earns a profit.

4.3.3. Sharpe ratio

$$R_3 = \frac{\sqrt{n_b}\left(\frac{1}{n_b}\sum_{t=t_b}^{t_b+n_b-1} \ln(r_t) - \ln(r_f)\right)}{std\{\ln(r_t), \ln(r_{t_b+1}), \dots\dots, \ln(r_{t_b+n_b+1})\}} \tag{30}$$

It measures the risk-adjusted return across a specific period of time. r_f represents daily risk-free rate of return which is a return of an absolutely risk-free investment over a period of time. Practically, it does not exist because every investment must contain a risk. The three-month U.S. Treasury bill was chosen as risk-free rate of this model because the market considers there to be almost impossible of the government defaulting on its obligations.

The function represents Sharpe ratio of the entire mini-batch. The definition of Sharpe ratio is (Mean portfolio return − Risk-free rate) divided by standard deviation of portfolio return. If the deviation of return is large, it implies the portfolio cannot be maintained in a steady way and vice

versa. Furthermore, the Sharpe ratio was calculated by daily return, it should be scaled by a factor of $\sqrt{n_b}$ to represent an actual value within a specific period defined in mini-batch (from t_b to $t_b + n_b - 1$).

4.3.4. Calmar ratio

$$R_4 = \frac{\left(\prod_{t=t_b}^{t_b+n_b-1} r_t\right) - 1}{M.D.D} \tag{31}$$

It measures the risk-adjusted return across a specific period of time. The numerator represents the overall compounded return while denominator is the maximum drawdown which measures maximum cumulative loss from a peak to a following bottom [12]. The concept of Calmar ratio is similar to Sharpe ratio, the only difference is that Sharpe ratio uses standard derivation of return as risk while Calmar ratio uses maximum drawdown to measures risk of the portfolio. Therefore, risk in Calmar ratio focuses on the greatest loss that a portfolio could suffer rather than the overall derivation of return.

4.4. Experience replay

Experience replay is one of the important technique to fasten the learning process of neural network. The concept of experience replay is to reuse previous states, the model is fed by these data and updated by mini-batch. With experience stored in a replay buffer, it enables the model to eliminate the temporal correlation by combining more and less recent experiences for update, and rare experience can be reused [13]. It provides the advantage of enhancing the sample efficiency by allowing mini-batch updates, and improving the computational efficiency, especially when the model runs on a GPU [14].

After model produces an action, the experience e_t should be wrote to replay buffer memory. Generally, experience e_t is defined as follow,

$$e_t = (s_t, a_t, R_t, s_{t+1}) \tag{32}$$

Noted that reward R_t is not day return r_t defined in section 3.4.

Under hypothesis 2, actions made by trading machine has no impact on market environment, so s_{t+1} is independent of a_t. In fact, next state s_{t+1} is not necessary for the whole replay process under this framework. In contrast, future open price y_{t+1} and y_{t+2} (defined at section 3.4) are essential for calculating reward under action a_t or w_{t+1}. For Q-learning, the neural network itself is an action-value approximator, Q-value can be obtained directly. In proposed model, the neural netowrk is a policy network, extra network (Graphs) was required to obtain a reward. Therefore, y_{t+1} and y_{t+2} should be included in the replay buffer, experience e_t for this model can be rewritten as:

$$e_t = (s_t, a_t, R_t, y_{t+1}, y_{t+2}) \tag{33}$$

The replay buffer operates in first in first out principle.

4.4.1. Stochastic Time-Order Batching

For supervised learning, batch is randomly drawn from the replay buffer, all data within the batch are unordered and disjoin. Nonetheless, market movement and corresponding action made by trading machine is continuous with time, it is reasonable that batching in this model should be a set of time-ordered prices, states and actions. That is, if the batch size is n_b, and a point of time t_b is randomly selected, the experience set should be $[e_{t_b}, e_{t_b+1}, e_{t_b+2} \dots \dots e_{t_b+n_b-1}]$. In consequence, the overall experience replay procedure can be illustrated as:

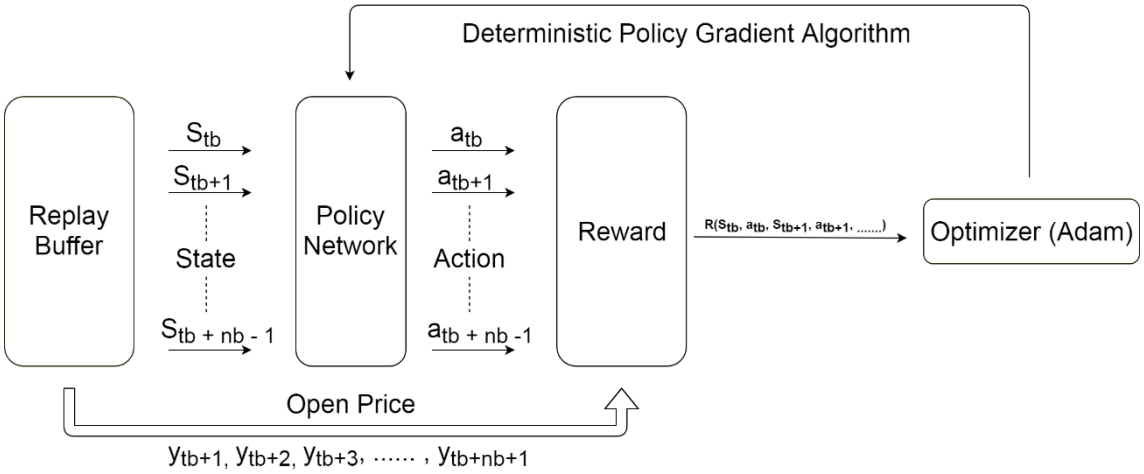

Figure 15 – Detailed diagram of experience replay

In training phase, experience replay was processed once each time step only, but the same period with random portfolio was iterated for many times. Conversely, since testing or validating period was only iterated once, it is necessary to increase the number of experience replay for each time step in order to enable the network to learn the most recent market movement. In this experiment, number of experience replay was tuned in validation phase and normally set to 0 to 30 per time step during testing phase.

4.4.2. Weighted experience

Whenever a mini-batch of experiences is drawn from replay buffer, the chance of each mini-batch to be selected follows a probability distribution. More recent experiences are more important than less recent experiences. For example, the price movement of last week is generally more important than price movement of last month when the agent attempts to predict tomorrow price. Therefore, each mini-batch of experience was allocated a weight which enlarges or dismisses its probability of being picked. If n_b is batch size, then the i^{th} mini-batch in replay buffer is defined as:

$$B_i = [e_i, e_{i+1}, e_{i+2} \dots \dots e_{i+n_b-1}] \quad (34)$$

As replay buffer follows first in first out principle, the first mini-batch in replay buffer is a set of least recent experiences and the last one is a set of most recent experiences. Then the corresponding weight of i^{th} mini-batch is

$$w_i = s^i \tag{35}$$

s is the sample bias which determines the tendency of model selecting recent experiences. The probability mass function can be expressed as:

$$\mathrm{P}(B_i) = \frac{s^i}{\sum_{j=1}^{Buffer\ size - Batch\ size} s^j} \tag{36}$$

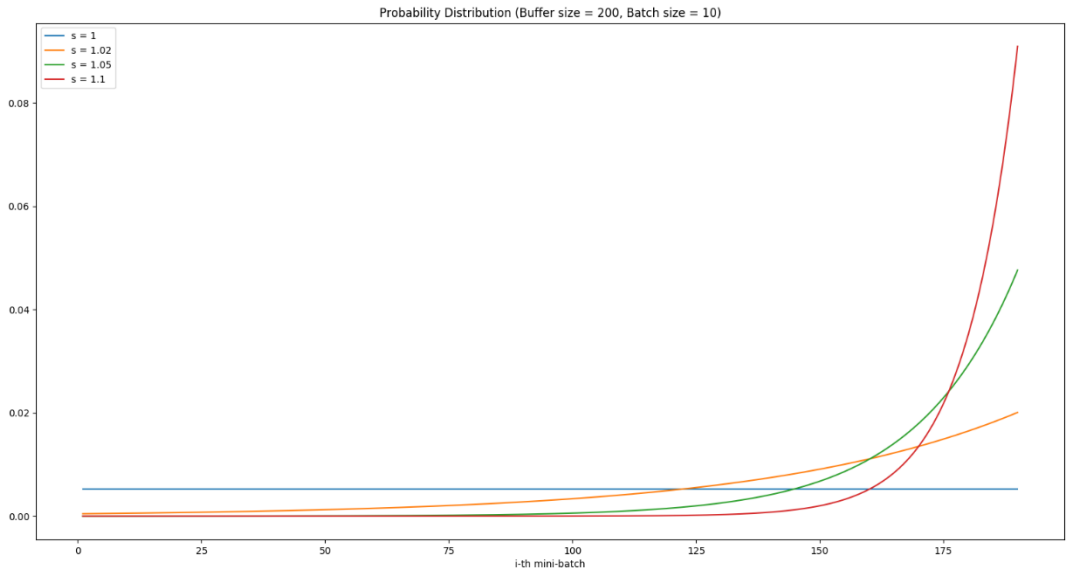

Figure 16 – Probability Distribution of picking a mini-batch

$s = 1$ implies every mini-batch shares the same probability of being picked. The sample bias is usually defined in a range of 1 to 1.15, which a value above causes the model to overly bias the latest data and degrade the overall performance eventually.

4.5. Benchmarking

To measure the performance, the profit produced by the proposed models should be compared with certain benchmarks. In this experiment, the following benchmarks were proposed.

4.5.1. Uniform Constant Rebalanced Portfolio (UCRP)

This is the strategy that keeps the uniform distribution of wealth among a set of stocks from day to day [15]. For example, if a portfolio comprises ten stocks, one-tenth of the capital is kept being allocated to each stock on a daily basis.

4.5.2. Uniform Buy and Hold Portfolio (UBHP)

This is the strategy that purchase the assets uniformly at the start of the trading period, and sell all the assets at the end of trading period. This is similar to Uniform Constant Rebalanced Portfolio except it does not recontribute any capital once the trading period starts.

4.5.3. Robust Median Reversion (RMR)

This strategy makes optimal portfolios based on the improved reversion estimation. Many

experiment on real market has shown that RMR can produce generally better performance than other mean reversion strategy [16]. Furthermore, this algorithm runs in linear time, and thus is suitable for large-scale trading applications.

4.5.4. On-line portfolio selection with moving average reversion (OLMAR)

This strategy exploits moving average reversion by unitizing online learning techniques. It is expected that this algorithm can overcome the problem of state of art caused by the single period mean reversion and achieve reasonable result in real markets. [17]

4.5.5. Passive aggressive mean reversion strategy (PAMR)

This strategy exploits both mean reversion relation of markets and powerful online passive aggressive learning technique in machine learning. The main idea is to establish a new loss function which is able to effectively utilize the mean reversion property, and then use passive aggressive online learning method to select a best portfolio among a group of assets to optimize the cumulative return. [18]

4.5.6. Anti-correlation (Anticor)

This strategy makes use of the negative autocorrelation and consistency of positive lagged cross-correlation to maintain the portfolio. It is a non-universal portfolio selection algorithm which does not aim to predict a winner among a pool of asset. This approach focuses on predictable statistical relations between all pairs of stocks in the market. [19]

4.5.7. Source

For ease of implementation, the effect of transaction cost was not included for all benchmark strategies above. The strategy of RMR, OLMAR, PAMR and Anticor were generated by universal-portfolios python package. It is the collection of different online portfolio selection algorithms. Details can be found on
https://github.com/Marigold/universal-portfolios

4.6. Price prediction model

The proposed model in this research was integrated with a price prediction model in order to enhance the overall profitability.

4.6.1. Model configuration

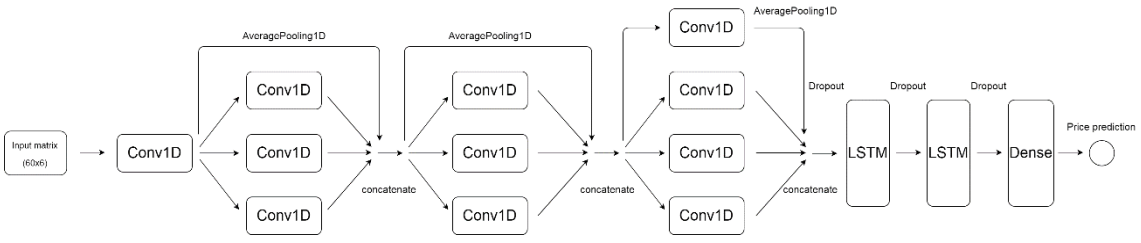

Figure 17 – Implementation of price prediction model

The price prediction model is composed of 1D convolution layers, LSTM layers and fully connected layers. The detail hyper-parameters of each layer are stated in Appendix. The input tensor has a size of 60x6, which implies it has a sliding window size of 60 days for each set of input and there are 6 financial indicators which are Open, High, Low, Close, Adjusted close and volume. Firstly, the Adjusted close and volume were normalized to range of 0 – 1 throughout experiment period which includes training, validating and testing. Then OHLC was normalized to range 0.1 – 0.9 within each sliding window. The output (label) of this model was logarithm change of next day close price. In other words, it was a next-1-day stock price forecast which made use of last 60 days data. Additionally, the training, validating and training period were defined exactly same as proposed portfolio selection model, but this model trained with almost every stock in S&P500.

4.6.2. Price prediction vector

The price prediction vector f_t is defined as predicted daily change of price of all equites held in a portfolio at time t. Since outputs from price prediction model is logarithm change r_i', the price prediction vector can be expressed as:

$$f_t = \left(e^{r_0'}, e^{r_1'}, \ldots\ldots, e^{r_{14}'}\right) \tag{37}$$

Therefore, each element in f_t representing a daily change prediction of a stock in portfolio. Then f_t was multiplied to the voting score layer to adjust the action generated by portfolio selection model.

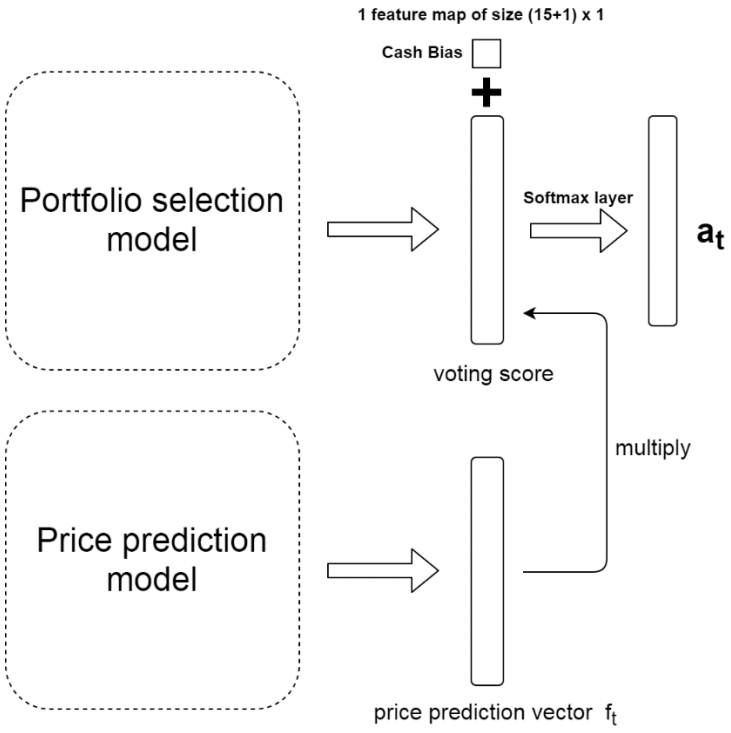

Figure 18 – Connection of portfolio selection model and price prediction model

Each voting score was multiplied by corresponding price prediction one by one. For example, if a stock is predicted to have a 5% increase in next day, the voting score of that stock is multiplied

by a factor of 1.05, and vice versa. Mathematically, a new voting score $v'_{t,i}$ of an equity at time t can be expressed as,

$$v'_{t,i} = v_{t,i} * {f_{t,i}}^{d} \tag{38}$$

where $v_{t,i}$ is the voting score of an equity directly generated from portfolio selection model, d is dependent factor which represents the tendency of final action relying on price prediction. If d is large, it implies output of portfolio selection model is mainly determined by price prediction model rather than itself. If d is zero, it indicates that model works without the aid of price prediction since all predicted values become 1 before multiplying to each voting score. Therefore, the price prediction vector can be treated as another input tensor to the network other than input price matrix X_t, last action a_{t-1}.

This implementation gives three advantages. The first one is that two models, portfolio selection model and price prediction model, can be developed and trained separately. When there is no prediction received, the price prediction vector can default to a vector which contains all one by setting dependent factor to zero, then the model works exactly same as normal. Therefore, the model can be trained and tuned without the existence of price prediction model. The second advantage is that improvement is guaranteed when directional accuracy of price prediction is high enough. For example, if the directional predictions of all holding equites of a single day are correct, by multiplying it to voting score vector, its daily return must be higher than the return generated without the aid of price prediction. Thirdly, it provides the flexibility of depending or not depending the price prediction model. Generally, performance of price prediction varies especially for downtrend period. Dependent factor d can be adjustable according to its historical performance. For instance, if price prediction model shows poor directional accuracy in last 3 months, dependent factor d can be reduced, and vice versa.

5. Experiment

Each experiment was conducted by three phases. Phase one was to provide a sufficient amount of data to train the model. Phase two was to feed the model with new data to evaluate the performance and adjust its hyper-parameters. Phase three was to evaluate model true performance, obtained profit will be compared to benchmarks defined in section 4.6. The following features were tested experimentally,

- Effect of learning rate
- Effect of cash bias
- Performance of reward functions
- Performance of models
- Performance of proposed model integrating with stock prediction

For each of experiment, cumulative return, maximum drawdown, Sharpe ratio and volatility were measured one by one against different model hyper-parameters. In this experiment, volatility is defined as standard derivation of daily return throughout a specific period.

5.1. Experimental settings

Pre-selected assets	Portfolio 1: [AAPL, AMAT, AMD, CSCO, EBAY, GLW, HPQ, IBM, INTC, KLAC, MSFT, MU, NVDA, QCOM, TXN] Portfolio 2: [AZO, BBY, DHI, F, GPS, GRMN, HOG, JWN, MAT, MCD, NKE, SBUX, TJX, TWX, YUM] Portfolio 3: [BA, CAT, CTAS, EMR, FDX, GD, GE, LLL, LUV, MAS, MMM, NOC, RSG, UNP, WM]	
Training period	From 2008-01-04 to 2014-12-31	
Validating period	From 2015-01-02 to 2015-12-31	
Testing period	From 2016-01-04 to 2017-12-31	
Episode	100 – 600	
Hyper-parameters	Value	Description
Training step	1759	Total number of iteration for each episode during training
Window size	20	Number of rows (look-back days) in input price matrix
Transaction factor	0.0025	Cost of equity transactions defined in 3.4.1
Replay Buffer size	200	Number of latest historical data available for online training
Batch size	10 / 60	Number of historical data (time-ordered) for each online training step
Learning rate	$2x10^{-6}$– $9x10^{-4}$	Parameter α (the step size) of the Adam optimization
Rolling steps	0 – 30	Number of online training steps in validating period and testing period
Cash bias	0 – 50	Tendency of holding cash
Sample bias	1 – 1.1	Tendency of picking recent historical data for online training

Table 3 – General Preliminary setting of experiments

As defined in section 3.3.1, three different portfolios which are high-beta only, low-beta only and high-low beta portfolios were tested. Beta value greater than one is regarded as high and vice versa. The beta of equity is based on Yahoo Finance, it is a monthly price relative to the monthly price change of the S&P500. The time period for beta is 3 years (36 months) which exactly covers the whole validating and testing period.

Some preliminary tests regarding batch size and sliding window size were conducted before this experiment. As mentioned in section 3.4.3, window size of 10, 20, 40, 60 were tested, the same set of number was tested for batch size as well. As a result, 16 combinations were tried, normally, smaller size of either window or batch achieved better performance and convergence

of neural network. On the contrary, larger window or batch size took more iterations to converge the weights. Since agent redistributes the assets daily, a short-term look-back period is good enough to make a decision. In fact, the information contained in a 20-days sliding window is more than just 20 days, as technical indicators like MACD, moving average and Sharpe ratio are included. Thus, by considering time-efficiency, a size of 10 and 20 were selected for batch and sliding window respectively in general case. Though, a batch size of 60-days was selected when reward function was Sharpe ratio or Calmar ratio defined in section 4.3. As these financial indicators usually require a longer period of time to be estimated especially for its risk component.

Additionally, some hyper-parameters are specified in a range, as it varied according different models or portfolios. For example, CNN model tended to iterate more episodes than other models to converge in training. Learning rate, rolling steps, cash bias and sample bias were tuned in validating period. Unless specified otherwise, reward function of all tests is average logarithmic cumulative return.

5.2. Effect of learning rate

This section demonstrates how the learning rate of the model affects the trading strategy and performance. The test was conducted on Portfolio 1 with CNN model, all hyper-parameters were remained the same except learning rate.

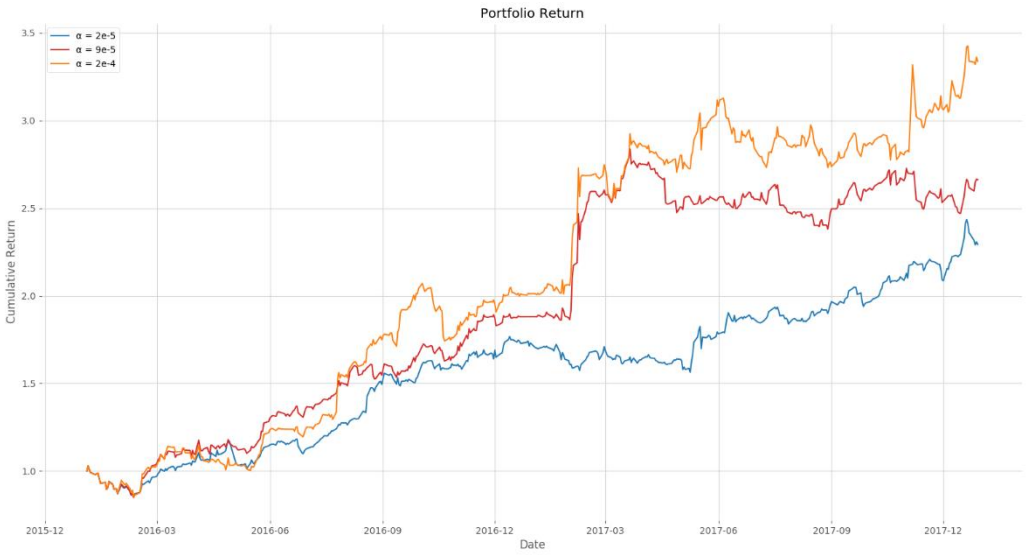

Figure 19 – Portfolio 1: Cumulative return with different learning rates (CNN model)

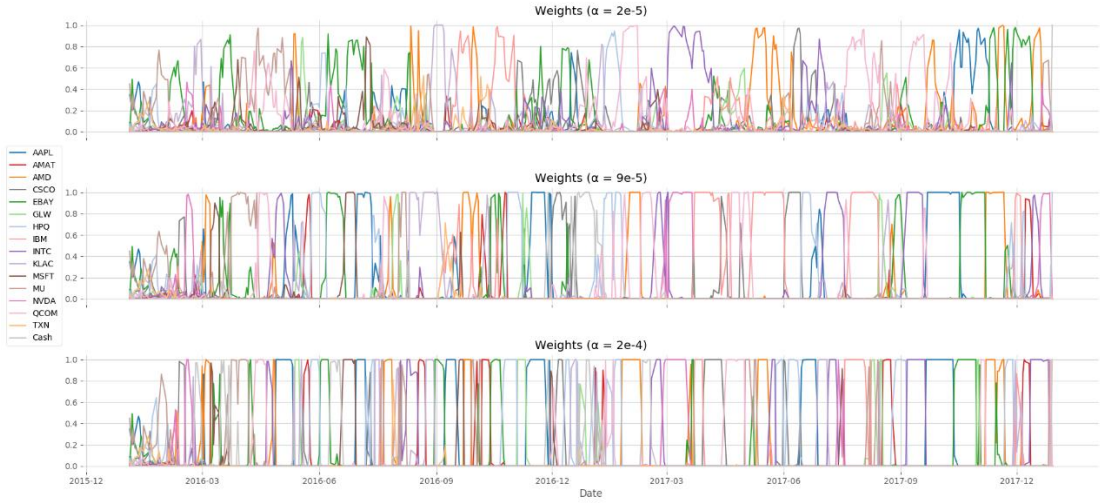

Figure 20 – Portfolio 1: Daily asset allocation with different learning rates (CNN model)

Learning rate	Cumulative Return	Max Drawdown	Sharpe Ratio	Volatility
$2x10^{-5}$	2.293929	**0.171695**	2.226310	**0.014181**
$9x10^{-5}$	2.662603	0.175658	**2.323822**	0.016448
$2x10^{-4}$	**3.338673**	0.173141	2.307035	0.020945

Table 4 – Portfolio 1: Performance of CNN model with different learning rates

The result demonstrated that the model generally achieved better performance for larger learning rate. During beginning of February 2017, the model with larger learning had established a strategy which rose the cumulative return dramatically in only few days. Nonetheless, small learning rate obtained the lowest maximum drawdown and volatility which are one of its benefits. Furthermore, from the daily assets allocation, model with smaller learning rate tended to hold two or three stocks in each interval. In contrast, model with larger learning rate only held one stock for most of the time. It can be concluded that learning rate directly relates to the aggressiveness of trading strategy, a large learning rate implies a strategy of high-risk high-return and vice versa.

5.3. Effect of cash bias

This section demonstrates how the cash bias defined in section 4.2.5 affects the trading strategy established by model. The back-test was conducted on Portfolio 2 with CNN + Dense model, five different values of cash bias were selected.

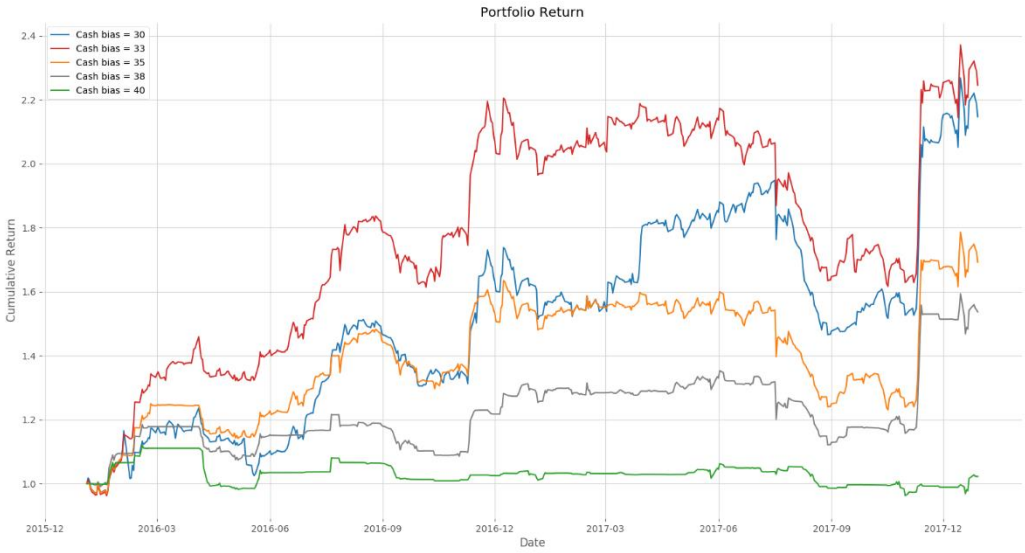

Figure 21 – Portfolio 2: Cumulative return with different cash bias (CNN + Dense model)

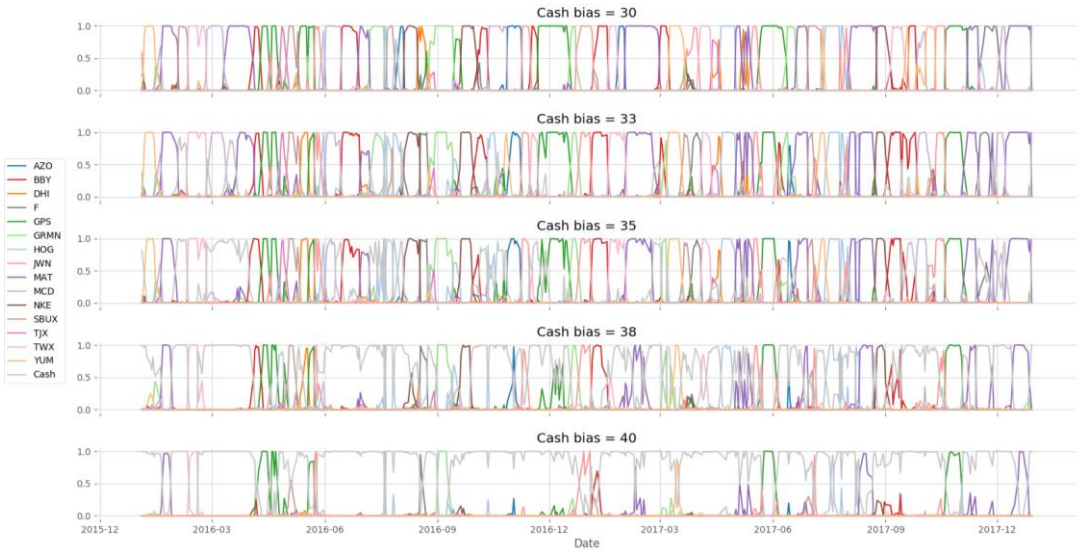

Figure 22 –Portfolio 2: Daily asset allocation with different cash bias (CNN + Dense model)

Cash Bias value	Cumulative Return	Max Drawdown	Sharpe Ratio	Volatility
30	2.146788	0.248056	1.439522	0.019876
33	**2.245852**	0.267065	**1.602876**	0.019106
35	1.692419	0.247403	1.016219	0.017710
38	1.536791	0.171713	0.931851	0.014693
40	1.022254	**0.141351**	-0.707669	**0.006364**

Table 5 – Portfolio 2: Performance of CNN + Dense model with different learning rates

The result illustrated that the cash bias value of 33 achieved the best performance which outperformed others throughout the testing period. From the daily asset allocation (Figure 22), the light grey line represents the percentage of capital distributed to cash holdings, it is observed that there was almost no tendency of holding cash when CB = 30, in contrast, the model tended to reallocate capital to cash when CB = 40. On the other hand, the horizontal movement of

portfolio return line (Figure 21) indicates that the model allocated most of the capital to cash, this pattern often happened when cash bias was large. Despite different cash bias values, all models except CB = 40 could still forecast a rising trend at the beginning of November 2017 and decide to allocate all capital into a single stock (MAT), then they obtained a huge return eventually. From Table 5, it can be concluded that model with large cash bias value often achieved best maximum drawdown and volatility. Therefore, value of cash bias directly relates to the aggressiveness of trading strategy. It also implies a reasonable tendency of holding cash can actually enhance the profitability.

Although CB = 33 obtained the highest return, in fact, CB = 30 shown a better performance than other values during validating period. Consequently, CB = 30 for Portfolio 2 with CNN + Dense model was selected to conduct back-test in later section 5.5.

5.4. Performance of reward functions

This section measures the performance of reward functions defined section 4.3 which two of them are based on cumulative return and others are based on risk-adjusted return. The back-test was conducted on Portfolio 3 with CNN + Dense model, some hyper-parameters were slightly tuned according to reward functions.

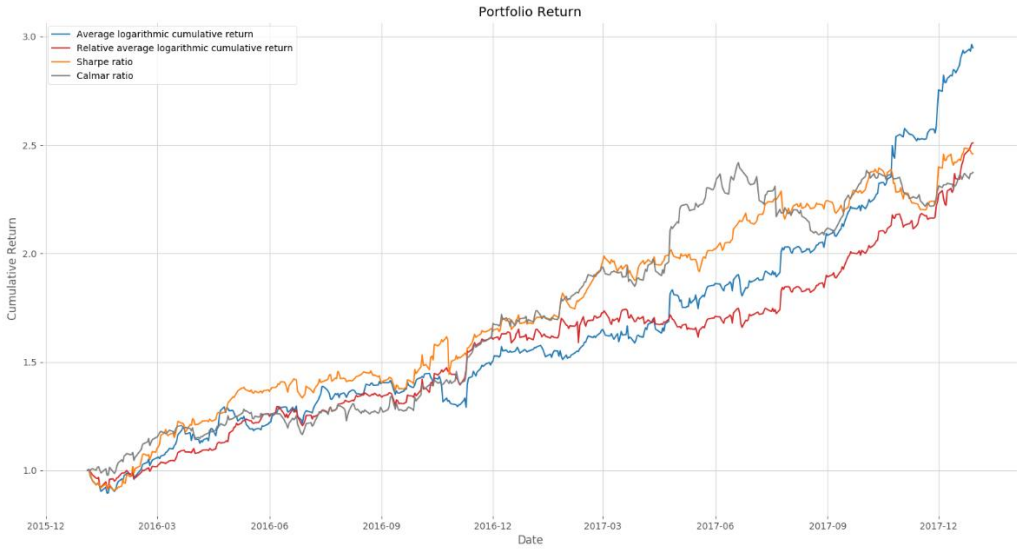

Figure 23 – Portfolio 3: Cumulative return with four reward functions (CNN + Dense model)

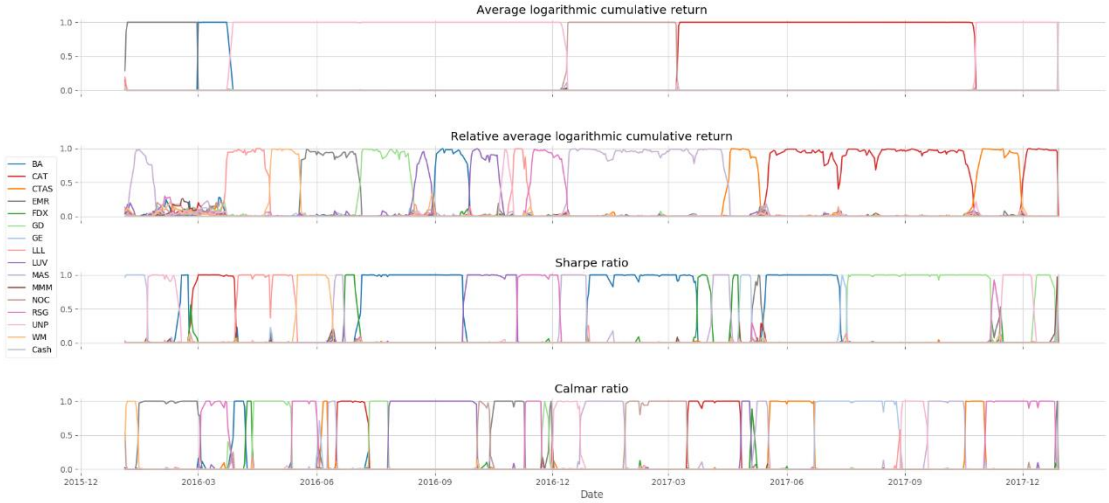

Figure 24 – Portfolio 3: Daily asset allocation with four reward functions (CNN + Dense model)

Reward function	Cumulative Return	Max Drawdown	Sharpe Ratio	Volatility
Average logarithmic cumulative return (R_1)	**2.947830**	0.109515	**3.248627**	0.013164
Relative average logarithmic cumulative return (R_2)	2.509965	**0.083060**	3.109916	**0.011443**
Sharpe ratio (R_3)	2.459217	0.107453	2.775872	0.012492
Calmar ratio (R_4)	2.373529	0.137642	2.758049	0.011999

Table 6 – Portfolio 3: Performance of CNN + Dense model with different reward functions

The results indicated that reward function of Average logarithmic cumulative return (R_1) achieved the best performance in terms of both cumulative return and Sharpe ratio, though, it did not intent to optimize the Sharpe ratio in the first place. This is because as long as the average return is large enough, the Sharpe ratio would become large regardless of its risk. In term of risk management, reward function of Relative average logarithmic cumulative return (R_2) maintained the lowest maximum drawdown and derivation of return throughout the period. It is out of original expectation since both reward function of Sharpe ratio (R_3) and Calmar ratio (R_4) did not show any outstanding performance among all reward functions. Nonetheless, reward function of Sharpe ratio and Calmar ratio kept maintaining at a higher portfolio value than others between December 2016 and September 2017. Despite the performance indicators, portfolio value generated by different models throughout the period were similar especially for the first half period, since all reward functions intended to maximize the actual return.

From the daily assets allocation, it is observed that reward function of Average logarithmic cumulative return tended to hold an individual stock for a long time, whereas other reward functions tended to switch from holding a single stock to another weekly or monthly. It is another out of original expectation which reward function of Sharpe ratio or Calmar ratio should establish

a less aggressive portfolio. One possible explanation of these patterns is that all reward functions were trained by high learning rate, as illustrated in section 5.2, it tended to establish a strategy of holding an individual stock. It implies reward function of Sharpe ratio or Calmar ratio should be trained by low learning rate instead.

5.5. Performance of models

Three tests with three portfolios were carried out to evaluate the performance of four models defined in section 4.2. All models used average logarithmic cumulative return as reward function and the best three algorithms of each performance indicators are highlighted.

5.5.1. Testing on Portfolio 1

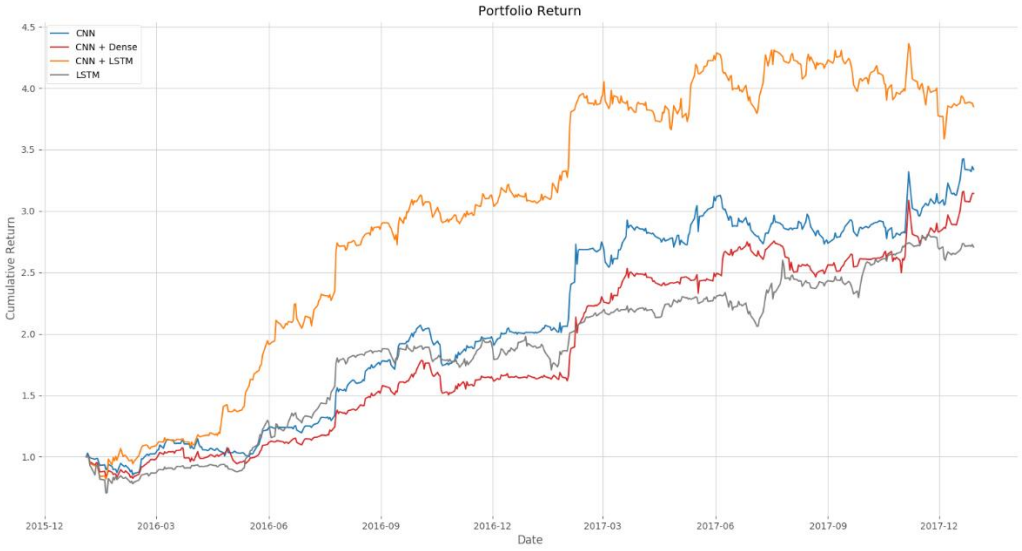

Figure 25 – Portfolio 1: Cumulative return of four models

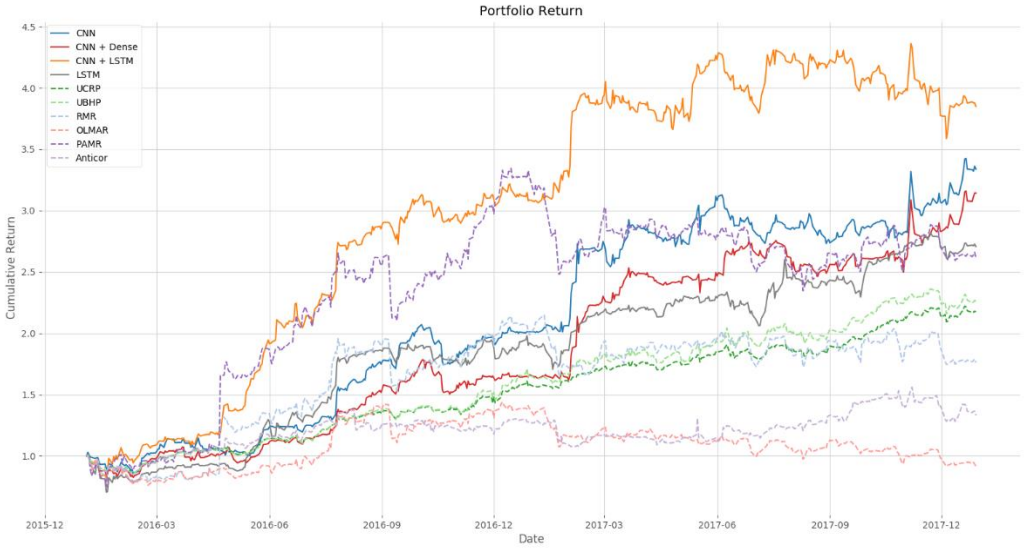

Figure 26 – Portfolio 1: Cumulative return of four models and benchmarks

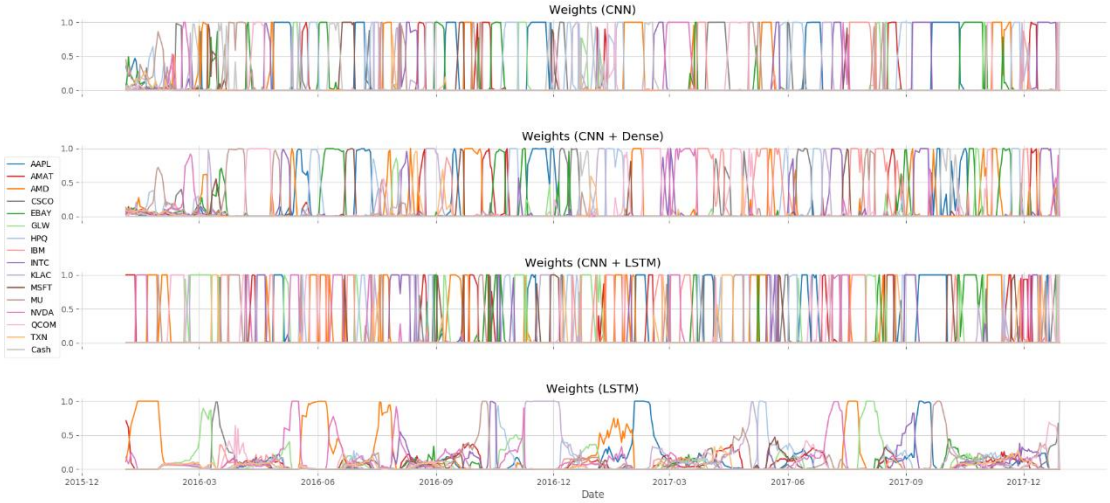

Figure 27 – Portfolio 1: Daily asset allocation of four models

Algorithm	Cumulative Return	Max Drawdown	Sharpe Ratio	Volatility
CNN	**3.338673**	**0.173141**	2.307035	0.020945
CNN + Dense	**3.142852**	0.182507	2.283164	**0.019983**
CNN + LSTM	**3.850162**	0.188245	**2.448957**	0.022329
LSTM	2.706543	0.301184	1.855366	0.020995
UCRP	2.174250	**0.159055**	**2.625713**	**0.011113**
UBHP	2.260932	**0.159269**	**2.343449**	**0.013196**
RMR	1.760933	0.229894	0.631427	0.031308
OLMAR	0.917758	0.364335	-0.371351	0.025088
PAMR	2.617286	0.298452	1.187677	0.031537
Anticor	1.330413	0.222186	0.360477	0.020128

Table 7 – Portfolio 1: Performance of four models and benchmarks

5.5.2. Testing on Portfolio 2

Figure 28 – Portfolio 2: Cumulative return of four models

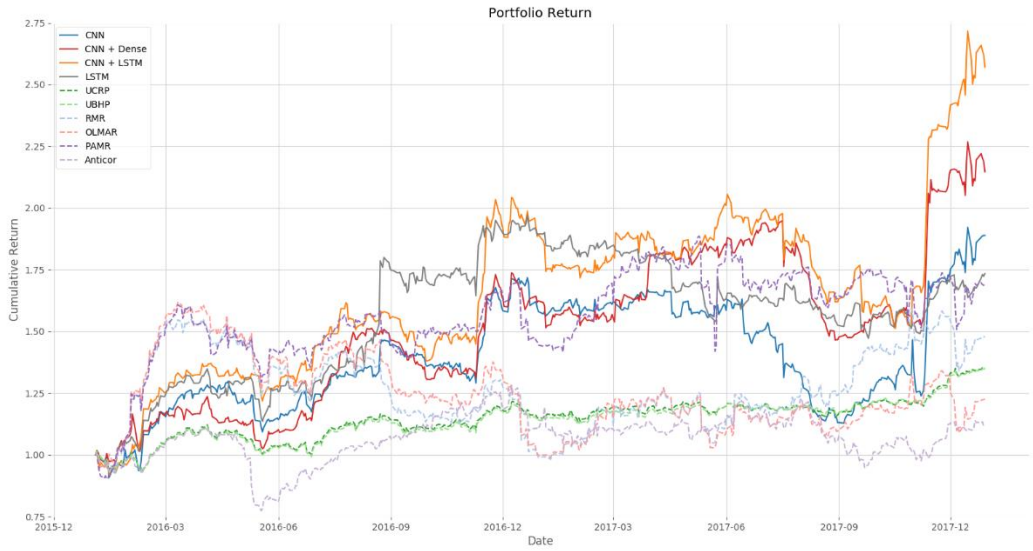

Figure 29 – Portfolio 2: Cumulative return of four models and benchmarks

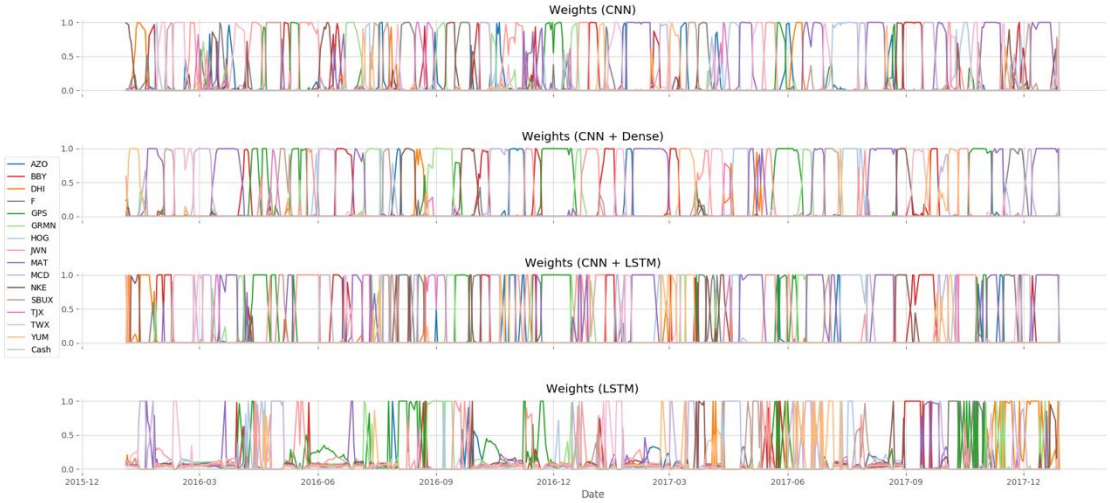

Figure 30 – Portfolio 2: Daily asset allocation of four models

Algorithm	Cumulative Return	Max Drawdown	Sharpe Ratio	Volatility
CNN	**1.889038**	0.343417	1.067797	0.021449
CNN + Dense	**2.146788**	0.248056	**1.439522**	0.019876
CNN + LSTM	**2.570882**	0.264816	**1.746760**	0.020986
LSTM	1.734209	0.253092	**1.123142**	0.016994
UCRP	1.350888	**0.107962**	0.920998	**0.008618**
UBHP	1.349108	**0.110464**	0.930146	**0.008470**
RMR	1.417715	0.250828	0.451803	0.022338
OLMAR	1.469106	0.311350	0.586893	0.019904
PAMR	1.691297	**0.247167**	0.804223	0.022342
Anticor	1.119102	0.303191	-0.027792	**0.016689**

Table 8 – Portfolio 2: Performance of four models and benchmarks

5.5.3. Testing on Portfolio 3

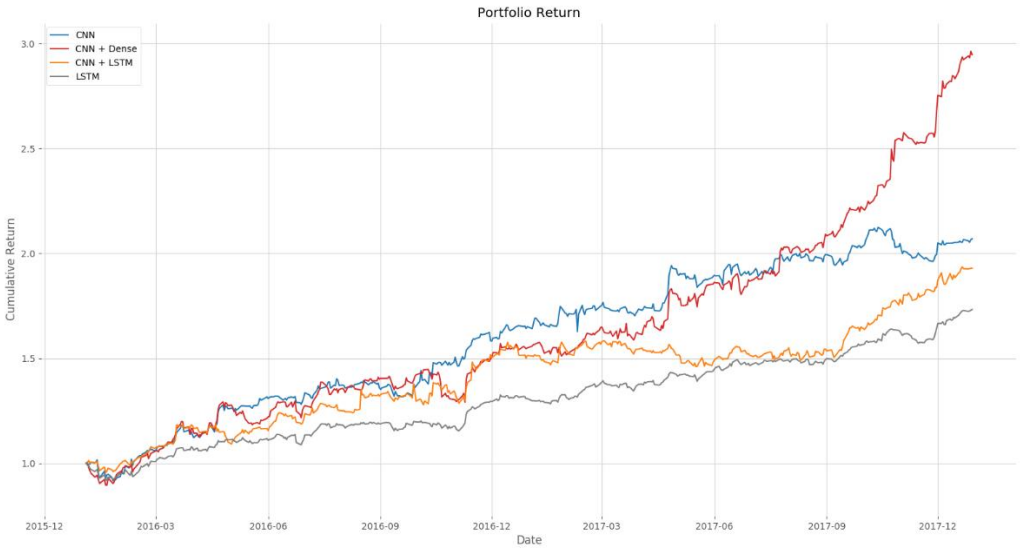

Figure 31 – Portfolio 3: Cumulative return of four models

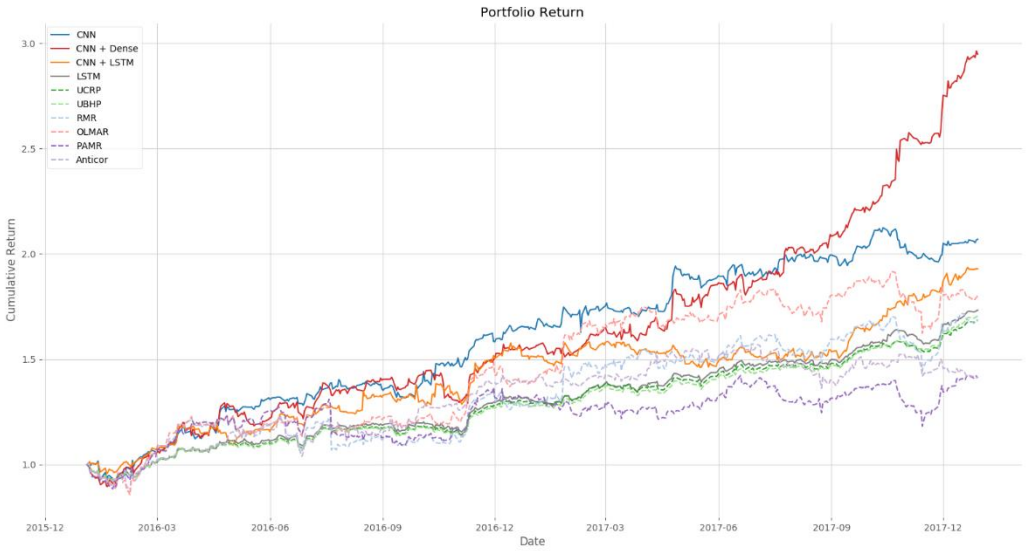

Figure 32 – Portfolio 3: Cumulative return of four models and benchmarks

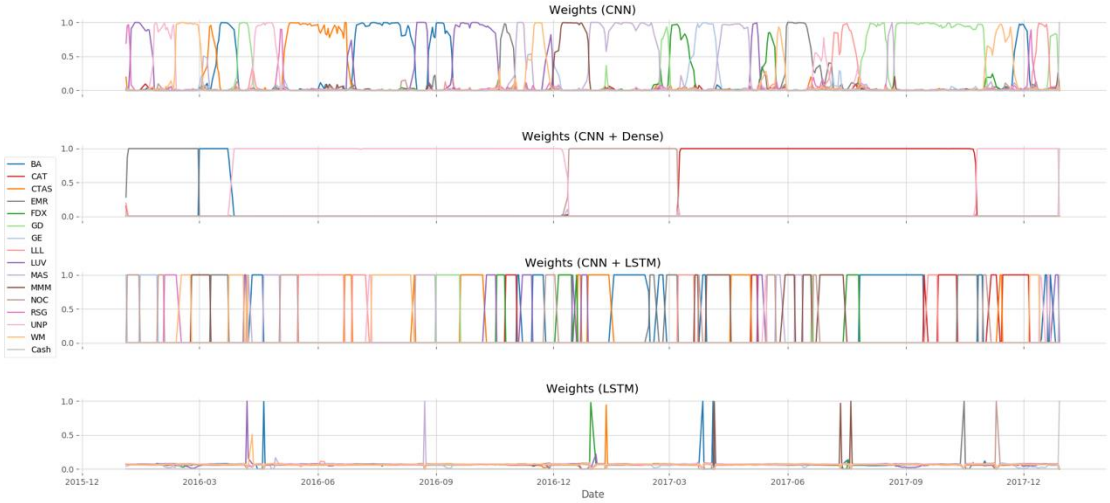

Figure 33 – Portfolio 2: Daily asset allocation of four models

Algorithm	Cumulative Return	Max Drawdown	Sharpe Ratio	Volatility
CNN	**2.069861**	0.098267	2.276373	0.011853
CNN + Dense	**2.947830**	0.109515	**3.248627**	0.013164
CNN + LSTM	**1.929472**	**0.081247**	2.094673	0.011385
LSTM	1.732735	**0.081857**	**2.634515**	**0.007230**
UCRP	1.683735	0.082040	2.499845	**0.007108**
UBHP	1.705367	**0.081804**	**2.593881**	**0.007070**
RMR	1.689991	0.152055	1.183263	0.015156
OLMAR	1.801414	0.152818	1.396539	0.014882
PAMR	1.409906	0.169383	0.731261	0.013464
Anticor	1.418605	0.159973	0.875281	0.011563

Table 9 – Portfolio 3: Performance of four models and benchmarks

5.5.4. Discussions

From the results, it was highlighted that the proposed models normally achieved a satisfactory result compared to benchmarks defined in section 4.5. From back-test on Portfolio 1, CNN + LSTM model almost outperformed all other algorithms throughout entire testing period while CNN model had a slightly better performance than CNN + Dense model. From back-test on Portfolio 2, CNN + LSTM model earned a highest portfolio profit again. Noted that all CNN models reacted to a sharp rising trend of a single stock (MAT) at the beginning of November 2017 and decided to distribute all capital into it, then earned a considerable profit eventually. From back-test on Portfolio 3, CNN + Dense model had the highest cumulative return followed by CNN model and CNN + LSTM model. It is observed that all algorithms suffered a small loss during November 2017 except CNN + LSTM model.

From all experiments, CNN, CNN + Dense and CNN + LSTM models obtained the highest

cumulative return. It reveals that the effectiveness of CNN is significantly better than solely use of LSTM layer. From the daily asset allocation, it is observed that LSTM model tended to uniformly hold all stocks for half of the time. On the other hand, CNN, CNN + Dense and CNN + LSTM models tended to hold only one individual stock and switch the position weekly or monthly for most of the time. Furthermore, the result indicated that additional layers of Dense or LSTM can actually enhance the profitability of CNN model. Generally, all proposed models cannot maintain low maximum drawdown and volatility, though, it is still slightly lower than algorithms of RMR, OLMAR and PAMR which are regarded as active trading strategies in some cases. It is mainly attributed to the use of reward function of average logarithmic cumulative return, the model did not optimize any risk but cumulative return only. Overall, in terms of both absolute return and relative return compared to UCRP, almost all proposed models generated their best results on Portfolio 1 which is a group of high beta stocks. It agrees with conclusion from previous back-test (in mid-term report) that CNN models are able to earn a higher profit when trading a high-risk portfolio. Since an aggressive strategy established by proposed model is easier to maximize its reward when all traded stocks swing more than the market.

5.6. Performance of proposed model integrating with price prediction

The methodology of integrating with price prediction model defined in section 4.6 was tested based on Portfolio 2. The price prediction was firstly trained with stocks within S&P500 between 2008-01-04 and 2014-12-31, validated between 2015-01-02 and 2015-12-31. Next, the daily price of stocks in portfolio was predicted between 2016-01-04 and 2017-12-31. These results were then fed to voting score layer of proposed model in order to further adjust the portfolio selection vector. In addition, different dependent factors defined in section 4.6 were tested as well, it is a variable which determines the dependency on price prediction model.

Stock	Correct prediction	Directional accuracy	Mean absolute error (10^{-4})
AZO	285	0.568862	0.1214
BBY	288	0.57485	0.0983
DHI	250	0.499002	1.449
F	284	0.566866	0.147
GPS	299	0.596806	0.0132
GRMN	277	0.552894	0.1208
HOG	270	0.538922	0.2717
JWN	279	0.556886	0.1854
MAT	289	0.576846	0.0398
MCD	284	0.566866	0.3402
NKE	292	0.582834	0.0835
SBUX	281	0.560878	0.1824
TJX	275	0.548902	0.5307

TWX	290	0.578842	0.0218
YUM	282	0.562874	0.1044

Table 10 – Portfolio 2: Performance of price prediction model

Figure 34 – Portfolio 2: Cumulative return of proposed model integrating with price prediction

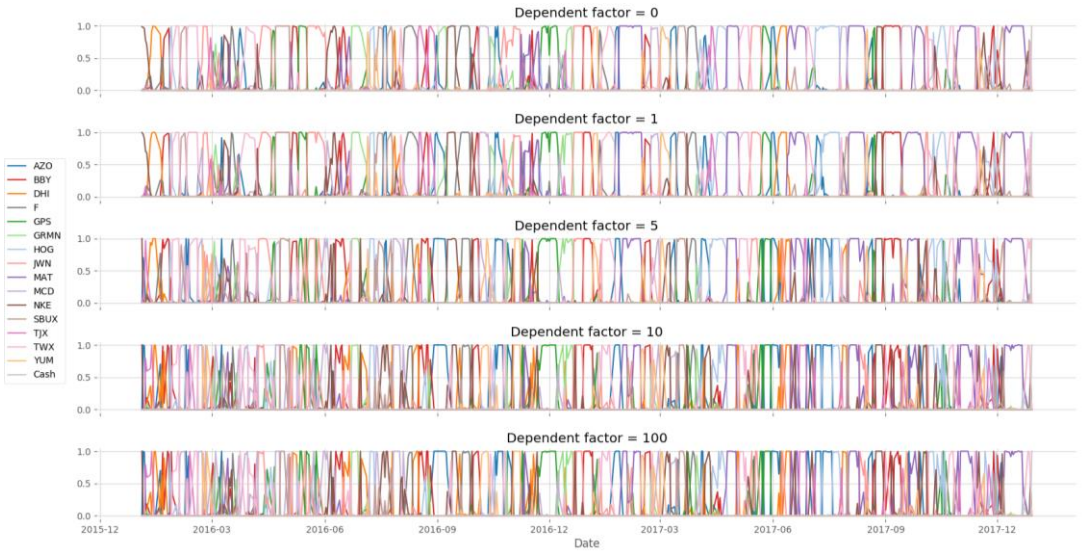

Figure 35 – Portfolio 2: Daily asset allocation of proposed model integrating with price prediction

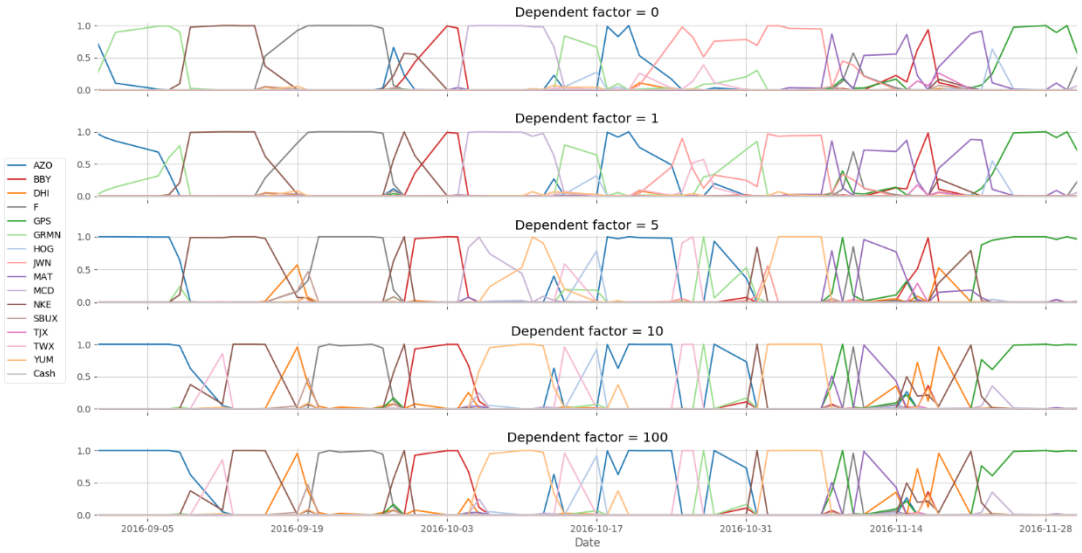

Figure 36 – Portfolio 2: Daily asset allocation of proposed model integrating with price prediction (enlarged)

Dependent factor	Cumulative Return	Max Drawdown	Sharpe Ratio	Volatility
0	1.889038	0.343417	1.067797	0.021449
1	2.199079	0.295325	1.332491	0.022278
5	**2.278529**	0.191537	**1.435246**	0.021787
10	1.980732	**0.222784**	1.269283	0.019711
100	0.701856	0.563673	-1.357165	**0.015685**

Table 11 – Portfolio 2: Performance of proposed model integrating with price prediction

The prediction model generally achieved 56% directional accuracy which might be slightly lower than performance obtained by my groupmate Garrett previously. The main reason is the hyper-parameters of price prediction model were slightly adjusted when integrating with this proposed model, such that the input length was changed from 120days to 60days. It is because the price prediction model was aimed to forecast close price after 1, 3 and 5 trading days in the first place. However, the 3-days or 5-days prediction may not be suitable for proposed model in this research as it reallocates all capital on a daily basis. In order to strengthen price prediction model to 1day forecast, the input length was decided to decrease.

From the overall performance, it can be concluded that the final portfolio value could be improved with an appropriate dependency on price prediction. Noted that dependent factor of 0 implies that the proposed model operated without price prediction model, it served as a benchmark to compare with other dependent factors. The dependent factor of 5 achieved the best cumulative return, whereas the performance degraded when dependent factor kept increasing. From the daily asset allocation, apparently, the portfolio selection vector with different dependencies on stock prediction were similar. Yet, there were certain differences when taking a closer look at each day, a larger dependent factor shows a significant difference significantly. For example, in the middle of October 2016 (from 2016-10-07 to 2016-10-12, Figure 36), the model without the aid of price

prediction (dependent factor = 0) held MCD (light purple line) only, in contrast, other models tended to hold YUM (light yellow line) when dependent factor increased. Eventually, this adjustment by price prediction helped the model to earn more return than before within this week. This is an example illustrating how integration of price prediction can enhance the overall profitability, nonetheless, it is also possible that the price prediction might worsen the overall profitability because of its poor accuracy especially in downturn period.

Therefore, the dependent factor should be varied smartly according to its past performance. For instance, the dependent factor should increase when its directional accuracy appears satisfied in last three months, and vice versa. This smart integration between proposed model and price prediction model is one of the future work to explore.

6. Conclusion

In present stage, the main contribution of this work is to evaluate the feasibility of adopting convolution layers, LSTM layers and fully connected layers for analyzing financial data. And the concept of online reinforcement learning was applied to portfolio management task.

This research built a portfolio management framework to produce a portfolio selection vector on a daily basis in order to optimize overall return. The procedures of this framework were as follows: First, a set of fundamental and technical indicators within a sliding window were normalized such that an input price matrix was constructed. Second, the input price matrix was computed by a policy network. Third, action from policy network was fed into reward function, and corresponding experience was added to replay buffer memory. Fourth, trainable parameters of policy network were updated by Deterministic Policy Gradient Algorithms (DPG) through experience replay.

The performance of different learning rates, cash bias, reward functions, models and portfolios was tested by four experiments. Firstly, the result indicated that learning rate and cash bias directly relates to the aggressiveness of trading strategy established by model. A large learning rates and small cash bias are likely to constitute an aggressive strategy which tends to distribute all capital to an individual asset, whereas a small learning rate and large cash bias tended to make a less aggressive strategy which holds two or more assets (including cash holding). Secondly, reward function of Average logarithmic cumulative return achieved the best performance among all proposed reward functions. The reward functions of Sharpe ratio and Calmar ratio did not obtain any outstanding performance, thus, this model might not be suitable to be optimized by risk-adjusted return. Thirdly, convolution layers with the aid of LSTM layer or fully connected layer had a better performance than other models including all benchmarks. Also, the result illustrated that all proposed model can obtain a higher portfolio value for a high beta portfolio within Information Technology sector. Furthermore, a price prediction model from other research was integrated with this model, and it demonstrated that the profitability could be improved by making use of price prediction result which was fed to voting score layer.

7. Future work

Additional work is required for improving policy network and reward function. Most of the experimental settings which obtained a good performance in this research built a high-risk strategy which tends to distribute all capital to a single asset. It is a major limitation of this research as a slightly lower-risk strategy which holds two or more stocks are preferred. Although some reward function which was supposed to optimize risk-adjusted return was attempted, it could not create a low-risk strategy effectively.

Additionally, a smart approach which integrates this model with price prediction model will be attempted in future, such that the dependency on price prediction result can be varied automatically according to recent prediction performance.

Reference

[1] M. Volodymyr, K. Kavukcuoglu, S. David, V. Joel and A. Graves, "Human-level control through deep reinforcement," *NATURE*, p. 531, 2015.

[2] O. Jin and H. El-Saawy, "Portfolio Management using Reinforcement Learning," in *Stanford University*.

[3] T. P. Lillicrap, J. J. Hunt, A. Pritzel and N. Heess, "CONTINUOUS CONTROL WITH DEEP REINFORCEMENT," *ICLR*, 2016.

[4] D. Silver and G. Lever, "Deterministic Policy Gradient Algorithms," *PMLR*, 2014.

[5] S. Qiu and B. Cai, "Flexible Rectified Linear Units for Improving Convolutional Neural Networks," in *South China University of Technology*, China, 2017.

[6] F. Beaufays, H. Sak and A. Senior, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," *Proc. Interspeech*, 2014.

[7] J. Wu, "Introduction to Convolutional Neural Networks," in *Nanjing University*, China, 2017.

[8] T. Aamodt, "Predicting Stock Markets with Neural Networks," in *University of Oslo*, 2015.

[9] Z. Jiang and J. Liang, "Cryptocurrency Portfolio Management with Deep Reinforcement Learning," in *Xi'an Jiaotong-Liverpool University*, 2016.

[10] INVESTOPEDIA, "What is the ideal number of stocks to have in a portfolio?," INVESTOPEDIA, New York, NY, 7 2005. [Online]. Available: https://www.investopedia.com/ask/answers/05/optimalportfoliosize.asp. [Accessed 12 2017].

[11] J. Krohnfeldt, "3 Reasons Cash Is a Smart Position in Your Portfolio," INVESTOPEDIA, New York, NY, 7 2016. [Online]. Available: https://www.investopedia.com/articles/investing/072316/3-reasons-cash-smart-position-your-portfolio.asp. [Accessed 3 2018].

[12] Saud Almahdi and Steve Y. Yang, "An adaptive portfolio trading system: A risk-return portfolio optimization using recurrent reinforcement learning with expected maximum drawdown," *Expert Systems With Applications*, pp. 267-279, 2017.

[13] T. Schaul, J. Quan, I. Antonoglou and D. Silver, "PRIORITIZED EXPERIENCE REPLAY," *ICLR*, 2016.

[14] T. d. Bruin, J. Kober, K. Tuyls and R. Babuska, "The importance of experience replay database," Delft Center for Systems and Control Delft University of Technology, 2015.

[15] A. Kalai and S. Vempalay, "Efficient Algorithms for Universal Portfolios," *Journal of Machine Learning Research*, pp. 423-440, 2002.

[16] D. Huang, J. Zhou and B. Li, "Robust Median Reversion Strategy for On-Line Portfolio Selection," *Twenty-Third International Joint Conference,* 2006.

[17] B. Li and S. C. H. Hoi, "On-Line Portfolio Selection with Moving Average Reversion," *International Conference on Machine Learning,* 2012.

[18] Bin Li, Peilin Zhao, Steven C. H. Hoi and Vivekanand Gopalkrishnan, "PAMR: Passive aggressive mean reversion strategy," *Mach Learn,* p. 221, 21 2 2012.

[19] Allan Borodin, Ran El-Yaniv and Vincent Gogan, "Can We Learn to Beat the Best Stock," University of Toronto, 2005.

[20] N. Y. S. Exchange, "New York Stock Exchange Price List 2017," 2017. [Online]. Available: https://www.nyse.com/publicdocs/nyse/markets/nyse/NYSE_Price_List.pdf. [Accessed 2017].

Appendix A. Details of Portfolio

GICS Sector	Ticker symbol	Security
Information Technology	AAPL	Apple Inc.
(Portfolio 1)	AMAT	Applied Materials Inc.
	AMD	Advanced Micro Devices Inc
	CSCO	Cisco Systems
	EBAY	eBay Inc.
	GLW	Corning Inc.
	HPQ	HP Inc.
	IBM	International Business Machines
	INTC	Intel Corp.
	KLAC	KLA-Tencor Corp.
	MSFT	Microsoft Corp.
	MU	Micron Technology
	NVDA	Nvidia Corporation
	QCOM	QUALCOMM Inc.
	TXN	Texas Instruments
Consumer Discretionary	AZO	AutoZone Inc
(Portfolio 2)	BBY	Best Buy Co. Inc.
	DHI	D. R. Horton
	F	Ford Motor
	GPS	Gap Inc.
	GRMN	Garmin Ltd.
	HOG	Harley-Davidson
	JWN	Nordstrom
	MAT	Mattel Inc.
	MCD	McDonald's Corp.
	NKE	Nike
	SBUX	Starbucks Corp.
	TJX	TJX Companies Inc.
	TWX	Time Warner Inc.
	YUM	Yum! Brands Inc
Industrials	BA	Boeing Company
(Portfolio 3)	CAT	Caterpillar Inc.
	CTAS	Cintas Corporation
	EMR	Emerson Electric Company
	FDX	FedEx Corporation

	GD	General Dynamics
	GE	General Electric
	LLL	L-3 Communications Holdings
	LUV	Southwest Airlines
	MAS	Masco Corp.
	MMM	3M Company
	NOC	Northrop Grumman Corp.
	RSG	Republic Services Inc
	UNP	Union Pacific
	WM	Waste Management Inc.

Table 12 – Details of portfolio tested in experiment

Appendix B. Python Codes used in experiment

1. DPG.py

It defines a reinforcement learning environment and iterates steps in training, validating, testing period.

```
import numpy as np
import tensorflow as tf
import pickle
from Portfolio import Portfolio
from ReplayBuffer import ReplayBuffer
from ActorNetwork import ActorNetwork
from sys import exit
import os
import Data
import random
import pandas as pd
import math

def gpu_settings():
    os.environ["CUDA_VISIBLE_DEVICES"] = '1' #use GPU with ID 4
    config = tf.ConfigProto()
    config.gpu_options.allow_growth = True
    config.gpu_options.per_process_gpu_memory_fraction = 0.3
    sess = tf.Session(config=config)
    from keras import backend as K
    K.set_session(sess)
    K.set_learning_phase(1)
    return sess

def load_weights(mode, actor, ep):
    print("Now loading the weight")
    try:
        if mode == "train" or mode == "valid":
            actor.model.load_weights("./model/CNN+dense.h5")
            # actor.model.load_weights("./model/actormodel_{}_ep{}.h5".format("train", ep-1))
            # print("Loaded actormodel_{}_ep{}.h5".format("train", ep-1))
        elif mode == "test":
            actor.model.load_weights("./model/actormodel_{}_ep{}.h5".format("valid", ep-1))
            print("Loaded actormodel_{}_ep{}.h5".format("valid", ep-1))
        else:
            raise Exception("Invalid mode")
        print("Loaded weight successfully ({})".format(ep-1))
    except:
        print("Cannot find the weight")

def save_weights_results(P, mode, actor, ep, performance):
    print("Now we save model")
    if mode == "train":
        actor.model.save_weights("./model/actormodel_{}_ep{}.h5".format(mode, ep),
overwrite=True)
        print("Saved actormodel_{}_ep{}.h5".format(mode, ep))
        filename = 'Portfolio_{}_{}_{}_{}_{}_ep{}.pickle'.format(mode,P.start,P.end,'-
'.join(P.tickers), performance, ep)
    else:
        if mode == "valid":
            actor.model.save_weights("./model/actormodel_{}_ep{}.h5".format(mode, ep-1),
overwrite=True)
            print("Saved actormodel_{}_ep{}.h5".format(mode, ep-1))
        filename = 'Portfolio_{}_{}_{}_{}_{}_ep{}.pickle'.format(mode,P.start,P.end,'-
'.join(P.tickers), performance, ep-1)

    with open("./result/{}".format(filename),"wb") as outfile:
        pickle.dump(P, outfile)
        print("Saved {}".format(filename) )

def save_epsoide_performance(mode, start, end, epsiode_reward):
    filename = 'Epsiode_Performance_{}_{}_{}'.format(mode,start,end)
    with open("./result/{}.pickle".format(filename),"wb") as outfile:
        pickle.dump(epsiode_reward, outfile)
```

```
        print("saved {}".format(filename))

def prepare_tickers(train_mode, mode, tickers, sector, start, end):

    # train mode 1: 10% of actual portfolio is replaced by other stocks
    # train mode 2: completely random portfolio
    # train mode 3: non random / noise portfolio

    if mode == "train":

        if train_mode == 1:
            print("Construct a portfolio with noise ({})".format(sector))
            rand_sample_num = int((len(tickers))*0.1)
            rand_tickers_num = np.random.randint(0,len(tickers),rand_sample_num)

            # create portfolio with noise (not actual one)
            train_tickers = tickers[:]

            if not
os.path.exists("ticker/sp500tickers_{}_{}_{}.pickle".format(sector,start,end)):
                Data.generate_list(sector,start,end)
            with open("ticker/sp500tickers_{}_{}_{}.pickle".format(sector,start,end),"rb") as
file:
                # randomly sample from sp500 but excluding original tickers
                rand_sample = random.sample([symbol for symbol in pickle.load(file) if symbol
not in tickers], rand_sample_num)
                for ix in range(rand_sample_num):
                    train_tickers[rand_tickers_num[ix]] = rand_sample[ix]
            return train_tickers

        elif train_mode == 2:
            print("Construct a random portfolio ({})".format(sector))
            if not
os.path.exists("ticker/sp500tickers_{}_{}_{}.pickle".format(sector,start,end)):
                Data.generate_list(sector,start,end)
            with open("ticker/sp500tickers_{}_{}_{}.pickle".format(sector,start,end),"rb") as
file:
                train_tickers = random.sample(pickle.load(file), len(tickers))
            return train_tickers

        else:
            return tickers

    else:
        return tickers

def prepare_states(P,window_size):
    state = []
    df = np.array([P.df_close.values, P.df_open.values, P.df_high.values, P.df_low.values,
                   P.df_volume.values, P.df_roc.values, P.df_macd.values, P.df_ma5.values,
                   P.df_ma10.values, P.df_ema20.values, P.df_sr20.values], dtype='float')

    # if 1-1-2013 is the first day of trading, last 60 days (not include today) is the input for
first day
    for j in range( P.start_index-1 , len(df[0])):
        temp = np.copy(df[:, j-window_size+1:j+1 , :])

        # to normalize the data
        # latest adjusted close price within sliding window
        # price_base = np.copy(temp[0,-1,:])
        for feature in [0,1,2,3,4,7,8,9]:
            for k in range(P.tickers_num):
                if temp[feature,-1,k] == 0:
                    temp[feature,:,k] /= temp[feature,-2,k]
                else:
                    temp[feature,:,k] /= temp[feature,-1,k]
        state.append(temp)
    return state

def experience_replay(actor, buff, BATCH_SIZE, mode,  state, step, future_price, last_action,
cash_bias, prediction,
                      train_rolling_steps, test_rolling_steps, sess):

    # define the number of experience replay in each step
```

```
    if mode == "train":
        rolling_steps = train_rolling_steps
    else:
        rolling_steps = test_rolling_steps

    # Add historical data to replay buffer
    buff.add(state[step], future_price, last_action, prediction)

    # online mini-batch replay, for both test and train data
    for _ in range(rolling_steps):
        batch, current_batch_size = buff.getBatch(BATCH_SIZE)
        states = np.asarray([e[0] for e in batch])
        future_prices = np.asarray([e[1] for e in batch])
        last_actions = np.asarray([e[2] for e in batch])
        predictions = np.asarray([e[3] for e in batch])

        # to test the output between layers
        # test = sess.run(actor.test, feed_dict={
        #     actor.state: states,
        #     actor.last_action: last_actions,
        #     actor.future_price: future_prices,
        #     actor.cash_bias: np.array([[cash_bias] for _ in range(current_batch_size)]),
        #     actor.prediction: predictions
        # })


        # Train only if the buffer filled with certain data
        if step > 10 :
            actor.train(states, last_actions, future_prices, np.array([[cash_bias] for _ in
range(current_batch_size)]), predictions)

def extract_prediction():
    df = pd.read_csv('prediction_v1.csv',index_col=False)
    return df.values

def Simulate(start, end, mode='train', ep_train=30, ep_start=0):

    # sector = ["Consumer Discretionary", "Consumer Staples", "Energy", "Financials", "Health
Care", "Industrials",
    #           "Information Technology", "Materials","Real Estate","Telecommunication
Services","Utilities"]

    # group 1
    sector = "Information Technology"
    tickers = ['AAPL', 'AMAT', 'AMD', 'CSCO', 'EBAY', 'GLW', 'HPQ', 'IBM', 'INTC', 'KLAC',
'MSFT', 'MU', 'NVDA', 'QCOM', 'TXN']

    # # group 2
    # sector = "Consumer Discretionary"
    # tickers = ['AZO', 'BBY', 'DHI', 'F', 'GPS', 'GRMN', 'HOG', 'JWN', 'MAT', 'MCD', 'NKE',
'SBUX', 'TJX', 'TWX', 'YUM']
    #
    # group 3
    # sector = "Industrials"
    # tickers = ['BA', 'CAT', 'CTAS', 'EMR', 'FDX', 'GD', 'GE', 'LLL', 'LUV', 'MAS', 'MMM',
'NOC', 'RSG', 'UNP', 'WM']

    train_mode = 2
    BUFFER_SIZE = 200
    BATCH_SIZE = 10
    sample_bias = 1.05                                          # probability weighting of
[sample_bias ** i for i in range(1,buffer_size-batch_size)]
    cash_bias = 33
    dependent_factor = 5
    if mode == "train":
        LRA = 2e-5                                              # Learning rate for Actor
(training)
    else:
        LRA = 9e-4                                              # Learning rate for Actor
(testing)
    train_rolling_steps = 10
    test_rolling_steps = 0
    window_size = 20                                            # window size per input
    tickers_num = len(tickers)                                  # the number of assets (exclude
```

```
Cash)
    action_size = tickers_num + 1
    feature_num = 11                                        # number of features (adjusted
open, close, high, low)
    state_dim = (tickers_num, window_size, feature_num)     # number of assets x window size x
number of features
    epsiode_reward = []                                     # reward of each episode =
Cumulative Return by CNN / Cumulative Return by UCR
    np.random.seed(1337)
    total_step = 0

    # Tensorflow GPU optimization
    sess = gpu_settings()

    # create network and replay buffer
    actor = ActorNetwork(sess, state_dim, action_size, BATCH_SIZE, LRA)
    buff = ReplayBuffer(BUFFER_SIZE, sample_bias)

    # start simulation
    print("Portfolio Management Simualation Experiment Start ({})".format(mode))
    print("{} period: {} to {}".format(mode,start,end))

    # iterate the episode
    for ep in range(ep_start+1, ep_start+ep_train+1):

        if not (mode == "train" and ep != ep_start+1):
            load_weights(mode, actor, ep)

        train_tickers = prepare_tickers(train_mode, mode, tickers, sector, start, end)

        # construct a portfolio within defined period
        P = Portfolio(train_tickers, start, end, mode=mode)
        if not P.consistent:
            exit(0)

        # construct states
        state = prepare_states(P, window_size)
        predictions = extract_prediction()

        cumulated_return = 1

        # iterate the defined period
        for step in range(len(state)-2):

            # extract the last action
            if step == 0:
                last_action = np.array([0 for _ in range(state_dim[0])])
            else:
                last_action = np.array(P.portfolio_weights[-1][:state_dim[0]])

            prediction = np.power(predictions[step], dependent_factor)

            # generate action, single batch = batch only conisists one element
            action = actor.model.predict([state[step].reshape([1, state_dim[2], state_dim[1],
state_dim[0]]),
                                          last_action.reshape(1, state_dim[0]),
np.array([[cash_bias]]), prediction.reshape(1, state_dim[0])])

            # generate daily return
            day_return, future_price = P.calculate_return(action[0], last_action, step)

            # extract historical data to re-train the model at each time step
            experience_replay(actor, buff, BATCH_SIZE, mode,  state, step, future_price,
last_action, cash_bias, prediction,
                              train_rolling_steps, test_rolling_steps, sess)


            cumulated_return = cumulated_return *  day_return
            total_step += 1
            print("Episode {} Step {} Date {} Cumulated Return {} Day return {}".format(ep,
total_step,

P.df_normalized.index[P.start_index+step+1].strftime('%Y-%m-%d'),
```

```
cumulated_return, day_return))
            print(action[0])

        # No trading at last day, clear the portfolio and set return to 1
        P.portfolio_weights.append(np.array([0 for _ in range(tickers_num)] + [1]))

        # Generate Uniform Constant Rebalanced Portfolios strategy
        P.UCRP()
        P.UBHP()

        # calculate performance: how cumulative return outperforms Uniform Constant Rebalanced
Portfolios strategy
        performance = cumulated_return / np.array(P.portfolio_UCRP).cumprod()[-1]
        epsiode_reward.append(performance)

        # save the model and result
        save_weights_results(P, mode, actor, ep, performance)

        # clear the Portfolio and buffer
        P.clear()
        buff.erase()
        if mode == 'valid':
            total_step = 0

        print("")

    # save_epsoide_performance(mode, start, end, epsiode_reward)

    print("Finish")
    print("")

    return epsiode_reward

def train(ep_start=0, ep_end=30):
    Simulate('2008-01-04', '2014-12-31', mode="train", ep_train=ep_end-ep_start,
ep_start=ep_start)

def valid(ep_start = -1, ep_end = -1):
    if ep_end == -1:
        Simulate('2015-01-02', '2015-12-31', mode="valid", ep_train=1 , ep_start=ep_start)
    # validate performance of each episode
    else:
        print(Simulate('2015-01-02', '2015-12-31', mode="valid", ep_train=ep_end-ep_start,
ep_start=ep_start+1))

def test(ep_start=30):
    Simulate('2016-01-04', '2017-12-31', mode="test",  ep_train=1 ,  ep_start=ep_start)


if __name__ == "__main__":
    # train period: '2008-01-04' to '2014-12-31'
    # valid period: '2015-01-02' to '2015-12-31'
    # test period:  '2016-01-04' to '2017-12-31'

    train(ep_start=0, ep_end=200)
    valid(ep_start=0, ep_end=200)
    # test(ep_start=40)
```

2. ActorNetwork.py

It defines the policy networks of the model and reward functions, 5 networks and 4 reward functions are implemented in this code.

```
import numpy as np
import math
from keras.models import Model
from keras.layers import Convolution2D, Flatten, Input, concatenate, Reshape, Dense, Permute,
LSTM, TimeDistributed, multiply
from keras.layers.core import Activation
import tensorflow as tf
import keras.backend as K
from keras import regularizers


class ActorNetwork(object):
    def __init__(self, sess, state_size, action_size, BATCH_SIZE, LEARNING_RATE, ):
        self.sess = sess
        self.BATCH_SIZE = BATCH_SIZE
        self.state_size = state_size

        # Global step for decay learning rate
        self.global_step = tf.Variable(0, trainable=False, name='global_step')
        self.LEARNING_RATE = tf.train.exponential_decay(LEARNING_RATE, self.global_step,
                                                        50000, 0.1, staircase=False)
        self.transaction_factor = 0.0025

        # Sets the global TensorFlow session
        K.set_session(sess)

        # Now create the model
        self.model , self.weights, self.state, self.last_action, self.cash_bias,
self.prediction, self.test = self.CNN_Dense(state_size)

        # future price includes cash bias which is 1 at all
        self.future_price = tf.placeholder(tf.float32, [None, state_size[0]+1])

        # Reward function to be maximized
        self.reward = self.reward_1()

        # perform gradient ascending
        self.optimize = tf.train.AdamOptimizer(LEARNING_RATE).minimize(self.reward,
global_step=self.global_step)
        self.sess.run(tf.global_variables_initializer())

    def reward_1(self):
        # Average logarithmic cumulated return
        print("Reward function 1 (Average logarithmic cumulated return)")

        # transaction cost
        self.transaction_cost = 1 - tf.reduce_sum(self.transaction_factor *
tf.abs(self.model.output[:,:-1] - self.last_action), axis=1)

        return -tf.reduce_mean(tf.log(self.transaction_cost * tf.reduce_sum(self.model.output *
self.future_price, axis=1)))

    def reward_2(self):
        # Average Uniform Constant Rebalanced reward
        print("Reward function 2 (Average Uniform Constant Rebalanced reward)")

        # transaction cost
        self.transaction_cost = 1 - tf.reduce_sum(self.transaction_factor *
tf.abs(self.model.output[:,:-1] - self.last_action), axis=1)

        return -tf.reduce_mean(tf.log(self.transaction_cost * tf.reduce_sum(self.model.output *
self.future_price, axis=1) /
                                      tf.reduce_sum(self.future_price[:,:-1] /
self.state_size[0], axis=1)))

    def reward_3(self):
        # Sharpe ratio
```

```
        print("Reward function 3 (Sharpe ratio)")

        # transaction cost
        self.transaction_cost = 1 - tf.reduce_sum(self.transaction_factor *
tf.abs(self.model.output[:,:-1] - self.last_action), axis=1)

        # Daily average portfolio return
        self.average_return = tf.log(self.transaction_cost * tf.reduce_sum(self.model.output *
self.future_price, axis=1))

        # Risk free rate for sharpe ratio (Three-month U.S. Treasury bill)
        self.risk_free_rate = np.log(1.0148) / 60

        return -(tf.reduce_mean(self.average_return)-self.risk_free_rate) * \
               tf.sqrt(tf.to_float(tf.size(self.average_return))) / K.std(self.average_return)

    def reward_4(self):
        # Calmar ratio
        print("Reward function 4 (Calmar ratio)")

        # transaction cost
        self.transaction_cost = 1 - tf.reduce_sum(self.transaction_factor *
tf.abs(self.model.output[:,:-1] - self.last_action), axis=1)

        # Daily average portfolio return
        self.average_return = self.transaction_cost * tf.reduce_sum(self.model.output *
self.future_price, axis=1)

        # Cumulative return
        self.cumulative_return = tf.cumprod(self.average_return)
        self.rolling_max = tf.scan(lambda a,x: tf.maximum(a,x), self.cumulative_return)
        self.drawdown = (self.rolling_max - self.cumulative_return) / self.rolling_max

        return -(tf.reduce_prod(self.average_return) - 1) / tf.reduce_max(self.drawdown)



    def train(self, states, last_actions, future_prices, cash_bias, prediction):
        # train model in batch to optimize reward
        self.sess.run(self.optimize, feed_dict={
            self.state: states,
            self.last_action: last_actions,
            self.future_price: future_prices,
            self.cash_bias: cash_bias,
            self.prediction: prediction
        })

    def CNN(self, state_size):
        print("Now we build actor (CNN only)")

        # Network Hyperparameters
        Conv2D_1_kernel = (5,1)
        Conv2D_1_filters = 3
        Conv2D_2_kernel = (state_size[1]-Conv2D_1_kernel[0]+1, 1)
        Conv2D_2_filters = 20

        # input layer
        State = Input(shape=[state_size[2], state_size[1], state_size[0]])
        prediction = Input(shape=[state_size[0]])

        # last action excludes cash bias component
        last_action = Input(shape=[state_size[0]])
        last_action_1 = Reshape((1, 1, state_size[0]))(last_action)

        # cash bias as constant input
        cash_bias = Input(shape=[1])

        # a set of convolution layers designed to extract 3 days pattern
        Conv2D_1 = Convolution2D(
            batch_input_shape=(self.BATCH_SIZE, state_size[2], state_size[1], state_size[0]),
# state_size[1] = time interval, state_size[0] = number of assests
            filters=Conv2D_1_filters,
            kernel_size=Conv2D_1_kernel,      # filters: make sure the row size of kernel is one
-> independent assets
```

```
            strides=1,
            padding='valid',                    # Padding method
            data_format='channels_first',       # (batch_size, channels, rows, cols),
            kernel_regularizer= regularizers.l2(1e-8),
            activity_regularizer=regularizers.l2(1e-8),
            bias_regularizer = regularizers.l2(1e-8),
            activation='relu'
        )(State)

        Conv2D_2 = Convolution2D(
            batch_input_shape= (self.BATCH_SIZE, Conv2D_1_filters, state_size[1]-
Conv2D_1_kernel[0]+1, state_size[0]),
            filters=Conv2D_2_filters,
            kernel_size=Conv2D_2_kernel,      # filters: make sure the row size of kernel is one
-> independent assets
            strides=1,
            padding='valid',                     # Padding method
            data_format='channels_first',        # (batch_size, channels, rows, cols),
            kernel_regularizer= regularizers.l2(1e-8),
            activity_regularizer= regularizers.l2(1e-8),
            bias_regularizer = regularizers.l2(1e-8),
            activation='relu'
        )(Conv2D_1)

        Concate = concatenate([Conv2D_2, last_action_1], axis=1)

        Conv2D_3 = Convolution2D(
            batch_input_shape=(self.BATCH_SIZE, Conv2D_2_filters+1, 1, state_size[0]),
            filters=1,
            kernel_size=(1, 1),               # filters: make sure the row size of kernel is one
-> independent assets
            strides=1,
            padding='valid',                    # Padding method
            data_format='channels_first',       # (batch_size, channels, rows, cols),
            kernel_regularizer= regularizers.l2(1e-8),
            activity_regularizer=regularizers.l2(1e-8),
            bias_regularizer = regularizers.l2(1e-8),
        )(Concate)

        # Flatten
        vote = Flatten()(Conv2D_3)

        # integrate with prediction
        vote_p = multiply([vote, prediction])

        F1 = concatenate([vote_p,cash_bias], axis=1)

        # output layer (actions)
        action = Activation('softmax')(F1)

        model = Model(inputs=[State, last_action, cash_bias, prediction], outputs=action)
        return model, model.trainable_weights, State, last_action, cash_bias, prediction,
[vote,prediction,vote_p,F1]

    def CNN_Dense(self, state_size):
        print("Now we build network (CNN + Dense)")

        # Network Hyperparameters
        Conv2D_1_kernel = (5,1)
        Conv2D_1_filters = 3
        Conv2D_2_kernel = (5,1)
        Conv2D_2_filters = 20
        Dense_units = 100

        # input layer
        State = Input(shape=[state_size[2], state_size[1], state_size[0]])
        prediction = Input(shape=[state_size[0]])

        # last action excludes cash bias component
        last_action = Input(shape=[state_size[0]])
        last_action_1 = Reshape((state_size[0], 1))(last_action)

        # cash bias as constant input
        cash_bias = Input(shape=[1])
```

```
        # a set of convolution layers designed to extract 3 days pattern
        Conv2D_1 = Convolution2D(
            batch_input_shape=(self.BATCH_SIZE, state_size[2], state_size[1], state_size[0]),
# state_size[1] = time interval, state_size[0] = number of assests
            filters=Conv2D_1_filters,
            kernel_size=Conv2D_1_kernel,      # filters: make sure the row size of kernel is one
-> independent assets
            strides=1,
            padding='valid',                  # Padding method
            data_format='channels_first',     # (batch_size, channels, rows, cols),
            kernel_regularizer= regularizers.l2(1e-8),
            activity_regularizer=regularizers.l2(1e-8),
            bias_regularizer = regularizers.l2(1e-8),
            activation='relu'
        )(State)

        Conv2D_2 = Convolution2D(
            batch_input_shape= (self.BATCH_SIZE, Conv2D_1_filters, state_size[1]-
Conv2D_1_kernel[0]+1, state_size[0]),
            filters=Conv2D_2_filters,
            kernel_size=Conv2D_2_kernel,       # filters: make sure the row size of kernel is
one -> independent assets
            strides= 1,
            padding='valid',                   # Padding method
            data_format='channels_first',      # (batch_size, channels, rows, cols),
            kernel_regularizer= regularizers.l2(1e-8),
            activity_regularizer= regularizers.l2(1e-8),
            bias_regularizer = regularizers.l2(1e-8),
            activation='relu'
        )(Conv2D_1)

        # Pass the feature maps according to assets one by one
        L1 = Permute((3,1,2))(Conv2D_2)
        L2 = Reshape((state_size[0], -1))(L1)
        L3 = concatenate([last_action_1, L2], axis=2)
        L4 = TimeDistributed(Dense(Dense_units, activation='relu'), input_shape=[state_size[0],-
1])(L3)
        L5 = TimeDistributed(Dense(1, activation='linear'), input_shape=[state_size[0],
Dense_units])(L4)

        # Generate voting score of each asset
        vote = Flatten()(L5)

        # integrate with prediction
        vote_p = multiply([vote, prediction])

        F1 = concatenate([vote_p,cash_bias],axis=1)

        # output layer (actions)
        action = Activation('softmax')(F1)

        model = Model(inputs=[State, last_action, cash_bias, prediction], outputs=action)
        return model, model.trainable_weights, State, last_action, cash_bias, prediction, F1

    def CNN_LSTM(self, state_size):
        print("Now we build network (CNN + LSTM)")

        # Network Hyperparameters
        Conv2D_1_kernel = (5,1)
        Conv2D_1_filters = 3
        Conv2D_2_kernel = (5,1)
        Conv2D_2_filters = 20
        Lstm_units = 10

        # input layer
        State = Input(shape=[state_size[2], state_size[1], state_size[0]])
        prediction = Input(shape=[state_size[0]])

        # last action excludes cash bias component
        last_action = Input(shape=[state_size[0]])

        # cash bias as constant input
        cash_bias = Input(shape=[1])
```

```
        # a set of convolution layers designed to extract 3 days pattern
        Conv2D_1 = Convolution2D(
            batch_input_shape=(self.BATCH_SIZE, state_size[2], state_size[1], state_size[0]),
# state_size[1] = time interval, state_size[0] = number of assests
            filters=Conv2D_1_filters,
            kernel_size=Conv2D_1_kernel,      # filters: make sure the row size of kernel is one
-> independent assets
            strides=1,
            padding='valid',                  # Padding method
            data_format='channels_first',     # (batch_size, channels, rows, cols),
            kernel_regularizer= regularizers.l2(1e-8),
            activity_regularizer=regularizers.l2(1e-8),
            bias_regularizer = regularizers.l2(1e-8),
            activation='relu'
        )(State)

        Conv2D_2 = Convolution2D(
            batch_input_shape= (self.BATCH_SIZE, Conv2D_1_filters, state_size[1]-
Conv2D_1_kernel[0]+1, state_size[0]),
            filters=Conv2D_2_filters,
            kernel_size=Conv2D_2_kernel,       # filters: make sure the row size of kernel is
one -> independent assets
            strides=1,
            padding='valid',                   # Padding method
            data_format='channels_first',      # (batch_size, channels, rows, cols),
            kernel_regularizer= regularizers.l2(1e-8),
            activity_regularizer= regularizers.l2(1e-8),
            bias_regularizer = regularizers.l2(1e-8),
            activation='relu'
        )(Conv2D_1)

        # Pass the feature maps according to assets one by one
        L1 = Permute((3,2,1))(Conv2D_2)
        L2 = TimeDistributed(LSTM(units=Lstm_units, activation='tanh', return_sequences=True),
input_shape=[state_size[0],-1,Conv2D_2_filters])(L1)
        L3 = TimeDistributed(LSTM(units=1, activation='linear', return_sequences=False),
input_shape=[state_size[0],-1,Lstm_units])(L2)

        # Generate voting score of each asset
        vote = Flatten()(L3)

        # integrate with prediction
        vote_p = multiply([vote, prediction])

        F1 = concatenate([vote_p,cash_bias],axis=1)

        # output layer (actions)
        action = Activation('softmax')(F1)

        model = Model(inputs=[State, last_action, cash_bias, prediction], outputs=action)
        return model, model.trainable_weights, State, last_action, cash_bias, prediction, vote

    def LSTM(self, state_size):
        print("Now we build network (LSTM only)")

        # Network Hyperparameters
        LSTM_units = 100

        # input layer
        State = Input(shape=[state_size[2], state_size[1], state_size[0]])
        prediction = Input(shape=[state_size[0]])

        # last action excludes cash bias component
        last_action = Input(shape=[state_size[0]])

        # cash bias as constant input
        cash_bias = Input(shape=[1])

        # LSTM layers
        L1 = Permute((3,2,1))(State)
        L2 = TimeDistributed(LSTM(units=LSTM_units, activation='tanh', return_sequences=True),
input_shape=[state_size[0],state_size[1],state_size[2]])(L1)
        L3 = TimeDistributed(LSTM(units=1, activation='linear', return_sequences=False),
```

```
input_shape=[state_size[0],state_size[1],LSTM_units])(L2)

        # Flatten
        vote = Flatten()(L3)

        # integrate with prediction
        vote_p = multiply([vote, prediction])

        F1 = concatenate([vote_p, cash_bias],axis=1)

        # output layer (actions)
        action = Activation('softmax')(F1)

        model = Model(inputs=[State, last_action, cash_bias, prediction], outputs=action)
        return model, model.trainable_weights, State, last_action, cash_bias, prediction, F1

    def LSTM_CNN(self, state_size):
        print("Now we build network (LSTM + CNN)")

        # Network Hyperparameters
        LSTM_units = 30

        # input layer
        State = Input(shape=[state_size[2], state_size[1], state_size[0]])
        prediction = Input(shape=[state_size[0]])

        # last action excludes cash bias component
        last_action = Input(shape=[state_size[0]])
        last_action_1 = Reshape((1, 1, state_size[0]))(last_action)

        # cash bias as constant input
        cash_bias = Input(shape=[1])

        # LSTM layers
        L1 = Permute((3,2,1))(State)
        L2 = TimeDistributed(LSTM(units=LSTM_units, activation='tanh', return_sequences=False),
input_shape=[state_size[0],state_size[1],state_size[2]])(L1)
        L3 = Permute((2,1))(L2)
        L4 = Reshape((LSTM_units,1,state_size[0]))(L3)

        Concate = concatenate([L4, last_action_1], axis=1)

        Conv2D = Convolution2D(
            batch_input_shape=(self.BATCH_SIZE, LSTM_units+1, 1, state_size[0]),
            filters=1,
            kernel_size= (1, 1),              # filters: make sure the row size of kernel is one
-> independent assets
            strides=1,
            padding='valid',                  # Padding method
            data_format='channels_first',     # (batch_size, channels, rows, cols),
            kernel_regularizer= regularizers.l2(1e-8),
            activity_regularizer=regularizers.l2(1e-8),
            bias_regularizer = regularizers.l2(1e-8),
        )(Concate)

        # Flatten
        vote = Flatten()(Conv2D)

        # integrate with prediction
        vote_p = multiply([vote, prediction])

        F1 = concatenate([vote_p, cash_bias],axis=1)

        # output layer (actions)
        action = Activation('softmax')(F1)

        model = Model(inputs=[State, last_action, cash_bias, prediction], outputs=action)
        return model, model.trainable_weights, State, last_action, cash_bias, prediction, F1
```

3. Portfolio.py

It defines a portfolio class which stores the fundamental and technical indicators and calculates a daily return when given an action

```
try:
    import quandl
except ImportError:
    pass

import pandas as pd
import os
import numpy as np
import random
import sys
import datetime
import Data
import Indicator
from pandas.tseries.offsets import BDay

class Portfolio(object):

    def __init__(self,tickers, start, end, mode='train'):
        self.tickers = tickers[:]
        self.tickers_num = len(self.tickers)
        self.start = start
        self.before_start = (datetime.datetime.strptime(start, '%Y-%m-%d') -
BDay(200)).strftime('%Y-%m-%d')
        self.end = end
        self.mode = mode
        self.consistent = True

        # fundamental indicator
        self.df_close = self.extract_data(self.tickers, self.start, self.end, label='Adj.
Close')
        self.df_high = self.extract_data(self.tickers, self.start, self.end, label='Adj.
High').fillna(method='ffill')
        self.df_low = self.extract_data(self.tickers, self.start, self.end, label='Adj.
Low').fillna(method='ffill')
        self.df_open = self.extract_data(self.tickers, self.start, self.end, label='Adj.
Open').fillna(method='ffill')
        self.df_volume = self.extract_data(self.tickers, self.start, self.end,
label='Volume').fillna(method='ffill')

        # technical indicator
        self.df_roc = Indicator.roc(self.df_close).fillna(method='ffill')
        self.df_macd = Indicator.macd(self.df_close).fillna(method='ffill')
        self.df_ma5 = Indicator.ma(self.df_close, 5).fillna(method='ffill')
        self.df_ma10 = Indicator.ma(self.df_close,10).fillna(method='ffill')
        self.df_ema20 = Indicator.ema(self.df_close,20).fillna(method='ffill')
        self.df_sr20 = Indicator.sharp_ratio(self.df_close,20).fillna(method='ffill')

        try:
            self.start_index = np.where(self.df_close.index == start)[0][0]
        except IndexError:
            pass

        self.df_normalized = (self.df_open.shift(-1) / self.df_open).fillna(1)            # next
day's price / today's price
        self.portfolio_weights = list()
        self.portfolio_return = list([1])
        self.portfolio_UCRP = list([1])
        self.portfolio_UBHP = list([1])
        self.transaction_factor = 0.0025

        print('Created Portfolio: {}'.format(tickers))
        print('Number of assets: {}'.format(self.tickers_num))
        print('Included Cash Bias')
        if mode == 'train':
            print('Training period: {} to {}'.format(start,end))
        elif mode == 'valid':
            print('Validating period: {} to {}'.format(start,end))
```

```
        elif mode == 'test':
            print('Testing period: {} to {}'.format(start,end))

    def calculate_return(self, weight, last_weight, step ):
        # make sure sum of weight equal to 1
        transaction_cost = 1 - self.transaction_factor * np.sum(np.abs(weight[:-1] -
last_weight))
        future_price = np.append(self.df_normalized.iloc[step + self.start_index].values, [1])
        day_return = np.dot(future_price, weight) * transaction_cost
        self.portfolio_return.append(day_return)
        self.portfolio_weights.append(weight)

        return day_return, future_price

    # clear the portfolio and return when starting a new episode
    def clear(self):
        self.portfolio_weights = list()
        self.portfolio_return  = list([1])


    # extract data individually, and store to data/stock
    def import_from_quandl(self, symbol, start, end):
        if os.path.exists('./data/stock/{}.csv'.format(symbol)):
            print('Read from stored data:', symbol)
        else:
            Data.download_data(symbol)
        df = pd.read_csv('./data/stock/{}.csv'.format(symbol), header=0, parse_dates=True,
index_col=0)[self.before_start : end]
        if df.empty:
            print("Historical data of {} does not cover period from {} to {}".format(symbol,
start, end))
            self.consistent = False
        elif df.index[0] != datetime.datetime.strptime(self.before_start,'%Y-%m-%d'):
            print("Historical data of {} does not cover period from {} to {}".format(symbol,
start, end))
            self.consistent = False
        return df

    # extract a group of data with same label according to tickers
    def extract_data(self, tickers, start, end, label):
        main_df = pd.DataFrame()
        for ticker in tickers:
            df = self.import_from_quandl(ticker, start, end)
            df.rename(columns={label:ticker}, inplace=True)
            df = df[[ticker]]
            if main_df.empty:
                main_df = df
            else:
                main_df = main_df.join(df, how='outer')
        return main_df

    # randomly shuffle the assets order
    def shuffle(self):
        random.shuffle(self.tickers)

        # fundamental indicator
        self.df_close = self.df_close[self.tickers]
        self.df_open = self.df_open[self.tickers]
        self.df_high = self.df_high[self.tickers]
        self.df_low = self.df_low[self.tickers]
        self.df_volume = self.df_volume[self.tickers]

        # technical indicator
        self.df_roc = self.df_roc[self.tickers]
        self.df_normalized = self.df_normalized[self.tickers]


    # Uniform Constant Rebalance Portfolio
    def UCRP(self):
        weight = np.array([1 for _ in range(self.tickers_num)]) / self.tickers_num
        for j in range(self.start_index, len(self.df_normalized) - 1):
            day_return = np.dot(self.df_normalized.iloc[j], weight)
            self.portfolio_UCRP.append(day_return)
```

```
# Uniform Buy and Hold Portfolio
def UBHP(self):
    weight = np.array([1 for _ in range(self.tickers_num)]) / self.tickers_num
    for j in range(self.start_index, len(self.df_normalized) - 1):
        day_return = np.dot(self.df_normalized.iloc[j], weight)
        self.portfolio_UBHP.append(day_return)
        weight *= self.df_normalized.iloc[j]
        weight = weight / sum(weight)
```

4. ReplayBuffer.py

It stores the experiences which was iterated and provides method to draw a mini-batch for experience replay

```
from collections import deque
import numpy as np
import random

class ReplayBuffer(object):

    def __init__(self, buffer_size, sample_bias):
        self.buffer_size = buffer_size
        self.num_experiences = 0
        self.sample_bias = sample_bias
        self.buffer = list()

    def getBatch(self, batch_size):
        if self.num_experiences <= batch_size:
            return self.buffer, self.num_experiences
        else:
            # the mini-batch has to be in time-order, batch is selected according weighted
probability
            random_batch = self.weighted_sample(self.num_experiences-batch_size,
self.sample_bias)
            # print(self.num_experiences-batch_size, random_batch)
            return self.buffer[random_batch : random_batch + batch_size], batch_size

    def weighted_sample(self, size, sample_bias):
        # the samples are drawn according to weighting of [1,2,3,4......100], latest experiences
has high chance to be drawn, vice verse
        weight = np.array([sample_bias**i for i in range(1,size+1)])
        return np.random.choice(size, 1, p=weight / sum(weight))[0]

    def size(self):
        return self.buffer_size

    def add(self, state, future_price, last_action, prediction):
        experience = (state, future_price, last_action, prediction)
        if self.num_experiences < self.buffer_size:
            self.buffer.append(experience)
            self.num_experiences += 1
        else:
            self.buffer.pop(0)
            self.buffer.append(experience)

    def count(self):
        # if buffer is full, return buffer size
        # otherwise, return experience counter
        return self.num_experiences

    def erase(self):
        self.buffer = list()
        self.num_experiences = 0
```

5. Result.py

It visualizes the result generated by model and compares its performance to benchmarks.

```
import pickle
import matplotlib.pyplot as plt
from matplotlib import style
from cycler import cycler
import pandas
from Portfolio import Portfolio
import Performance
import os
import Ssh
import numpy as np
from universal import algos
import logging

# plot settings
logging.basicConfig(format='%(asctime)s %(message)s', level=logging.DEBUG)
style.use('ggplot')
plt.rcParams['grid.color'] = 'lightgrey'
plt.rcParams['axes.facecolor'] = 'w'
plt.rcParams['axes.prop_cycle'] =
cycler(color=[u'#1f77b4',u'#d62728',u'#ff7f0e',u'#7f7f7f',u'#2ca02c',u'#98df8a',u'#aec7e8',u'#ff
9896',u'#9467bd',u'#c5b0d5',

u'#8c564b',u'#c49c94',u'#e377c2',u'#f7b6d2',u'#ffbb78',u'#c7c7c7',u'#bcbd22',u'#dbdb8d',u'#17bec
f',u'#9edae5'])

class Result(object):

    def __init__(self,filename):

        P = self.read_pickle(filename)
        self.tickers = P.tickers
        self.start_index = P.start_index
        self.date = P.df_close.index[self.start_index:]
        self.weights = P.portfolio_weights
        self.close = P.df_close
        self.model = ['model','UCRP','UBHP','RMR','OLMAR','PAMR','Anticor']
        self.model_return = []
        self.model_return.append(P.portfolio_return)

        # Passive trading strategy (UCRP and UBHP)
        self.model_return.append(P.portfolio_UCRP)
        self.model_return.append(P.portfolio_UBHP)

        # Robust median reversion strategy
        RMR = algos.RMR()
        RMR_return = RMR.run(self.close.iloc[self.start_index:])
        self.model_return.append(RMR_return.r)

        # Online moving average reversion strategy
        OLMAR = algos.OLMAR()
        OLMAR_return = OLMAR.run(self.close.iloc[self.start_index:])
        self.model_return.append(OLMAR_return.r)

        # Passive aggressive mean reversion strategy
        PAMR = algos.PAMR()
        PAMR_return = PAMR.run(self.close.iloc[self.start_index:])
        self.model_return.append(PAMR_return.r)

        # Anticor (anti-correlation)
        Anticor = algos.Anticor()
        Anticor_return = Anticor.run(self.close.iloc[self.start_index:])
        self.model_return.append(Anticor_return.r)

    def read_pickle(self, filename):
        if not (os.path.exists('./result/{}'.format(filename))):
            # Ssh.getall('./result','./result')
            Ssh.get(local = './result/{}'.format(filename), server =
'./result/{}'.format(filename))
```

```
        with open('./result/'+filename,'rb') as file:
            P = pickle.load(file)
        return P

    def plot_weights(self):
        plt.title('Weights of assets')
        plt.xlabel('Date')
        plt.ylabel('Weights')
        plt.plot(self.date, self.weights)
        plt.legend(np.concatenate([self.tickers,['Cash']]))

    def plot_price(self):
        plt.title('Adjusted Close')
        plt.xlabel('Date')
        plt.plot(self.close.iloc[self.start_index:])
        plt.legend(self.tickers)

    def plot_porfolio_return(self):
        plt.title('Portfolio Return')
        plt.ylabel('Cumulative Return')
        plt.xlabel('Date')
        # plt.ylim((0.75,2.75))

        for i in range(len(self.model)):
            if i < 4:
                plt.plot( self.date, Performance.cummulative_return(self.model_return[i]) ,
linestyle = '-')
            else:
                plt.plot( self.date, Performance.cummulative_return(self.model_return[i]) ,
linestyle = '--')
        plt.legend(self.model, loc=2)

    def plot_epsiode_reward(self,filename):
        with open('./result/'+ filename,'rb') as file:
            self.epsiode_reward = pickle.load(file)
        plt.xlabel('Episode')
        plt.ylabel('Performance')
        plt.title('Episode Performance')
        plt.plot(self.epsiode_reward)

    def plot_return_and_weights(self):
        ax1 = plt.subplot(211)
        plt.title('Portfolio Return')
        ax2 = plt.subplot(212, sharex=ax1)
        plt.title('Weights of assets')
        ax1.plot( self.date, Performance.cummulative_return(self.model_return[0]) )
        ax1.tick_params(labelbottom='off')
        ax2.plot( self.date, self.weights)
        ax2.set_xlabel('Date')
        ax2.legend(np.concatenate([self.tickers,['Cash']]))

    def plot_price_and_weights(self):
        ax1 = plt.subplot(211)
        plt.title('Adjusted Close (normalized)')
        ax2 = plt.subplot(212, sharex=ax1)
        plt.title('Weights of assets')
        ax1.plot(self.close.iloc[self.start_index:] / self.close.iloc[self.start_index])
        ax1.legend(self.tickers)
        ax1.tick_params(labelbottom='off')
        ax2.plot(self.date, self.weights)
        ax2.set_xlabel('Date')
        ax2.legend(np.concatenate([self.tickers,['Cash']]), loc=5)

    def perforamnce(self):
        max_drawdown = []
        sharpe_ratio = []
        cumulative_return = []
        volatility = []
        performance_indicator = ['Cumulative Return','Max Drawdown','Sharpe Ratio','Volatility']
        for i, model in enumerate(self.model):
            max_drawdown.append(Performance.max_drawdown(self.model_return[i]))
            sharpe_ratio.append(Performance.sharpe_ratio(self.model_return[i]))
            cumulative_return.append(Performance.cummulative_return(self.model_return[i])[-1])
            volatility.append(Performance.volatility(self.model_return[i]))
```

```
        table = pandas.DataFrame(np.array([cumulative_return,max_drawdown,sharpe_ratio,
volatility]), performance_indicator, self.model)
        print("")
        print("Trading Period: {} - {}".format(self.date[0].strftime('%Y-%m-%d'),self.date[-
1].strftime('%Y-%m-%d')))
        print(table)
        print("")
        print("Highest Cumulative Return:
{}".format(self.model[np.argmax(cumulative_return).item()]))
        print("Lowest Maximum Drawdown: {}".format(self.model[np.argmin(max_drawdown).item()]))
        print("Highest Sharpe Ratio: {}".format(self.model[np.argmax(sharpe_ratio).item()]))
        print("Lowest Volatility: {}".format(self.model[np.argmin(volatility).item()]))

    def plot_all(self):
        plt.figure(1)
        self.plot_weights()
        plt.figure(2)
        self.plot_price()
        plt.figure(3)
        self.plot_porfolio_return()
        plt.figure(4)
        self.plot_epsiode_reward('Epsiode_Performance_{}_{}_{}.pickle'.format(self.P.mode,
self.P.start, self.P.end))
        plt.figure(5)
        self.plot_return_and_weights()
        plt.figure(6)
        self.plot_price_and_weights()
        self.perforamnce()
        plt.show()


if __name__ == "__main__":
    filename = "CNN_result.pickle"
    result = Result(filename)
    result.plot_all()
    plt.show()
```

6. Indicator.py

It provides a set of methods to generate technical indicators for input price matrix.

```
import pandas as pd
import numpy as np

# price should be in form of dataframe

# Moving Average Convergence / Divergence
def macd(price):
    return price.ewm(span=12,min_periods=0,adjust=True,ignore_na=False).mean() /
price.ewm(span=26,min_periods=0,adjust=True,ignore_na=False).mean()

# rate of change
def roc(price):
    return (price / price.shift(1)).fillna(1)

# Simple moving average
def ma(price, day):
    return price.rolling(window=day,min_periods=1,center=False).mean()

# Exponential Moving Average
def ema(price, day):
    return price.ewm(span=day, min_periods=0,adjust=True,ignore_na=False).mean()

def volatility(price, day):
    return price.rolling(window=day,min_periods=1,center=False).std()

def sharp_ratio(price, day):
    price = np.log((price / price.shift(1)).fillna(1))
    sliding_window = price.rolling(window=day,min_periods=1,center=False)

    # Risk free rate for sharpe ratio (Three-month U.S. Treasury bill)
    risk_free_rate = np.log(1.0148) / 60

    return np.sqrt(sliding_window.count()) * (sliding_window.mean() - risk_free_rate)  /
sliding_window.std()
```

7. Performance.py

It provides a set of methods to evaluate the performance of result

```
import numpy as np

def cummulative_return(day_return):
    return np.array(day_return).cumprod()

def sharpe_ratio(day_return):
    log_day_return = np.log(np.array(day_return))
    risk_free_rate = np.log(1.0148) / 60    # Three-month U.S. Treasury bill
    return np.sqrt(log_day_return.size) * (np.average(log_day_return) - risk_free_rate) /
np.std(log_day_return)

def volatility(day_return):
    log_day_return = np.log(np.array(day_return))
    return np.std(log_day_return)

def max_drawdown(day_return):
    cumulative_return = cummulative_return(day_return)
    max_drawdown = 0
    peak = cumulative_return[0]
    for x in cumulative_return:
        if x > peak:
            peak = x
        daily_drawdown = (peak - x) / peak
        if daily_drawdown > max_drawdown:
            max_drawdown = daily_drawdown
    return max_drawdown
```

8. Data.py

It downloads the historical data of stocks in S&P500 from quandle and groups them by GICS sector

```
import bs4 as bs
import pickle
import requests
import os
import pandas
import numpy as np
import random
import datetime
from sklearn.preprocessing import MinMaxScaler
from pandas.tseries.offsets import BDay
try:
    import quandl
except ImportError:
    pass

# GICS sector
# sector = ["Consumer Discretionary", "Consumer Staples", "Energy", "Financials", "Health Care",
"Industrials",
#           "Information Technology", "Materials","Real Estate","Telecommunication
Services","Utilities"]

def sp500_tickers(sector = "All"):

    if os.path.exists("ticker/sp500tickers_{}.pickle".format(sector)):
        with open("ticker/sp500tickers_{}.pickle".format(sector),"rb") as file:
            tickers = pickle.load(file)
    else:
        resp = requests.get('http://en.wikipedia.org/wiki/List_of_S%26P_500_companies')
        soup = bs.BeautifulSoup(resp.text, 'lxml')
        table = soup.find('table', {'class': 'wikitable sortable'})
        tickers = []
        for row in table.findAll('tr')[1:]:
            if row.findAll('td')[3].text == sector or sector == "All":
                ticker = row.findAll('td')[0].text
                tickers.append(ticker)
        if tickers == []:
            print("Invalid sector !")
            exit()
        else:
            with open("ticker/sp500tickers_{}.pickle".format(sector),"wb") as file:
                pickle.dump(tickers,file)
    return tickers

def download_data(symbol):
    if not os.path.exists('./data/stock/{}.csv'.format(symbol)):
        df = quandl.get('WIKI/{}'.format(symbol).replace('.','_'),
authtoken='x5eLEX_BwoVKBfXYWSLz')
        try:
            df.to_csv('./data/stock/{}.csv'.format(symbol))
            print('Imported from quandl: {}'.format(symbol))
            # record all the downloaded data tagged by period and mode
            # if os.path.exists("tickers_stocks.pickle"):
            #     with open("tickers_stocks.pickle","rb") as file:
            #         tickers = pickle.load(file)
            #     tickers.append(symbol)
            #     with open("tickers_stocks.pickle","wb") as file:
            #         pickle.dump(tickers,file)
            # else:
            #     with open("tickers_stocks.pickle","wb") as file:
            #         tickers = [symbol]
            #         pickle.dump(tickers,file)
        except IndexError:
            pass
    else:
        print('Data already exists:', symbol)
```

```
def download_tickers(tickers):
    for ticker in tickers:
        download_data(ticker)

def download_sp500():
    # download stocks from sp500
    tickers = sp500_tickers("All")
    for ticker in tickers:
        download_data(ticker)

def generate_list(sector, start, end):
    # generate a list of stocks which is available within a defined period
    with open("ticker/sp500tickers_{}.pickle".format(sector),'rb') as file:
        tickers = pickle.load(file)
    available_stock = []
    for symbol in tickers:
        print(symbol)
        if not os.path.exists('./data/stock/{}.csv'.format(symbol)):
            download_data(symbol)
        df = pandas.read_csv('./data/stock/{}.csv'.format(symbol), header=0, parse_dates=True,
index_col=0)
        before_start = (datetime.datetime.strptime(start, '%Y-%m-%d') - BDay(200)).strftime('%Y-
%m-%d')
        df = df[before_start : end]
        if df.empty:
            print("Historical data of {} does not cover period from {} to {}".format(symbol,
start, end))
            continue
        elif df.index[0] != datetime.datetime.strptime(before_start,'%Y-%m-%d'):
            print("Historical data of {} does not cover period from {} to {}".format(symbol,
start, end))
            continue
        available_stock.append(symbol)
    with open("ticker/sp500tickers_{}_{}_{}.pickle".format(sector, start, end),"wb") as file:
        pickle.dump(available_stock, file)

def generate_test_file(tickers, start, end):
    start = (datetime.datetime.strptime(start, '%Y-%m-%d') - BDay(200))
    end = datetime.datetime.strptime( end, "%Y-%m-%d")
    test_data = []

    for ticker in tickers:
        temp = pandas.read_csv('./data/stock/{}.csv'.format(ticker), header=0, parse_dates=True,
index_col=0).loc[start:end]
        scaler = MinMaxScaler(feature_range=(0,1))
        temp['Scaled_adj_close'] = scaler.fit_transform(temp[['Adj. Close']])
        temp['Scaled_volume'] = scaler.fit_transform(temp[['Volume']])
        test_data.append(temp[['Open','High','Low','Close','Adj.
Close','Volume','Scaled_adj_close','Scaled_volume']])
    print(test_data[0])

    for i in range(len(test_data[0])):
        temp = pandas.DataFrame(columns=['Open','High','Low','Close','Adj.
Close','Volume','Scaled_adj_close','Scaled_volume'])
        for ii, ticker in enumerate(tickers):
            temp.loc[ii] = test_data[ii].iloc[i]
        temp.to_csv('./test/{}.csv'.format(test_data[0].index[i].strftime('%Y-%m-%d')))
        print('wrote',test_data[0].index[i].strftime('%Y-%m-%d'))
```

9. Strategy.py

It computes the benchmark strategies which are generated by package universal-portfolio.

```
from universal import algos
from Portfolio import Portfolio
import logging
import numpy as np

logging.basicConfig(format='%(asctime)s %(message)s', level=logging.DEBUG)
import matplotlib.pyplot as plt

# It produces benchmarking strategy
# group 1
sector = "Information Technology"
tickers = ['AAPL', 'AMAT', 'AMD', 'CSCO', 'EBAY', 'GLW', 'HPQ', 'IBM', 'INTC', 'KLAC', 'MSFT',
'MU', 'NVDA', 'QCOM', 'TXN']

# # group 2
# sector = "Consumer Discretionary"
# tickers = ['AZO', 'BBY', 'DHI', 'F', 'GPS', 'GRMN', 'HOG', 'JWN', 'MAT', 'MCD', 'NKE', 'SBUX',
'TJX', 'TWX', 'YUM']
#
# group 3
# sector = "Industrials"
# tickers = ['BA', 'CAT', 'CTAS', 'EMR', 'FDX', 'GD', 'GE', 'LLL', 'LUV', 'MAS', 'MMM', 'NOC',
'RSG', 'UNP', 'WM']

P = Portfolio(tickers, '2016-01-04', '2016-12-31', mode='train')
date = P.df_close.index[P.start_index:]

RMR = algos.RMR()
RMR_return = RMR.run(P.df_close.iloc[P.start_index:])
RMR_return.plot(weights=True, assets=False, ucrp=True, logy=False)

OLMAR = algos.OLMAR()
OLMAR_return = OLMAR.run(P.df_close.iloc[P.start_index:])
OLMAR_return.plot(weights=False, assets=False, ucrp=True, logy=False)

PAMR = algos.PAMR()
PAMR_return = PAMR.run(P.df_close.iloc[P.start_index:])
PAMR_return.plot(weights=False, assets=False, ucrp=True, logy=False)

Anticor = algos.Anticor()
Anticor_return = Anticor.run(P.df_close.iloc[P.start_index:])
Anticor_return.plot(weights=False, assets=False, ucrp=True, logy=False)

plt.show()
```

10. DLTestBench107.py

A price prediction model which was written by my groupmate Garrett. It generates a set of daily predicted price and then integrates to proposed model in this research.

```
import numpy
import pandas
import os
import pickle
import datetime
import math
from keras.layers import *
from keras.models import *
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint
from sklearn.preprocessing import MinMaxScaler
import tensorflow as tf
from keras.backend.tensorflow_backend import set_session

os.environ["CUDA_VISIBLE_DEVICES"] = '0'
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.3
set_session(tf.Session(config=config))

input_timesteps = 120
predict_timesteps = 5
data_dim = 6
num_portfolio = 5

test_name = "model107"

filepath = "./time"
#filepath = "/Users/Vanguardcreed/Desktop/time"
destinationpath = "./result"
#destinationpath = "/Users/Vanguardcreed/Desktop/time"

start_date = datetime.datetime.strptime( '1/1/10', "%m/%d/%y")
split_date = datetime.datetime.strptime( '1/1/16', "%m/%d/%y")

def get_dates(mode="all"):
    # return an array containing the dates
    df = pandas.read_csv("{}/date.csv".format(filepath))
    df['Date'] = pandas.to_datetime(df['Date'])
    if mode=="train":
        result = df['Date'][(df['Date']<split_date) & (df['Date']>start_date)]
    elif mode == "test":
        train = df.loc[(df['Date']<split_date) & (df['Date']>start_date)].index
        limit = train[-1]
        result = df['Date'].iloc[limit-input_timesteps+1:]
    else:
        return
    return result

def get_model(LSTM_units=200, dropout=0.2):

    # return the model
    def conv_unit(x, depth, field, strides=1, padding="same"):
        x = Conv1D(depth,field, strides = strides, padding=padding)(x)
        x = BatchNormalization(axis=1,momentum=0.99)(x)
        x = Activation('relu')(x)
        return x

    input_tensor = Input(shape=(input_timesteps, data_dim,))
    x = conv_unit(input_tensor, 32, 5, strides=2, padding="valid")

    branch11 = conv_unit(x, 64, 1)
    branch31 = conv_unit(x, 64, 3)
    branch51 = conv_unit(x, 64, 5)
    branch_pool1 = AveragePooling1D(3, strides=1, padding='same')(x)

    x = concatenate([branch11, branch31, branch51, branch_pool1])
```

```
    branch32 = conv_unit(x, 128, 3)
    branch52 = conv_unit(x, 128, 5)
    branch72 = conv_unit(x, 128, 7)
    branch_pool2 = AveragePooling1D(3, strides=1, padding="same")(x)

    x = concatenate([branch32, branch52, branch72, branch_pool2])

    branch33 = conv_unit(x, 192, 3)
    branch73 = conv_unit(x, 192, 7)
    branch103 = conv_unit(x, 192, 10)
    branch_pool3 = conv_unit(x, 128,1)
    branch_pool3 = AveragePooling1D(3, strides=1, padding='same')(branch_pool3)
    x = concatenate( [branch33, branch73, branch103, branch_pool3])
    x = Dropout(dropout)(x)
    x = LSTM(LSTM_units, activation="tanh",
recurrent_activation="hard_sigmoid",return_sequences=True)(x)
    x = Dropout(dropout)(x)
    x = LSTM(LSTM_units,activation="tanh", recurrent_activation="hard_sigmoid",
return_sequences=False)(x)
    x = Dropout(dropout)(x)
    x = Dense(100, activation="tanh")(x)
    x = Dense(1, activation="tanh")(x)
    model = Model(input_tensor, x)
    optimizer = Adam(lr=0.0001, beta_1=0.8, beta_2=0.9)
    model.compile(loss='mse', optimizer='adam')
    return model

def load_test(filepath):
    model = load_model(filepath)
    test_model(model)
    return

def check_y(y):
    print("check y")
    base = len(y)
    extreme = [i for i in y if i>=1 or i<=-1]
    print("extreme case is" + str(len(extreme)/base))
    positive = sum( [1 for i in y if i>0])/base
    print("positive case is" + str(positive))
    returnx

def train_model(model, num_epochs=100, batch_size=256, start_from=0):

    if os.path.exists("./train_features.pickle"):
        with open('./train_features.pickle', 'rb') as file:
            X,y = pickle.load(file)
            print("loaded features and label")
    else:
        # list of trained or tested date
        dates = get_dates(mode = "train")

        # list of daily data
        dflist = []
        new_dates = []

        # fetch the daily data
        for date in dates:
            try:
                df = pandas.read_csv("{}/{}.csv".format(filepath, date.strftime("%Y-%m-%d")))
                dflist.append(df)
                new_dates.append(date)
                print("read {}.csv".format(date.strftime("%Y-%m-%d")))
            except:
                print("cannot read {}/{}.csv".format(filepath, date.strftime("%Y-%m-%d")))

        dates = new_dates
        data_len = input_timesteps + predict_timesteps

        # prepare the first set of features and labels
        X,y = assemble_data(dflist[0:data_len])

        for i in range(1, len(dflist)-data_len+1, 5):
            tmp_X, tmp_y = assemble_data(dflist[i:i+data_len])
```

```
            X = numpy.concatenate((X,tmp_X), axis=0)
            y = numpy.concatenate((y,tmp_y), axis=0)
            print("Prepared feature set of of {}".format(dates[i].strftime("%Y-%m-%d")))

        with open('train_features.pickle', 'wb') as file:
            pickle.dump([X,y],file)

    if start_from != 0:
        try:
            model.load_weights("{}/model_%03d.hdf5".format(destinationpath)%start_from)
            print("loaded weights ({})".format(start_from))
        except:
            print("unable to load weights ({})".format(start_from))
            print("start from 0 epoch")

    print("training period: {} to {}".format(start_date.strftime("%Y-%m-
%d"),split_date.strftime("%Y-%m-%d")))
    print("input timesetpes: {} day".format(input_timesteps))
    print("predict timesteps: {} day".format(predict_timesteps))

    checkpoint = ModelCheckpoint(destinationpath+"/model2_{epoch:03d}.hdf5", verbose=1,
save_best_only=False)
    history = model.fit(X, y, epochs=num_epochs-start_from, batch_size=batch_size,
shuffle=False, callbacks=[checkpoint])
    trainingDoc = pandas.DataFrame()
    trainingDoc['loss'] = history.history['loss']
    trainingDoc.to_csv("{}/train_loss_{}.csv".format(destinationpath, test_name))
    print("saved train loss of {}".format(test_name))
    return model

def assemble_data(dates_df):
    # preprocess the data and construct a input matrix X and label Y to network

    # number of stocks to be trained
    num_samples = len(dates_df[0].index)
    X = []
    y = []

    for stock in range(num_samples):
        # for each stock:
        SX = []
        for i in range(0, input_timesteps):
            # 8 financial indicators:
Open,High,Low,Close,Adj_close,Volume,Scaled_adj_close,Scaled_volume
            SX.append(dates_df[i].iloc[stock, 2:10].values)
        SX = numpy.array(SX)

        # close price as label
        sy = dates_df[input_timesteps+predict_timesteps-1].iloc[stock, 5]

        # logarithm change of close price
        tmp_y = (sy /SX[input_timesteps-1][3])
        y.append(math.log(tmp_y))

        # preprocessing
        scaler = MinMaxScaler(feature_range=(0.1,0.9))

        # dropped adjusted close and volume, scale the Open,High,Low,Close
        tmp_x = numpy.concatenate( (SX[:,6:8],scaler.fit_transform(SX[:, 0:4]) ),axis=1)

        # add to main features
        X.append(tmp_x)

        # y.append( (tmp_y[0][3] - tmp_x[input_timesteps-1][5])/tmp_x[input_timesteps-1][5] )
        # tmp = numpy.concatenate( (sy[6:8].reshape(1,-1),scaler.transform(sy[0:4].reshape(1,-
1))), axis=0)
    return numpy.array(X), numpy.array(y).reshape(-1,1)

def test_model(model):

    def top_k(data):
        top_list = [0] * num_portfolio
        position_list = [-1]* num_portfolio
```

```
        for i in range(len(data)):
            if data[i] > top_list[0]:
                top_list, position_list = insert_top_k(data[i],i,top_list,position_list)
        return top_list, position_list

    #auxiliary fucntion for top_k
    def insert_top_k(sim, pos, top_list, position_list):
        limit = len(top_list)
        for i in range(1, limit + 1):
            if i == limit:
                top_list[i-1] = sim
                position_list[i-1] = pos
            elif top_list[i] < sim:
                top_list[i-1] = top_list[i]
                position_list[i-1] = position_list[i]
            else:
                top_list[i-1] = sim
                position_list[i-1] = pos
                break
        return top_list, position_list

    dates = get_dates(mode="test")
    column_list = ['Date','MSE','Accuracy'] + [str(i) for i in range(0, num_portfolio)]
    result_df = pandas.DataFrame(columns=column_list)
    dflist = []
    new_dates = []
    for date in dates:
        try:
            #print(filepath+"/"+dat.strftime("%Y-%m-%d")+".csv")
            df = pandas.read_csv("{}/{}.csv".format(filepath,date.strftime("%Y-%m-%d")))
            dflist.append(df)
            new_dates.append(date)
        except:
            pass
    data_len = input_timesteps + predict_timesteps
    test_date = new_dates[input_timesteps-1: len(new_dates)-predict_timesteps]
    for i in range(0, len(test_date)):
        print(test_date[i])
        X_test, y_test = assemble_data(dflist[i:i+data_len])
        y_pred = model.predict(X_test)[:,0]
        print(y_pred)
        print(y_test)
        #mse = (( y_pred - y_test ) ** 2).mean(axis=None)
        mse = ((y_pred-y_test)**2).mean()
        y_act = X_test[:, input_timesteps-1 ,5]
        change = (y_pred - y_act)
        pred, portfolio = top_k(change)
        direction = (y_test[:,0] - y_act)
        accuracy = sum([1 for x,y in zip(y_pred,y_test) if x*y>0] )/direction.shape[0]
        print(accuracy)
        result_df = result_df.append(pandas.Series([test_date[i], mse,
accuracy]+portfolio,index=column_list), ignore_index=True)
        #result_df[i] = numpy.nparray([test_date[i], mse, accuracy] + portfolio)

    result_df.to_csv("{}/{}_test.csv".format(destinationpath,test_name))
    return

def run():
    if os.path.exists("{}/{}.h5".format(destinationpath,test_name)):
        model = load_model("{}/{}.h5".format(destinationpath,test_name))
        print("loaded model")
    else:
        model = get_model()
        model = train_model(model, start_from=81)
        try:
            model.save("{}/{}.h5".format(destinationpath,test_name))
            print("model {}.h5 is saved".format(test_name))
        except:
            print("Error: model is not saved")
    test_model(model)
    return

run()
```

11. Ssh.py

A set of method to communicate with GPU machine in a convenient way.

```
import sys
import paramiko
from glob import glob
import os


def run(local, server = 'run.py', ip='147.8.182.2',
username='lok419',password='12345678',port=50888):

    # Connect to remote host
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, username=username, password=password,port=port)
    print('Successfully connected to %s'%(ip))

    # Setup sftp connection and transmit this script
    sftp = client.open_sftp()
    sftp.put(local,server)
    print('Successfully copied the code to server')
    sftp.close()

    # Run the transmitted script remotely without args and show its output.
    # SSHClient.exec_command() returns the tuple (stdin,stdout,stderr)
    print('Now executing %s ...'%(local))
    stdin, stdout, stderr = client.exec_command('python %s'%(server), get_pty=True)

    for line in iter(stdout.readline, ""):
        print(line, end = "")
    for line in iter(stderr.readline, ""):
        print(line, end = "")

    client.exec_command('rm %s'%(server))
    client.close()

def copy(local, server, ip='147.8.182.2', username='lok419',password='12345678',port=50888):

    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, username=username, password=password,port=port)
    print('Successfully connected to 147.8.182.2')
    sftp = client.open_sftp()
    sftp.put(local,server)
    print('Successfully copied from %s to %s'%(local,server))

    sftp.close()
    client.close()

def copyall(ip='147.8.182.2', username='lok419',password='12345678',port=50888):
    # except the data files (csv)
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, username=username, password=password,port=port)
    print('Successfully connected to 147.8.182.2')
    sftp = client.open_sftp()
    # copy all python code
    # exclude_prefixes = ('__', '.')
    for path, dirs, files in os.walk('.'):
        for f in files:
            if path not in ['.\.idea','./.idea','.\__pycache__', './__pycache__',
                            '.\data\stock','./data/stock','.\data','./data',
                            './test','.\\test']:
                local = (path+'/'+f).replace('\\','/')
                sftp.put(local,local)
                print('copied',local)
    sftp.close()
    client.close()

def copy_data(ip='147.8.182.2', username='lok419',password='12345678',port=50888):
```

```
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, username=username, password=password,port=port)
    print('Successfully connected to 147.8.182.2')
    sftp = client.open_sftp()
    # copy all python code
    # exclude_prefixes = ('__', '.')
    for path, dirs, files in os.walk('.'):
        for f in files:
            if path in ['.\data\stock','./data/stock','.\data','./data']:
                local = (path+'/'+f).replace('\\','/')
                sftp.put(local,local)
                print('copied',local)
    sftp.close()
    client.close()

def get(local, server, ip='147.8.182.2', username='lok419',password='12345678',port=50888):

    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, username=username, password=password,port=port)
    print('Successfully connected to 147.8.182.2')
    sftp = client.open_sftp()
    sftp.get(server,local)
    print('Successfully received from %s to %s'%(server,local))
    sftp.close()

    client.close()

def getall(local, server, ip='147.8.182.2', username='lok419',password='12345678',port=50888):

    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, username=username, password=password,port=port)
    print('Successfully connected to 147.8.182.2')
    stdin, stdout, stderr = client.exec_command('cd {}; ls'.format(server))
    filelist = []
    for line in iter(stdout.readline, ""):
        if '.' in line:
            filelist.append(line[:-1])

    sftp = client.open_sftp()
    for file in filelist:
        print(file)
        sftp.get(server+'/'+file, local+'/'+file)

    print('Successfully received from %s/ to %s/'%(server,local))
    sftp.close()
    client.close()

def removeall(ip='147.8.182.2', username='lok419',password='12345678',port=50888):
    # remove all except the data files (csv)
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, username=username, password=password,port=port)
    print('Successfully connected to 147.8.182.2')

    client.exec_command('rm -rf *.py')
    client.exec_command('rm -rf *.pickle')
    client.exec_command('rm -rf result')
    client.exec_command('rm -rf model')
    client.exec_command('mkdir model')
    client.exec_command('mkdir result')
    client.exec_command('mkdir ticker')
    print('Successfully removed all files')

    client.close()

def remove_data(ip='147.8.182.2', username='lok419',password='12345678',port=50888):
    # remove all data files
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, username=username, password=password,port=port)
```

```
    print('Successfully connected to 147.8.182.2')

    client.exec_command('rm -rf data')
    client.exec_command('mkdir data')
    client.exec_command('cd data; mkdir stock')
    print('Successfully removed all data files')

    client.close()

def connect(ip='147.8.182.2', username='lok419',password='12345678',port=50888):

    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, username=username, password=password,port=port)
    print('Successfully connected to 147.8.182.2')

    cd_path = []

    while True:
        cmd = input("->  ")

        if cmd == 'exit':
            client.close()
            sys.exit(0)

        if 'cd' in cmd.split(' '):
            cd_path.append('cd')
            cd_path.append( cmd.split(' ')[-1] )
            cd_path.append(';')

        else:
            stdin, stdout, stderr = client.exec_command(' '.join([' '.join(cd_path),cmd]))
            for line in iter(stdout.readline, ""):
                print('... ', line, end="")
            for line in iter(stderr.readline, ""):
                print('... ', line, end="")


if __name__ == "__main__":
    # local path: C:\Users\Cheung\PycharmProjects\DDPG\
    # server path: home\lok419\

    copy(local = 'Stock_prediction.py', server = 'Stock_prediction.py')
```