Cover Page

Name: Xiaoqiong Zhang

Student ID: 7965001610

Term: Spring 2020

Project Title: An Adaptive Trading System with Recurrent Reinforcement Learning for Index

Date: April 17, 2020

Abstract

In this research, I am considering the application of artificial intelligence in trading to see how it reacts to the dynamic financial market without human interference. The trading agent developed for the S&P 500 Index is based on Policy Gradient Actor-Critic reinforcement learning algorithm with a recurrent neural network to make a sequence of trading decisions to maximize the cumulative return. For both training and back-testing periods, the adaptive trading policy achieved higher cumulative returns than the benchmark, which is the goal of the trading system. In general, the trading agent tends to achieve an excellent return performance with a high level of risk by making more profit when the market is promising.

I. Introduction

Nowadays, algorithm trading is being widely used in quantitative investing. There has been a steady increase in interest from both specialists and scholars in the application of artificial intelligence into financial markets. However, lots of trading algorithms are designed with supervised learning models with labeled data which is given by the investment experts. The labeled data might be constrained by the knowledge and experience of investors and tends to be irrational sometimes like at the beginning of the financial crisis in 2007. As a consequence, those models trained by labeled data might be unreliable and have the risk to make worse performance.

How to develop a strategy without feedback from human beings? Reinforcement learning models provide an opportunity for the trading agent to develop a strategy (which is called policy in reinforcement learning) from history data by itself with the feedback of the true market. Currently, reinforcement learning also has attracted an increasing number of financial traders and researchers after Lee Sedol, the strongest human contemporary Go board game player was defeated by AlphaGo, the computer program with a reinforcement learning algorithm. As we know, financial trading is a zero-sum game in some sense if without considering transaction fees, which is similar to the Go board game but with numerous players at the same time. It's interesting and meaningful to construct a reinforcement learning-based trading system.

The project focuses on the construction of a deep reinforcement learning algorithm with a recurrent neural network for the equity index investment purpose. Reinforcement learning explicitly takes into account the whole problem of a goal-directed agent interacting with an uncertain environment[1]. It makes trading decisions, receives feedback on its performance, and then adjusts its internal parameters to increase its future profit. In financial markets, past decisions are necessary inputs to the trading system, which results in the recurrent decision system[2]. The problem is how to define the adaptive trading system with deep reinforcement learning and recurrent neural network.

In the past two decades, several RRL-based trading strategies with different techniques have been proposed, mainly in FX and HFT markets. Most studies agree that reinforcement learning finds approximate solutions to stochastic dynamic programming problems, but several essential components have to be adequately designed in the trading system in addition to the core algorithm. Otherwise, those components can significantly compromise the benefits of the advanced algorithm.

Formally, a basic reinforcement learning architecture consists of a set of environment states S, a set of agent actions A and a scalar reward or value function R[1]. Its goal is to select actions to maximize cumulative rewards. However, actions in financial markets have long term consequences, so the reward for each action may be delayed. It's necessary to train the system on how total rewards are allocated to each step.

Besides, the financial market is a partially observable environment with dynamic time-varying patterns, which means the agent indirectly observes the environment and adapts to a new environment dynamically. The agent needs to discover a good policy from the experience of the environment without losing too much reward. This is the balance between exploration and exploitation, which is one of the key considerations in reinforcement learning. Exploration finds more new information about the environment, while exploitation exploits known information to maximize reward[1]. It is usually important to explore as well as to exploit. Moreover, almost all reinforcement learning problems can be formalized as a Markov Decision Process (MDP), which formally describes the environment of reinforcement learning. The difficulty of modeling the states of financial markets comes from its partially observable and continuous-time MDP.

Moreover, how to determine the return function for the trading system has to be considered. In the financial market, the return is not the only performance indicator. Risk, transaction costs, time are all important factors to be measure. Moody found that maximizing the differential Sharpe ratio yields more consistent results than maximizing profits[3]. Almahdi, Steven, and Yang applied the recurrent reinforcement learning method with a statistically coherent downside risk-adjusted performance objective function under different transaction cost scenarios[4].

Therefore, this research aims to construct an adaptive trading system that makes sequences of trading decisions based on its evolutional policy developed by a deep reinforcement learning approach with a recurrent neural network. According to this purpose and the characteristic of financial data, I am going to use the Policy Gradient Actor-Critic reinforcement learning model. This model is introduced in detail in the IV part. The outcomes and performance of the trading system is shown in the V part. The evaluation is the most used performance assessment method and compared with the benchmark performance of simple Buy and Hold strategy. The comparison involves the annualized rate of return, annualized standard deviation, Sharpe Ratio, Maximum Drawdown, and VaR, etc.

II. Problem Statement

As I mentioned in the first part, the project proposes an adaptive trading system that uses a deep reinforcement learning approach with a recurrent neural network to make appropriate trading decisions. The agent interacts with the financial world unsure of dynamics, but to well define the reinforcement learning-based trading architecture, it's important to design key components including the states, actions, and reward. I have to consider how to define the financial environment and how to understand and solve the interaction between the environment and the trading agent.

The design of the environment requires defining the state, action space and reward function to implement. First, the states in the financial market are dynamic and insufficient to be defined within one dimension, so the state in my trading system is composed of four variables: close price, trading volume,

open price, and reward of the previous action. In particular, the reward of the previous action is the key input of recurrent reinforcement learning. This project chooses the recurrent neural network as the parameterized function to generate states from those features. Second, there are three actions of the trading system: hold, buy and sell. Third, the reward is determined by the return function that calculates return from trading in the financial market. Moreover, I don't allow short selling in the evolutional policy of the trading system and the trading cost is 10 bps per share.

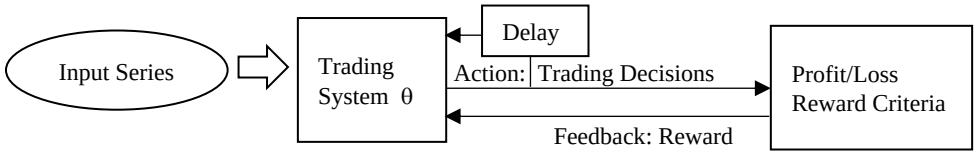

Figure 1 Framework of RRL-based trading system

As Moody's framework of a recurrent reinforcement learning-based trading system[3] indicated in Figure X, multiple state features are the input series to the trading system, and the trading decision made by the agent is imported to the environment to generate reward based on value function toward next state. In other words, Actions are actually given by the dynamic function along with the current state to produce the next state, and the next state is passed to reward function that encodes the desired behavior.

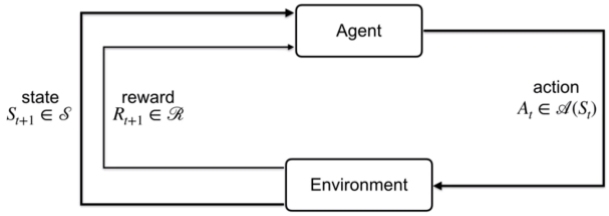

Figure 2 Interaction between agent and environment under the MDP framework

The environment of the adaptive trading system can be formed as the Markov Decision Processes (MDP), where the future is independent of the past given the present information. MDP formalized the interaction between the environment and agent with a general framework for the trading system. In this frame, the trading agent and financial market interact at discrete time steps to take actions which are trading decisions. I set the time step to one day, so our trading system updates the trading decision every day. At each day, the agent receives a state S_t that is featured by some variables from the environment and selects an action A_t from the action space A. One day later, based on the trading decision, the agent finds itself in a new state S_{t+1}. The environment also provides the agent a reward R_{t+1} draw from a set of possible rewards. This is the sequence of MDP: $\{\dots S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}, \dots\}$

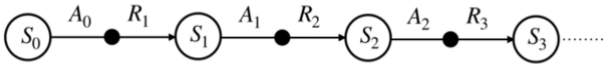

Figure 3 Observed Sequences of Markov Decision Processes

The agent-environment interaction shown above generates a trajectory of evolutional experience consisting of states, actions, and rewards. Actions influence immediate rewards as well as future states, and through those, future rewards. Besides, the Markov property of MDP indicates the present state is sufficient and remembering earlier states, so the present state contains all the information necessary to predict the future.

How to solve the MDP depends on the characteristics of state space and policy of the trading system. The Policy Gradient Actor-Critic algorithm would be a good fit to solve it. As I mentioned above, states from the environment are generated by multiple series of features. The goal of the system is to find the optimal policy or strategy to maximum reward, so I haven't defined any specific policy of the system to decide which action to take. Therefore, the deep recurrent reinforcement learning algorithm I select to solve the interaction between environment and agent is the Policy Gradient Actor-Critic model which is introduced more in detail at IV part.

III. Data Collection

Data in this project is used to generate states and rewards. Data of feature variables is required to develop the parameterized function of the state, and data of the financial market is necessary to calculate rewards of action from a state. Feature variables of states involve close price, trading volume, open price, the reward of the previous action, and the return function needs a daily rate of return of the asset to calculate reward and risk-free rate as the discount factor.

This research uses the S&P 500 as the trading asset of the adaptive trading system. The last paragraph lists all categories of data required, which are close price, trading volume, open price, and daily returns (daily rate of return, overnight rate of return, day rate of return) of S&P 500 and the risk-free rate. Training data is from 6/1/1962 to 12/31/2010 and the testing data is from 1/1/2010 to 1/1/2020. The adaptive trading system requires a large number of data to learn and improve its policy, and the financial data is expensive and limited, so a large proportion of data is used to train the model and get the optimal policy. Besides, I am going to add Experience Replay that allows the system to remember potential useful transitions from many time steps ago. The size of the buffer inside the Experience Replay is 3000, which is around 10 years.

Return features including daily rate of return $RETRN_t$, day rate of return DR_t, and overnight rate of return NR_t are calculated with close price at previous day CP_{t-1} and close price CP_t and open price today OP_t by financial equations as follow:

$$RETRN_t = \frac{CP_t - CP_{t-1}}{CP_{t-1}}, \qquad NR_t = \frac{OP_t - CP_{t-1}}{CP_{t-1}}, \qquad DR_t = \frac{CP_t - OP_t}{OP_t}$$

Except reward, features variables of states have to be normalized before entering neural network to associate with actions. One-month (20 days) is the scale range of all daily inputs. So, data is rescaled by the moving average and the moving standard deviation to a normal form. Here is the process of standardization.

$$MEAN_t = \frac{1}{20}\sum_{i=1}^{20} Variable_{t-i}$$

$$STD_t = \sqrt[20]{\frac{1}{20}\sum_{i=1}^{20}(Variable_{t-i} - MEAN_t)^2}$$

$$Normalized\ Variable_t = \frac{Variable_t - MEAN_t}{STD_t}$$

Since the start day of available data is 6/1/1962 and one-month data has to be used to calculate mean and standard deviation, the training period is actually from 6/1/1962. Discount factor is going to use ten-years treasury bond yield as the risk-free rate, so it is 0.98 in this project.

The average ten-years treasury bond yield is around 2%, which is found at the official website of U.S. Department of the Treasury. Except that, all data used in this project is downloaded from the website of Yahoo Finance and calculated in python.

IV. Method Used

This project constructs an adaptive trading system where agent makes trading decision based on its evolutional and optimal policy. Recurrent Neural Network is used to associate state feature variables with actions, Policy Gradient Actor- Critic algorithm is built to solve the interaction between agent and environment, and Experience Relay is required to improve training efficiency of limited sample dataset. This part introduces those methods in detail and describe how they match and achieve the goal of the research.

1. Agent Reward criteria

In this project, I design a new return function to calculate the reward of an action from a state to the next state.

	t = 0		t = 1		t = 2	…	t
Trading Signal& Shares	N_0		$\Delta_1 = 1$ ($N_1 = N_0 + \Delta_1$)		$\Delta_2 = -1$ ($N_2 = N_1 + \Delta_2$)	…	$\Delta_t = 0, -1, 1$ ($N_t = N_{t-1} + \Delta t$)
Open Price	$P_{o,0}$		$P_{o,1}$		$P_{o,2}$	…	$P_{o,t}$
Close Price	$P_{c,0}$		$P_{c,1}$		$P_{c,2}$	…	$P_{c,t}$
Return Function	$(P_{c,0} - P_{o,0})N_0$		$(P_{c,1} - P_{c,0})(N_0 + \Delta_1) + (P_{c,1} - P_{o,1})\Delta_1$		$(P_{c,2} - P_{c,1})(N_1+\Delta 2) + (P_{c,2} - P_{o,2})\Delta_2$	…	$(P_{c,t} - P_{c,t-1})(N_{t-1}+\Delta_t) + (P_{c,t} - P_{o,t})\Delta_t$
Reward	$(P_{c,0} - P_{o,0})N_0$	*RRL* / *Environment*	$(P_{c,1} - P_{c,0})N_1 + (P_{c,1} - P_{o,1})\Delta_1$		$(P_{c,2} - P_{c,1})N_2 - (P_{c,2} - P_{o,2})\Delta_2$	…	$(P_{c,t} - P_{c,t-1})N_t + (P_{c,t} - P_{o,t})\Delta_t$

Figure 4 Reward criteria for the trading system

To make trading decisions (Δ_t), the agent needs to know the open price $P_{o,t}$, close price $P_{c,t-1}$ and $P_{c,t}$ and trading volume at the previous day. Then, the return function is:

$$R_t = (P_{c,t} - P_{c,t-1})\, N_t + (P_{c,t} - P_{o,t})\Delta_t$$

The reward G_t can be decomposed into immediate reward calculated by the return function plus discounted reward of successor state.

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+1} + \cdots$$

2. State coding with Neural Networks

In financial market, we clearly cannot store a table entry for every possible states of an agent. It's necessary to have a general way to approximate the state-value function by incorporating a set of real valued weights of feature variables, which can be adjusted to change the function. Both linear and non-linear parameterized functions can be used, but neural network is more appropriate to the complex and dynamic financial environment. Neural networks provide a strategy for learning a useful set of features.

Neural network makes it easier to approximate complex functions. One reason for this is that the depth allows composition of multiple features, which can produce more specialized states by combining modular components. By adding more layers or more units to each layer, we can represent more complex functions. Besides, the depth can also be helpful for obtaining abstractions. This is because neural networks compose many layers of lower-level abstractions with each successive layer contributing to increasingly abstract representations. Overall, neural network can significantly improve the agent's ability to learn state features.

For the feed-forward neural networks, a set of state feature vectors as input is passed to the network and all connections in the network correspond to real valued weights. It's important for neural network to specify the initial weights. In the research, weights are initialized with random distribution. Then, the weight is multiplied by its corresponding input, and the weighted sum of input is passed to a non-linear activation function resulting in a non-linear function of inputs. Specifically, the non-linear activation function used in the project is the rectified linear unit (ReLU) function. This process is done again and again for each node in the layer and transforms the input state variables through a sequence of layers.

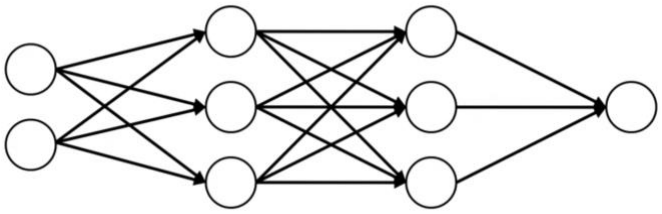

Figure 5 Neural Network Architecture

Neural network with the input of reward from previous state can be regarded as recurrent neural networks, which are a class of neural networks that allow previous outputs to be used as inputs. The final output of the neural network is state-value function V(St), given state features at time t. Then, agent has to

use backward propagation of errors method such as gradient descent and select optimization method like Adam optimization to train the networks to get the optimal parameterized state function. Backpropagation and Adam optimization are introduced the next part.

3. Policy Gradient Actor-Critic Algorithm

To solve Markov Decision Processes in the project, I use the policy-gradient actor-critic algorithms that can operate over continuous state space to find optimal policy. It's off-policy and model-free. This algorithm uses a stochastic behavior policy for good exploration and utilizes a form of policy iteration to estimate a deterministic target policy. An actor-critic algorithm includes two neural networks: Actor and Critic. These two parts are learning simultaneously. The Critic tries to approximate the value function of the action coming from the Actor component, while the Actor tries to develop the optimal policy at time step t considering the Q-value from Critic component. Critic learning uses an update rule of parameters involved in action selection and usually is based on the TD error signal[5].

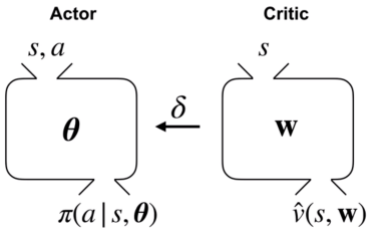

Figure 6 Critic part the Actor-Critic algorithm

As indicated in Figure X, the parameterized policy plays the role of an actor, while the value function plays the role of a critic, evaluating the actions selected by the actor. Parameter Theta in the Actor network and parameter W in the Critic network needs to update simultaneous under this framework. TD error δ is the key connection between Critic and Actor networks.

3.1 Critic Network

To train the Critic network, there are several state value learning algorithms. Considering dimension of state inputs and function complexity in real financial market, the research is going to use recurrent neural network with semi-gradient TD (0) method. The process of recurrent neural network to transform state features to state value function is discussed above. Then, it focuses on backpropagation and optimization, given selected action and policy from the Actor Policy.

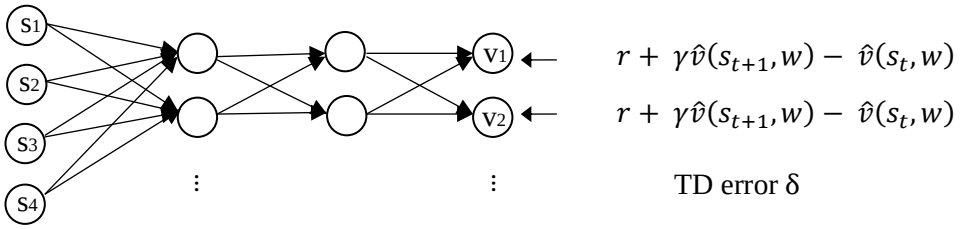

Figure 7 Critic network: neural network with TD error

3.1.1 One Step Temporal Difference Error

Temporal Difference learning is a sample-based and model-free reinforcement learning approach which learn from the current estimate of value function by bootstrapping. One-step Temporal Difference $TD(0)$ updates the value of one state towards to its own estimate of the value in the next state. The key idea of $TD(0)$ in the project is the TD error δ, which is used not only to backpropagate the neural network in the Critic, but also to improve policy in the Actor network.

Update function of $TD(0)$ in the network with learning rate α:

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

Then, TD error δ is defined with discount factor γ as:

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

Temporal Difference learning provides update function of weights, which is called semi-gradient TD. Due to the update function of $TD(0)$, there is only semi-gradient for temporal difference $\nabla\widehat{\mathrm{V}}(S, W)$.

$$W \leftarrow W + \alpha\delta\nabla\widehat{\mathrm{V}}(S, W)$$

3.1.2 Adam Optimization

Besides of the calculation of TD error δ_t, another goal in Critic network of TD agent is to update weights in neural networks, which is an optimization problem. More precisely, the neural networks modify weights in each layer to reduce the TD error on each time step. Weights updated for the output correspond to the action that was selected in the Actor network. Adam optimization is an advanced and widely used optimization method to update weights in the neural networks, which is used to optimize both Critic and Actor networks parameters in the research.

The Adam algorithm combines both momentum and vector step-sizes ideas in the update. It keeps a moving average of the gradients to compute the momentum. The β_m parameter is a meta-parameter that controls the amount of momentum. Adam algorithm also keeps a moving average of the square of the gradient, this gives us a vector of step-sizes. This update typically results in more data-efficient learning because each update is more effective.

Momentum can accelerate the learning, especially when the TD error is in a flat region. Here is the update function of momentum effect with the mean decay rate β_m:

$$m_{t+1} = \beta_m m_t + (1 - \beta_m)\Delta_{w,t}$$

$$\widehat{m_t} = {}^{m_t}/_{1 - \beta_m^t}$$

Each network has its own step-size adapted based on the statistics of the learning process. Then, it can make larger updates to some weights and smaller updates to the others. Here is the update function of vector step-sizes with the second momentum decay rate β_v:

$$v_{t+1} = \beta_v v_t + (1 + \beta_v)\Delta_{w,t})^2$$

$$\widehat{v_t} = {}^{v_t}/_{1 - \beta_v^t}$$

Weight updates of Adam combing momentum effect and vector step-sizes, given small offset ε and global step-size η:

$$w_{t+1} = w_t - \eta \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \varepsilon}$$

3.2 Actor Network

As for the Actor network, the goal is to update θ of the parameterized policy $\pi(a|S,\theta)$, so it eventually converge to the optimal policy. Policy Gradient is method that this project chooses to update the parameter θ, given TD error δ_t from the Critic network.

$$\theta_{t+1} = \theta_t + \alpha \nabla \ln \pi(A_t|S_t,\theta_t)\, \delta_t$$

Actor network also use Adam algorithm as the optimizer to update θ inside the policy. The detailed introduction of Adam optimization is on the previous part.

3.3 Actor-Critic Network summary

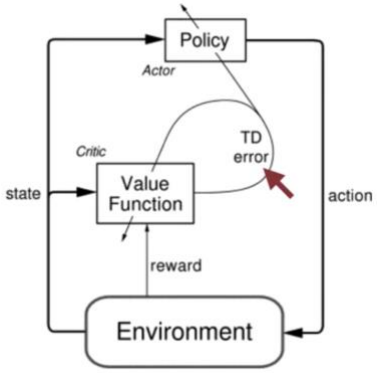

Figure 8 How the actor network and critic network interact

The TD error is used to decide how good the action was compared to the average for that state. If the TD error is positive, then it means the selected action resulted in a higher value than expected. Taking that action more often should improve our policy. That is exactly what this update does. It changes the policy parameters to increase the probability of actions that were better than expected according to the critic.

Correspondingly, if the critic is disappointed and the TD error is negative, then the probability of the action is decreased. The actor and the critic learn at the same time, constantly interacting. The actor is continually changing the policy to exceed the critics expectation, and the critic is constantly updating its value function to evaluate the actors changing policy.

Table 1 Pseudocode of Policy Gradient Actor-Critic Algorithm[1]

Actor-Critic (Continuing), for estimating $\pi_\theta \approx \pi_*$
Input: a parameterized epsilon-greedy policy $\pi(a\|S,\theta)$

Input: a parameterized recurrent neural network state-value function $\widehat{V}(S, W)$
Initialize state-value weights w and θ
Algorithm parameters: $\gamma > 0,\ \alpha^w > 0, \alpha^\theta > 0$
Loop for number of episodes (for each time step):
$A \sim \pi(\cdot \mid S, \theta)$
Take the action A, observe S', R
$\delta \leftarrow R + \gamma \hat{V}(S', w) - \hat{V}(S, w)$
$w \leftarrow w + \alpha^w \delta \nabla \hat{V}(S, w)$ with Adam optimizer
$\theta \leftarrow \theta + \alpha^\theta \delta \nabla \ln \pi(a \mid S, \theta)$ with Adam optimizer
$S \leftarrow S'$

4. Experience Replay

Experience Replay is a way to make the trading agent more sample efficient when using function approximation. The basic idea is to save a buffer of experience and then simulate more experience with a model. Then, the agent applies the functions learned from simulated experience into the sample experience from this buffer and updates the approximation of functions.

Experience is stored into a buffer when the agent interacts with the world. During the interaction, the trading agent observes a state, action, reward, next state. As the agent continue interacting with the world, more samples are added into this buffer until it's eventually filled up. When this happens, the agent can pick an older experienced to delete and replace with a new experience.

The size of the buffer has to be considered. Larger buffer can remember more potentially useful transitions from many times steps ago. However, the amount of memory used by large buffers and computational impacts from storing and accessing large buffers are key limitations. In this project, the buffer contains 3000 time-steps of data, which is around 10 years.

V. Results

The adaptive trading agent in the project trades the S&P 500 index with 1 million initial investment. No short selling is allowed in the trading system and the trading cost is 10 bps per share.

1. Train performance *(1962~2009)*

Table 2 Annual return of agent policy and buy & hold strategy during the train periods

Year	Agent Policy (%)	Buy & Hold (%)	Year	Agent Policy (%)	Buy & Hold (%)	Year	Agent Policy (%)	Buy & Hold (%)
1962	0.12	17.75	1978	-1.34	1.06	*1994*	-4.60	-1.54
1963	10.44	18.89	**1979**	28.72	12.31	**1995**	78.82	34.11
1964	-2.57	12.97	**1980**	54.16	25.77	**1996**	42.68	20.26
1965	5.21	9.06	*1981*	-14.93	-9.73	1997	65.96	31.01
1966	-7.56	-13.09	**1982**	25.40	14.76	**1998**	53.83	26.67
1967	21.52	20.09	**1983**	37.37	17.27	**1999**	38.30	19.53

1968	16.45	7.66	1984	-1.84	1.40	*2000*	-23.38	-10.14
1969	-9.16	-11.36	**1985**	61.11	26.33	*2001*	-28.05	-13.04
1970	-0.77	0.10	**1986**	36.39	14.62	*2002*	-43.56	-23.37
1971	1.44	10.79	1987	-9.02	2.03	**2003**	55.10	26.38
1972	33.80	15.63	**1988**	20.29	12.40	**2004**	17.20	8.99
1973	-34.50	-17.37	**1989**	58.05	27.25	2005	4.90	3.00
1974	-48.50	-29.72	*1990*	-15.76	-6.56	**2006**	28.81	13.62
1975	28.23	31.55	**1991**	57.65	26.31	**2007**	5.63	3.53
1976	38.74	19.15	**1992**	8.07	4.46	*2008*	-64.19	-38.49
1977	-23.16	-11.50	**1993**	14.17	7.06	**2009**	59.96	23.45

Table 1 shows that during the training period, the agent performed much better than the benchmark when the market was promising, while the agent experienced more loss than the benchmark when the market was in the downside. And this circumstance becomed more and more significant as an increasing number of years had been trained to the trading agent.

Table 3 Performance summary of agent and benchmark of train data

Measurement	Agent Policy	Benchmark (B&H)
Statistics		
Mean	14.08%	7.94%
Std	30.54%	16.76%
Min	-64.19%	-38.49%
Max	78.82%	34.11%
Return		
Annualized Return	9.84%	6.53%
Cumulative Return (million)	58.31	20.81
Sharpe Ratio	1.5186	1.8529
Risk		
Annualized Volatility	6.70%	3.52%
VaR	-2.88%	-1.50%
CVaR	-4.44%	-2.31%
Maximum Drawdown	83.37%	56.78%

Table 2 summarizes the performance of the evolutional policy learned in the trading system and compares them with the benchmark from the perspectives of statistics, return, and risk. The agent had a higher mean, higher standard deviation, higher maximum, and lower minimum value than the benchmark, indicating that the trading agent tends to higher return with high risks.

The agent had a 9.84% annualized rate of return with a 6.7% annualized standard deviation. And its Sharpe ratio was a bit lower than the benchmark even though 1.52 looks acceptable. This is might due to insufficient learning at the beginning of training and the high risk of the whole trading policy. As we can see from the first column in Table 1, the performance of the trading agent in the first 15 years of the training was not good. And the return in the last 30 years of training had huge volatility, which leads to the high risk of the trading policy. The Maximum Drawdown arrived at 83.37%, and VaR was twice times of the benchmark.

Figure 6 plots the cumulative value and maximum drawdown of the agent and benchmark. The agent reached the highest value on March 24, 2000, and experienced the maximum drawdown on March 9, 2009. The maximum drawdown period for the buy and hold benchmark strategy was from October 9, 2007, to March 9, 2009. Though the agent and benchmark didn't have the same day to reach a maximum point, they both experienced huge loss on the same day during the training period.

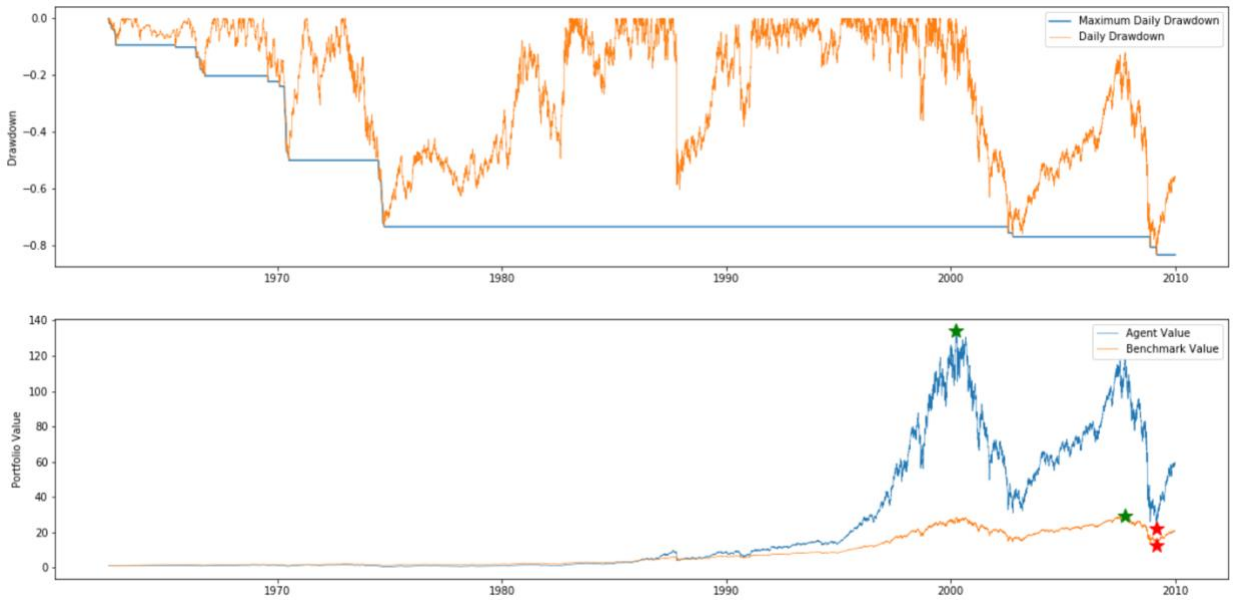

Figure 9 S&P500 Cumulative Value and Maximum Drawdown (Train)

Cumulative return is the most important return indicators in the project. The trading agent and benchmark both started with 1 million initial investment, but the trading agent achieved 58.31 million cumulative value, which was around 3 times the amount earned by the benchmark at the end of the training period. Considering the goal of the agent, which is to find an optimal policy to maximize cumulative return, this output does indicate that the adaptive trading agent with reinforcement learning is able to perform well to get a high cumulative return.

2. Test Performance *(2010~2019)*

Table 4 Annual return of agent and benchmark of test data

Year	Agent Policy (%)	Buy & Hold (%)	Year	Agent Policy (%)	Buy & Hold (%)
2010	25.32	12.78	2015	-5.00	-0.73
2011	-2.37	-0.01	2016	12.42	9.54
2012	26.51	13.41	2017	17.85	19.42
2013	57.80	29.60	2018	-26.00	-6.24
2014	15.91	11.39	2019	34.61	28.50

The trend of test performance looks similar to the train performance. The agent performed better than the benchmark when the market was increasing and lost more when the market was depressing. It

seems that the optimal strategy developed by the reinforcement learning-based trading agent is to make a high return by experiencing high risks.

Table 5 Performance summary of agent and benchmark of test data

Measurement	Agent Policy	Benchmark (B&H)
Statistics		
Mean	15.51%	11.77%
Std	20.19%	11.95%
Min	-26.00%	-6.24%
Max	57.80%	29.60%
Return		
Annualized Return	14.56%	10.19%
Cumulative Return (million)	3.27	2.89
Sharpe Ratio	2.45	3.32
Risk		
Annualized Volatility	5.94%	3.22%
VaR	-2.96%	-1.54%
CVaR	-4.30%	-2.31%
Maximum Drawdown	42.07%	19.78%

According to *Table 4*, the test performance of the agent policy earned a high average return with huge volatility. The agent achieved a 14.56% annualized rate of return with a 5.94% standard deviation, and both are higher than the benchmark. The risk taken by the agent was significant. The VaR of trading policy was negative 3%, which was twice larger than the benchmark. It means the agent tends to lose the double amount of money at the worse market condition if the agent and benchmark have the same investment value.

The Maximum Drawdown of the agent was more than twice the amount of the benchmark. Agent reached the highest value on January 26, 2018 and experienced the maximum drawdown on December 24, 2018. The maximum drawdown period for the benchmark was from September 20, 2018 to December 24, 2018. In addition to the training period, both the agent and benchmark also experienced the maximum drawdown on the same day at the test period.

Figure 10 S&P500 Cumulative Value and Maximum Drawdown (Test)

Even though the Sharpe ratio of the agent in the past decade was relatively lower than the benchmark, the trading policy still had a higher cumulative return. Figure 9 shows that the cumulative return of the trading agent was much higher than the benchmark before 2018, but it dropped a lot to a value similar to the benchmark on December 24, 2018, then it continued to exceed the benchmark after experiencing a huge drawdown.

In sum, the agent performed better than the simple buy and hold strategy to achieve a high return in a relatively risky manner. Since the goal set in the project is to develop an optimal policy in the trading system to maximize the cumulative return, the agent constructed by the Gradient Policy Actor-Critic reinforcement learning algorithm does make sequences of trading decisions based on its adaptive policy to get a high cumulative return.

VI. Conclusions

In this paper, I proposed and tested an adaptive trading agent with recurrent reinforcement learning for trading the S&P 500 index. Policy Gradient Actor-Critic with a recurrent neural network is one of the most suitable methods to solve the financial Markov Decision Processes. The ultimate goal of training the Actor-Critic-based agent is to find an optimal trading policy to maximize the cumulative return. Several state features are selected to depict the dynamic environment of the financial market, and they are input into the critic network to calculate the Temporal Difference error to update parameters in the Actor network. Trading decisions made in the actor network update parameters in the Critic network in turn. This process is done again and again to get the most optimal policy in the adaptive trading system. Then, the policy trained from 1962 to 2010 is back-tested in the past decade.

The results of the trading agent with the Policy Gradient Actor-Critic reinforcement learning method are promising. This agent did make a huge amount of cumulative return during both training and

testing periods. Besides, the trading policy of the agent was able to beat the benchmark and get a much higher return when the market was strong, while it lost more when the market was weak. The agent achieved an excellent rate of return by taking a high level of risk. All risk measurements including volatility, VaR, and Maximum Drawdown for the trading system were at least twice times of those for the benchmark. In sum, the trading strategy of the reinforcement learning-based agent can achieve high returns with high risks.

This reinforcement learning-based trading agent is suitable for long-term investors, who don't care much about short-term returns. This is because the agent can have enough data to train, and the agent does have the ability to make a very high cumulative return in a long run. Moreover, the agent can be modified and customized by designing additional risk constraints into the return function to match a variety of needs from different investors.

VII. Reference

[1] Richard S. Sutton, Andrew G. Barto, Reinforcement learning: An introduction, volume 1. MIT press Cambridge, 1998.

[2] Dimitri P. Bertsekas, Dynamic Programming and Optimal Control, Athena Scientific, Belmont, MA, 1995.

[3] John Moody, Wu Lizhong, Liao Yuansong, Mattew Saffell, Performance functions and reinforcement learning for trading systems and portfolios, Journal of Forecast. 1998, 17, 441–470.

[4] Saud Almahdi, 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, 2017, 87, 267–279.

[5] Stelios D.Bekiros, Heterogeneous trading strategies with adaptive fuzzy Actor–Critic reinforcement learning: A behavioral approach, Journal of Economic Dynamics & Control, 2010, 34, 1153–1170.

Appendix

```
import numpy as np
import pandas as pd
np.random.seed(0)

original_data = pd.read_csv('^GSPC.csv')

# Helper function
def _sharpe(Returns, freq=252) :
    """Given a set of returns, calculates naive (rfr=0) sharpe """
    return (np.sqrt(freq) * np.mean(Returns))/np.std(Returns)

def _prices2returns(prices):
    px = pd.DataFrame(prices)
    nl = px.shift().fillna(0)
    R = ((px - nl)/nl).fillna(0).replace([np.inf, -np.inf], np.nan).dropna()
    R = np.append( R[0].values, 0)
    return R

def Maximum_DD(data):
    Roll_Max = data.cummax()
    Daily_DD = -(data/Roll_Max - 1.0)
    MaxDD = Daily_DD.max()
    Daily_MaxDD = Daily_DD.cummax()

    d2 = Daily_DD.idxmax()
    d1 = Roll_Max.loc[:d2].idxmax()
    d1_value = data.loc[d1]
    d2_value = data.loc[d2]
    return MaxDD,Daily_MaxDD,Daily_DD,d2,d1,d1_value,d2_value

def summary_table(port_ret_data,alpha=0.05):
    port_ret_data.dropna(inplace=True)
    y_ret = (port_ret_data+1).resample('Y').prod() - 1
    y_ret = y_ret.to_period('Y')
    value = (port_ret_data+1).cumprod()

    x = y_ret.describe()
    x.drop(['count','25%','50%','75%'], inplace=True)
    x.loc['Ann_Ret'] = ((y_ret+1).prod())**(1/len(y_ret)) - 1
    x.loc['Ann_Vol'] = port_ret_data.std() * np.sqrt(12)
    x.loc['Sharpe Ratio'] = x.loc['Ann_Ret']/x.loc['Ann_Vol']
    x.loc['VaR'] = port_ret_data.quantile(alpha)
```

```
    x.loc['CVaR'] = port_ret_data[port_ret_data<=x.loc['VaR']].mean()
    x.loc['MMD'] = Maximum_DD(value)[0]
    summary = pd.concat([y_ret,x])

    return summary,value

# Data Process
class Data(object):
    MinPercentileDays = 100

    def __init__(self, ticker, days, scale=True ):
        self.days = days+1
        print(self.days)
        df = pd.read_csv(ticker +".csv")
        df = df[ ~np.isnan(df.Volume)][['Close','Volume','Open']]

        # we calculate returns and percentiles, then kill nans
        # Input
        df = df[['Close','Volume','Open']]
        df.Volume.replace(0,1,inplace=True) # days shouldn't have zero volume..

        df['Return'] = (df.Close-df.Close.shift())/df.Close.shift()
        df['NightReturn'] = (df.Open-df.Close.shift())/df.Close.shift()
        df['DayReturn'] = (df.Close-df.Open)/df.Open

        pctrank = lambda x: pd.Series(x).rank(pct=True).iloc[-1]
        df['ClosePctl'] = df.Close.expanding(self.MinPercentileDays).apply(pctrank)
        df['VolumePctl'] = df.Volume.expanding(self.MinPercentileDays).apply(pctrank)

        df.dropna(axis=0,inplace=True)
        R = df.Return
        NR = df.NightReturn
        DR = df.DayReturn

        if scale:
            mean_values = df.expanding(20).mean()
            std_values = df.expanding(20).std()
            df = (df.iloc[20:] - mean_values.iloc[20])/ std_values.iloc[20]

        df['Return'] = R # we don't want our returns scaled
        df['NightReturn'] = NR
        df['DayReturn'] = DR

        self.min_values = df.min(axis=0)
```

```
        self.max_values = df.max(axis=0)
        self.data = df
        self.step = 0
        print(len(self.data.index))

    def reset(self):
        # we want contiguous data
        self.idx = 0
        self.step = 0

    def set_test(self):
        self.idx = self.days
        self.step = self.days

    def take_step(self):
        obs = self.data.iloc[self.idx].to_numpy()
        self.idx += 1
        self.step += 1
        print(self.idx)
        next_obs = self.data.iloc[self.idx].to_numpy()
        done = self.idx >= self.days-1
        return obs,done,next_obs

    def take_test(self):
        obs = self.data.iloc[self.idx].to_numpy()
        self.idx += 1
        self.step += 1
        next_obs = self.data.iloc[self.idx].to_numpy()
        done = self.step >= len(self.data.index)-1
        return obs,done,next_obs

# Trading at one time step
class TradingSimulator(object) :
    """ Implements core trading simulator for single-instrument univ """

    def __init__(self, steps, trading_cost_bps = 1e-3, time_cost_bps = 1e-4):
        # invariant for object life
        self.trading_cost_bps = trading_cost_bps
        self.time_cost_bps    = time_cost_bps
        self.steps            = steps
        # change every step
        self.step             = 0
        self.buy_step         = 0
        self.sell_step        = 0
```

```
        self.actions          = np.zeros(self.steps)
        self.navs             = np.ones(self.steps)
        self.mkt_nav          = np.ones(self.steps)
        self.daily_retrns     = np.ones(self.steps)
        self.posns            = np.zeros(self.steps)
        self.costs            = np.zeros(self.steps)
        self.trades           = np.zeros(self.steps)
        self.mkt_retrns       = np.zeros(self.steps)

    def reset(self):
        self.step = 0
        self.buy_step         = 0
        self.sell_step        = 0
        self.actions.fill(0)
        self.navs.fill(1)
        self.mkt_nav.fill(1)
        self.daily_retrns.fill(0)
        self.posns.fill(0)
        self.costs.fill(0)
        self.trades.fill(0)
        self.mkt_retrns.fill(0)

    def take_step(self, action, nr, dr, retrn ):
        """ Given an action and return for prior period, calculates costs, navs,
            etc and returns the reward and a  summary of the day's activity. """

        bod_posn = 0.0 if self.step == 0 else self.posns[self.step-1] # begining of day
position:initialization=0 or =previous day
        total_posn = 0.0 if self.step == 0 else np.sum(self.posns[self.step-1])
        bod_nav  = 1.0 if self.step == 0 else self.navs[self.step-1] # initial principles
        mkt_nav  = 1.0 if self.step == 0 else self.mkt_nav[self.step-1] # market return

        night_ret = nr*total_posn

        self.mkt_retrns[self.step] = retrn

        if action == 2:
            self.buy_step += 1
        elif action == 0 and np.sum(self.posns) == 0: # We cannot do short selling in my strategy
            self.sell_step += 1
            action = 1

        self.actions[self.step] = action
        self.posns[self.step] = action - 1   # action is 0,1,2 --> position is -1,0,1
```

```
        self.trades[self.step] = self.posns[self.step] - bod_posn  # 0: -1,0,1; -1: 0,1,2; 1:-2,0,2

        trade_costs_pct = abs(self.trades[self.step]) * self.trading_cost_bps
        self.costs[self.step] = 0


        ret =  dr*(total_posn + action - 1)# holding return daily
        self.daily_retrns[self.step] = ret

        if self.step != 0 :
            self.navs[self.step] =  bod_nav * (1 + ret)
            self.mkt_nav[self.step] =  mkt_nav * (1 + self.mkt_retrns[self.step])

        reward = self.navs[self.step]
        info = { 'reward': reward, 'nav':self.navs[self.step], 'costs':self.costs[self.step] }

        self.step += 1
        return reward, info


    def result(self):
        """returns internal state in new dataframe """
        cols = ['action', 'bod_nav', 'mkt_nav','mkt_return','sim_return',
                'position','costs', 'trade' ]
        rets = _prices2returns(self.navs)
        #pdb.set_trace()
        df = pd.DataFrame( {'action':     self.actions, # today's action (from agent)
                            'bod_nav':    self.navs,
                            'mkt_nav':  self.mkt_nav,
                            'mkt_return': self.mkt_retrns,
                            'sim_return': self.daily_retrns,
                            'position':   self.posns,   # EOD position
                            'costs':  self.costs,   # eod costs
                            'trade':  self.trades },# eod trade
                           columns=cols)
        return df

# Create Critic network

from keras import layers, models, optimizers
from keras import backend as K

class Critic:
```

```
    def __init__(self,state_size,action_size):
        self.state_size = state_size
        self.action_size = action_size

        self.build_model()

    def build_model(self):
        states = layers.Input(shape=(self.state_size,),name='states')
        actions = layers.Input(shape=(self.action_size,),name='actions')

        net_states = layers.Dense(units=16,kernel_regularizer=layers.regularizers.l2(1e-6))(states)
        net_states = layers.BatchNormalization()(net_states)
        net_states = layers.Activation("relu")(net_states)

        net_states = layers.Dense(units=32,kernel_regularizer=layers.regularizers.l2(1e-6))(net_states)

        net_actions = layers.Dense(units=32,kernel_regularizer=layers.regularizers.l2(1e-6))(actions)
        net = layers.Add()([net_states,net_actions])
        net = layers.Activation("relu")(net)

        Q_values = layers.Dense(units=1, name='q_values', kernel_initializer =
layers.initializers.RandomUniform(minval=-0.003,maxval=0.003))(net)

        self.model = models.Model(inputs = [states,actions],outputs=Q_values)

        optimizer = optimizers.Adam(lr=0.001)
        self.model.compile(optimizer=optimizer,loss='mse')

        action_gradients = K.gradients(Q_values,actions)

        self.get_action_gradients = K.function(
            inputs = [*self.model.input, K.learning_phase()],
            outputs = action_gradients
        )

# Creat Actor network

class Actor:

    def __init__(self,state_size,action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.build_model()
```

```
    def build_model(self):
        states = layers.Input(shape=(self.state_size,),name='states')

        net = layers.Dense(units=16,kernel_regularizer=layers.regularizers.l2(1e-6))(states)
        net = layers.BatchNormalization()(net)
        net = layers.Activation("relu")(net)
        net = layers.Dense(units=32,kernel_regularizer=layers.regularizers.l2(1e-6))(net)
        net = layers.BatchNormalization()(net)
        net = layers.Activation("relu")(net)

        actions = layers.Dense(units=self.action_size,activation='softmax',name='actions')(net)
        self.model = models.Model(inputs=states, outputs=actions)

        action_gradients = layers.Input(shape=(self.action_size,))
        loss = K.mean(-action_gradients * actions)

        optimizer = optimizers.Adam(lr=.001)
        updates_op = optimizer.get_updates(params = self.model.trainable_weights, loss=loss)
        self.train_fn = K.function(
            inputs = [self.model.input,action_gradients,K.learning_phase()],
            outputs = [],
            updates = updates_op
        )

# Creat Experience Replay
import numpy as np
from numpy.random import choice
import random
from collections import namedtuple, deque

class ReplayBuffer:

    def __init__(self,buffer_size,batch_size):
        self.memory = deque(maxlen=buffer_size) # Memory size of replay buffer
        self.batch_size = batch_size # Training batch size for neural nets
        self.experience = namedtuple("Experience",field_names=['state','action','reward','next_state','done'])

    def add(self,state,action,reward,next_state,done):
        e = self.experience(state,action,reward,next_state,done)
        self.memory.append(e)

    def sample(self,batch_size=32):
        return random.sample(self.memory,k = self.batch_size)
```

```
    def __len__(self):
        return len(self.memory)

# Creat trading agent with the actor-critic network
class Agent:
    def __init__(self,time_steps,batch_size,ticker,is_eval=False):
        self.time_steps = time_steps # day
        self.src = Data(ticker,days = self.time_steps) # Data Source
        self.sim = TradingSimulator(steps = self.time_steps, trading_cost_bps=1e-3,time_cost_bps=1e-4) #
Trading eact time step
        self.test = TradingSimulator(steps = len(self.src.data.index) - self.time_steps+120,
trading_cost_bps=1e-3,time_cost_bps=1e-4)
        self.state_size = 5 # Close price and training volume

        self.action_size = 3
        self.reset()

        # Define replay memory size
        self.buffer_size = 1000000
        self.batch_size = batch_size
        self.memory = ReplayBuffer(self.buffer_size,self.batch_size)
        self.inventory = []

        self.is_eval = is_eval

        # Discount factor in Bellman equation:
        self.gamma = 0.98
        # A soft update of the actor and critic networks can be done as following:
        self.tau = 0.001

        self.actor_local = Actor(self.state_size,self.action_size)
        self.actor_target = Actor(self.state_size,self.action_size)

        self.critic_local = Critic(self.state_size,self.action_size)

        self.critic_target = Critic(self.state_size,self.action_size)
        self.critic_target.model.set_weights(self.critic_local.model.get_weights())

        # Set the target model parameters to local model parameters
        self.actor_target.model.set_weights(self.actor_local.model.get_weights())

    def reset(self):
        self.src.reset()
        self.sim.reset()
```

```
        return self.src.take_step()[0]

    def set_test(self):
        self.src.set_test()
        self.test.reset()
        return self.src.take_test()[0]

    def act(self,state):
        options = self.actor_local.model.predict(state)
        self.last_state = state
        if not self.is_eval:
            return choice(range(3),p=options[0])
        return np.argmax(options[0])

    def memory_step(self,action,reward,next_state,done):
        self.memory.add(self.last_state,action,reward,next_state,done)

        if len(self.memory)> self.batch_size:
            experiences = self.memory.sample(self.batch_size)
            self.learn(experiences)
            self.last_state = next_state

    def learn(self,experiences):
        states = np.vstack([e.state for e in experiences if e is not None]).astype(np.float32).reshape(-
1,self.state_size)
        actions = np.vstack([e.action for e in experiences if e is not None]).astype(np.float32).reshape(-
1,self.action_size)
        rewards = np.array([e.reward for e in experiences if e is not None]).astype(np.float32).reshape(-1,1)
        dones = np.array([e.done for e in experiences if e is not None]).astype(np.float32).reshape(-1,1)
        next_states = np.vstack([e.next_state for e in experiences if e is not
None]).astype(np.float32).reshape(-1,self.state_size)

        actions_next = self.actor_target.model.predict_on_batch(next_states)
        Q_targets_next = self.critic_target.model.predict_on_batch([next_states,actions_next])
        Q_targets = rewards + self.gamma * Q_targets_next * (1-dones)
        self.critic_local.model.train_on_batch(x=[states,actions],y=Q_targets)

        action_gradients = np.reshape(self.critic_local.get_action_gradients([states,actions,0]),(-
1,self.action_size))

        self.actor_local.train_fn([states,action_gradients,1])
        self.soft_update(self.actor_local.model,self.actor_target.model)

    def soft_update(self,local_model,target_model):
```

```
    local_weights = np.array(local_model.get_weights())
    target_weights = np.array(target_model.get_weights())
    assert len(local_weights) == len(target_weights)
    new_weights = self.tau * local_weights + (1-self.tau)*target_weights
    target_model.set_weights(new_weights)

def action_step(self,action):
    observation, done,next_observation = self.src.take_step()

    yret = observation[3] #retrn: return
    nr = observation[4]
    dr = observation[5]
    reward, info = self.sim.take_step(action, nr, dr,yret)
    return observation, reward, next_observation, done, info

def test_step(self,action):
    observation, done,next_observation = self.src.take_test()

    yret = observation[3] #retrn: return
    nr = observation[4]
    dr = observation[5]
    reward, info = self.test.take_step(action, nr, dr,yret)
    return observation, reward, next_observation, done, info

def run_strat(self,return_df=True):
    """run provided strategy, returns dataframe with all steps"""
    observation = self.reset()
    prev_reward = 0
    done = False
    while not done:
        # Five state features
        state_1 = observation[[0,1,2,4]].reshape(1,-1)
        state = np.append(state_1,[prev_reward]).reshape(1,-1)
        action = self.act(state) # call strategy
        action_prob = self.actor_local.model.predict(state)
        # update observation
        observation, reward, next_observation, done, info = self.action_step(action)
        print(done)
        next_state_1 = next_observation[[0,1,2,4]].reshape(1,-1)
        next_state = np.append(next_state_1,[reward]).reshape(1,-1)
        self.memory_step(action_prob,reward,next_state,done)
        prev_reward = reward
    return self.sim.result() if return_df else None
```

```
    def test_strat(self,return_df=True):
        observation = self.set_test()
        prev_reward = 0
        done = False
        while not done:
            # Five state features
            state_1 = observation[[0,1,2,4]].reshape(1,-1)
            state = np.append(state_1,[prev_reward]).reshape(1,-1)
            action = self.act(state) # call strategy
            print(action)
            # update observation
            observation, reward, next_observation, done, info = self.test_step(action)
            next_state_1 = next_observation[[0,1,2,4]].reshape(1,-1)
            next_state = np.append(next_state_1,[reward]).reshape(1,-1)
            self.memory_step(action_prob,reward,next_state,done)
            prev_reward = reward
        return self.test.result() if return_df else None

# Run the experiment
time_steps = 12100
batch_size = 3000 #Replay Buffer Size
agent = Agent(time_steps,batch_size,ticker ='^GSPC')
result_train = agent.run_strat()
result_test = agent.test_strat()

result_train.index = pd.to_datetime(original_data['Date'])[120:(120+ time_steps) ]
result_test.index = pd.to_datetime(original_data['Date'])[ (120+ time_steps):]

# Performance Comparison

train_summary,train_value = summary_table(result_train['sim_return'])
train_mkt_summary,train_mkt_value = summary_table(result_train['mkt_return'])
test_summary,test_value = summary_table(result_test['sim_return'])
test_mkt_summary,test_mkt_value = summary_table(result_test['mkt_return'])

# - Train Performance
result_train.describe()
print(train_summary)
print(train_mkt_summary)
## Maximum Drawdown
train_MaxDD, train_Daily_MaxDD, train_Daily_DD, train_d2, train_d1, train_d1_value, train_d2_value
= Maximum_DD(train_value)
train_mkt_MaxDD,train_mkt_Daily_MaxDD,train_mkt_Daily_DD,train_mkt_d2,train_mkt_d1,train_mkt
_d1_value,train_mkt_d2_value = Maximum_DD(train_mkt_value)
```

```
print('For agent:')
print('The highest value happened at the end of',train_d1)
print('The highest maximum drawdown happened at the end of',train_d2)
print('the maximum drawdown period is from %s to %s.' % (train_d1,train_d2))
print('\n')
print('For market')
print('The highest value happened at the end of',train_mkt_d1)
print('The highest maximum drawdown happened at the end of',train_mkt_d2)
print('the maximum drawdown period is from %s to %s.' % (train_mkt_d1,train_mkt_d2))
print('\n')

## Plot
fig = plt.figure(figsize=(20,10))
ax1 = fig.add_subplot(2,1,1)
ax2 = fig.add_subplot(2,1,2)

ax1.plot(-train_Daily_MaxDD,label='Maximum Daily Drawdown',linewidth=1.5)
ax1.plot(-train_Daily_DD,label='Daily Drawdown',linewidth=0.6)
ax1.set_ylabel('Drawdown')
ax1.legend()

ax2.plot(train_value,label='Agent Value',linewidth=0.6)
ax2.plot(train_mkt_value,label='Benchmark Value',linewidth=0.6)
ax2.plot(train_d1,train_d1_value,marker="*",c='g',ms=15)
ax2.plot(train_d2,train_d2_value,marker="*",c='r',ms=15)
ax2.plot(train_mkt_d1,train_mkt_d1_value,marker="*",c='g',ms=15)
ax2.plot(train_mkt_d2,train_mkt_d2_value,marker="*",c='r',ms=15)
ax2.set_ylabel('Portfolio Value')
ax2.legend(loc='upper right')

plt.show()

# - Test Performance
print(test_summary)
print(test_mkt_summary)

## Maximum Drawdown
test_MaxDD, test_Daily_MaxDD, test_Daily_DD, test_d2, test_d1, test_d1_value, test_d2_value =
Maximum_DD(test_value)
test_mkt_MaxDD, test_mkt_Daily_MaxDD, test_mkt_Daily_DD, test_mkt_d2, test_mkt_d1,
test_mkt_d1_value, test_mkt_d2_value = Maximum_DD(test_mkt_value)
print('For agent:')
print('The highest value happened at the end of',test_d1)
print('The highest maximum drawdown happened at the end of',test_d2)
```

```
print('the maximum drawdown period is from %s to %s.' % (test_d1,test_d2))
print('\n')
print('For market')
print('The highest value happened at the end of',test_mkt_d1)
print('The highest maximum drawdown happened at the end of',test_mkt_d2)
print('the maximum drawdown period is from %s to %s.' % (test_mkt_d1,test_mkt_d2))
print('\n')

## Plots
fig = plt.figure(figsize=(20,10))
ax1 = fig.add_subplot(2,1,1)
ax2 = fig.add_subplot(2,1,2)

ax1.plot(-test_Daily_MaxDD,label='Maximum Daily Drawdown',linewidth=1.5)
ax1.plot(-test_Daily_DD,label='Daily Drawdown',linewidth=0.6)
ax1.set_ylabel('Drawdown')
ax1.legend()

ax2.plot(test_value,label='Agent Value',linewidth=0.6)
ax2.plot(test_mkt_value,label='Benchmark Value',linewidth=0.6)
ax2.plot(test_d1,test_d1_value,marker="*",c='g',ms=15)
ax2.plot(test_d2,test_d2_value,marker="*",c='r',ms=15)
ax2.plot(test_mkt_d1,test_mkt_d1_value,marker="*",c='g',ms=15)
ax2.plot(test_mkt_d2,test_mkt_d2_value,marker="*",c='r',ms=15)
ax2.set_ylabel('Cumulative Value')
ax2.legend(loc='upper right')

plt.show()
```