Reinforcement Learning for Financial Portfolio Management

A study of Neural Networks for Reinforcement Learning on currency exchange market

Amin Alizadeh

Master's Thesis

Master of Engineering – Big Data Analytics

2021

MASTER'S THESIS	
Arcada	
Degree Programme:	Big Data Analytics
Identification number:	
Author:	Amin Alizadeh
Title:	Reinforcement Learning for Financial Portfolio Management A study of Neural Networks for Reinforcement Learning on currency exchange market
Supervisor (Arcada):	Magnus Westerlund
Commissioned by:	

Abstract:
Portfolio management is the process of continually reallocating funds into financial instruments, aiming to maximize the return. This paper presents a Reinforcement Learning framework where an agent interacts with the trading environment to learn a strategy for decision making. Forex exchange data for five currencies are used in this study. We briefly introduce RL methods and advance towards what practically can be implemented in this study; then, model the agent's behavior using Artificial Neural Networks, Convolutional Neural Networks, and devise an Actor-Critic approach. A variety of network topologies for each of the three approaches are studied. Furthermore, several hyperparameters, optimizers, and activation and loss functions have been applied to the models. The aim of the agent is to buy, sell, or hold the currencies to maximize the expected return of the portfolio. The performance and the profit yields of all models are evaluated. Several models from various classes of approaches, make over 10% profit with 0.1% commission rate. On the other hand, several other models result in loss or do not converge. A2C approaches are more likely to converge than CNNs and ANNs are the least likely ones to converge. We propose what modifications can be made to the framework to study to make improvements.

Keywords:	Reinforcement Learning, Decision Making, Deep Learning, Portfolio Management, Machine Learning, Artificial Neural Networks, Actor-Critic
Number of pages:	
Language:	English
Date of acceptance:	

CONTENTS

Figures

Tables

Procedures

INTRODUCTION

A Portfolio is a collection of investment assets, such as stocks, currencies, collectibles, bonds, objects, and futures grouped together. Portfolio theory discusses the relation of each asset to its history, other assets, and their history. Investing in a portfolio mitigates the risk of loss while maximizing the return. Portfolio theory helps with finding and choosing the optimal portfolio (Wang, 2013).

Portfolio management is the decision-making process of continuously reallocating an amount of fund into a number of different financial investment products, aiming to maximize the return while restraining the risk (Haugen, 1986). The funds are distributed among a set of financial instruments; then an investment strategy is devised to make decisions based on market behavior and change the distribution of wealth. The behavior of the market is not known in advance; therefore, it cannot affect the decision-making model (Agarwal, et al., 2006). The first studies of such models can be traced back to the 1950s to work by J. L. Kelly (Kelly, 1956).

Portfolio management distinguishes four approaches: 1) “Follow-the-Winner”, 2) “Follow-the-Loser”, 3) “Pattern- Matching” and 4) “Meta-Learning Algorithms” (Li & Hoi, 2014). Machine Learning approaches, and the Deep Reinforcement Learning approaches in particular, can be categorized as a combination of “Pattern-Matching” and “Meta-Learning”.

Creating autonomous agents that can interact with the environment they are in, respond to stimuli, and improve on their interaction based on the trial-and-error process has been an important aim of the field of Artificial Intelligence (AI) (Arulkumaran, et al., 2017). These agents experience the environment and evaluate what was successful and what was not. Then they use the experience they gained to decide what to do instead of explicitly being told what actions to take (Cumming, 2015).

Deep Learning has become a widely researched area of Machine Learning in recent years. A main contributing factor has been the enormous increase in computational power that was inaccessible or out of reach several years ago. Deep learning is referred to modeling using a neural network that has several, ordinarily more than one, hidden layers. They are also known as multilayer perceptrons (MLP). An MLP has an input layer where features

are fed into and an output layer where the results of the operations are returned. Between the input and the output layer are so-called sub-function layers of inter-connected perceptrons (Figure 1). These sub-function layers are also known as the *hidden* layers (Bengio, et al., 2016).

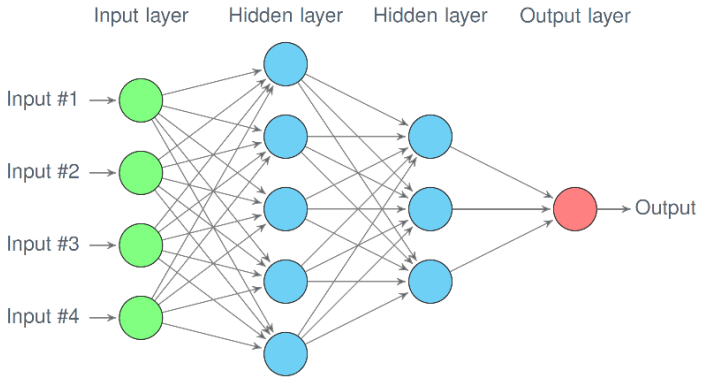


Figure 1. Multilayer Perceptron (MLP) also referred to as a Neural Network

1.1 Motivation

Trading in financial markets has attracted the attention of many scientists in the fields of Economics, Mathematics, Computer Science, Accounting, and so on, as well as many other ordinary, non-professional people. An interesting aspect of financial markets and asset trading is their unpredictable nature. They can be massively profitable and at the same time, some may bear huge losses. Such markets are excessively complex for an average person to have a good grasp of what investment can yield a profit and what can cause loss. Especially, nowadays with the advances in electronics, computer software, and high-speed networks, trading in such markets is done in a (near) real-time manner. The prices, trends, and trajectories can change in a matter of seconds which makes the task of predicting and identifying such features impossible for human beings.

With computers becoming commonplace, data becoming more accessible, and algorithms getting more sophisticated, one might wonder how to leverage all these advances to try to maximize the return of investment. A sufficiently sophisticated algorithm can observe past data, find trends and patterns, make recommendations, or trade independently. This drive gives way to algorithmic trading; particularly, the ones which utilize methods based on AI.

The complexity of algorithms may vary significantly. Some of them may employ simple analyses such as technical analysis or candlesticks based on a few positions (Pring, 2014), whilst some other approaches may take millions of points into account, make predictions of the future prices, and decide based on them (Cumming, 2015). The algorithms can be based on Artificial Neural Networks (ANN), Bayesian probability methods, Reinforcement Learning (RL), or many other derivatives of AI technics. RL methods have become a growing field of research and experimentation in algorithmic financial trading in recent years.

This research will investigate how AI methods, especially RL approaches, are used in portfolio management. Several flavors of RL with various settings and configurations will be examined. Even though reinforcement learning may utilize different prediction methods, this study will focus solely on methods using Neural Networks.

1.2 Research Questions

As stated above, this study will investigate the use of reinforcement learning, using neural networks, in algorithmic trading and financial portfolio management. This focus will open several areas of interest where the research can explore further and experiment with various setups.

In the next chapters, we would like to examine and find answers to the following questions:

1. How are various RL methods implemented in practice?
2. How do different RL approaches for financial portfolio management compare to one another?
3. What are the effects of hyperparameters and topologies of Neural Networks on performances and returns of trading agents?

We will develop a reinforcement learning agent that can trade within the environment which employs different types of RL approaches to financial portfolio management. In the heart of each of these RL agents lies an artificial neural network (ANN). These ANNs can have various topologies, activation and loss functions, optimizers, or initializers.

1.3 Objectives and Limitations

Since there exist several different methods of RL, one goal of this project is to probe various approaches and compare them to one another. Moreover, we would like to see if the same algorithm with the same configurations can perform equally or comparably well on different resolutions and volumes of data. We wish to also assess the effectiveness of various topologies of underlying ANNs in the RL's agent. Several configurations and hyperparameters will be applied to study their impact on the performance of the algorithms. We will also work on feature engineering and evaluate whether they make a difference or not.

This study will not evaluate approaches implemented in other prior studies, instead, we will build upon existing studies. The outcome of this project is not meant to be used for high-frequency trading, it is rather to examine algorithms, their performances, and their profitability in a controlled manner. Various approaches are evaluated against each other to form a baseline to compare them with more naïve methods. This, hopefully, may pave the way for further investigations of use cases of reinforcement learning not only in trading but also in other applications.

We came across a limitation of data trends due to the COVID-19 pandemic. Value of currencies fluctuated rapidly, and exchange rates were changing constantly. On the other hand, the Euro currency (€) increased in value compared to the other currencies. This problem is explained in more details in subsection 4.2.1 where a method is explained which would generate synthetic data from real data to be used in the study. All hypotheses will remain the same and utilizing the synthesized data will not affect the analyses of the algorithms.

2 MACHINE LEARNING AND PORTFOLIO MANAGEMENT

Using Machine Learning for financial portfolio management is not a new topic, but in recent years there have been a growing interest and investigation in Reinforcement Learning for decision making. The aim of portfolio management is to maximize the expected profit by distributing the funds on various financial products whilst minimizing the risk

(Jiang, et al., 2017). In this project, we are exploring newly developed or theorized technologies in the realm of Machine Learning.

Various methods have been developed to tackle the problem and find optimized solutions. They include manual management and automatic management using computerized technologies. The automatic agents vary from naïve methods that are simply based on some pre-defined rules to more modern automatic agents based on advances in Artificial Intelligence and Machine Learning (Cumming, 2015). There is also a large variety of methods that utilize Machine Learning techniques. These methods may include Linear or Polynomial Regression (Nunno, 2014), Extreme Learning Machines (Li, et al., 2014), Artificial Neural Networks and Deep Learning (Nino, et al., 2017), Recurrent Neural Networks (Ponomarev, et al., 2019), and Reinforcement Learning (Jiang, et al., 2017).

Maximizing the profit of the portfolio while minimizing the risk at the same time and achieving these criteria optimally have proven to be challenging. The financial market fluctuates rapidly, and many factors are involved. A large number, or probably a near-infinite number, of external variables, may influence the market prices in different ways. Fluctuations on inter-dependent markets, natural forces and phenomena, and political events are among the influential external variables. Clearly, taking all these variables into account is an extremely complex task that is nearly impossible to accomplish and is therefore out of the scope of existing research projects (Kanwar, 2019).

The share of computerized trading has continuously increased over the past years to the point that an overwhelming portion of trades is now done electronically. This has given room to "continuous" or "high-frequency" trading to become a widely used trading mechanism in financial markets (Farmer & Skouras, 2012). The recent improvements to computer hardware that are made cheaper and more readily available, as well as the growth of available data on historic prices of markets have encouraged more research in AI-driven price discovery and trading strategies (Dixon, 2017). All these reasons have caused that high-frequency trading strategies to gain a significant advantage over other traders which put even more pressure to develop faster approaches (Farmer & Skouras, 2012).

Typically, most algorithmic approaches to portfolio management are Machine Learning based, especially the more general class of supervised learning algorithms (Dixon, 2017). A supervised learning approach is mainly based on price predictions where at each point

of time the prices of financial instruments (stock, currency, etc.) available in the portfolio are predicted based on the historical prices. This means that the algorithm is dependent on the accuracy of the price prediction. The price prediction can be notoriously difficult due to a large number of influencers, however. Additionally, a decision-making agent should be added to the algorithm that would act based on the predicted prices (Jiang, et al., 2017).

In the following sections, we describe how to develop an agent that experiences an environment and makes decisions. In our environment, the agent learns by examining the individual market's historical prices, trends, and other hidden factors make decisions and receives a reward signal from the environment. Then by utilizing Machine Learning techniques the agent generates a model that can translate the state of the environment (prices and trends) into actions.

2.1 Reinforcement Learning

Reinforcement learning can be described as learning by interacting with the environment and receiving feedback based on the taken actions. It is the nature of learning when we think about how the learning process works. For example, a horse foal learns how to stand on its four legs within a few minutes. It struggles to keep the balance at first and falls several times. The ability to stand on four legs is learned by receiving signals from the environment. Each attempt of the foal to stand upright is usually better than the previous attempt. The sensory information about the environment is used to fine-tune and learn which actions lead to success and which actions fail.

In other words, Reinforcement learning is like making decisions based on previous experience. Interacting with the environment helps to gain more experience. At each state, a decision is made, an action is taken, and a feedback is received. The feedback can be a reward or a punishment. Later, when the same or a very similar state is observed, the gained experience from the past can be utilized to make a more informed decision, to maximize reward and minimize the punishment.

The learner must explore and discover what actions can be taken given the current situation. Mapping the current situation into action in order to maximize the reward is a fundamental part of reinforcement learning (Sutton & Barto, 2018).

2.1.1 Elements of Reinforcement Learning

Here we define the basic elements of reinforcement learning. Per Figure 2, the *agent* interacts with the *environment*. The interaction between the *agent* and the *environment* takes place discretely, one step at a time. The *agent* takes an *action* and in return, it receives a set of new *states* and a *reward*. The *reward* is a single number that the agent receives on each step from the environment. An *episode* is a period from the initial state until a terminal state is reached.

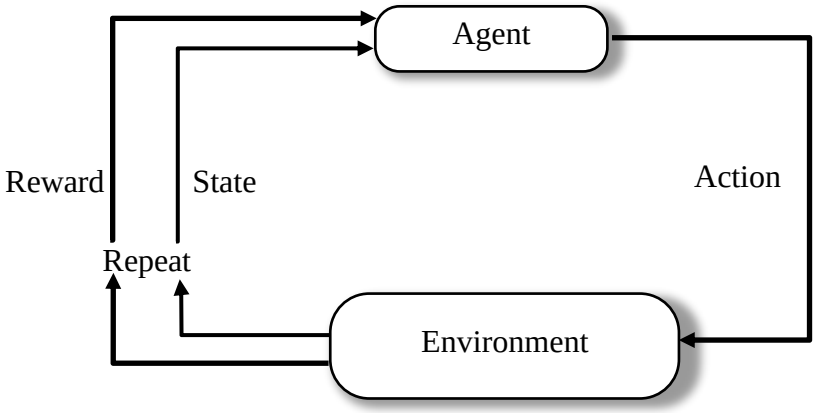


Figure 2 Interaction between the agent and the environment

The "Reinforcement Learning: An Introduction" book by Sutton and Barto (Sutton & Barto, 2018), in addition to *states*, *environment*, *reward*, and *agent*; defines a *policy*, a *value function*, and possibly a *model* can be defined as sub-elements of a reinforcement learning system.

Value Function. A value function specifies the accumulated rewards in the future. It can be assigned to a *state*, to an *action*, or to a (*state*, *action*) pair (Cumming, 2015). In contrast to a reward, which is an immediate outcome of taking an action, a value function indicates the expected value of rewards in a long run (Sutton & Barto, 2018). It can be interpreted as the estimation of how good it is for the agent to be in a given state or how good it is to perform a given action in a given state.

Policy. A policy is mapping of the current states to an action. It is a set of rules that the agent uses to decide what actions to take given a set of states (Cumming, 2015). A *policy* is a strategy of performing actions given states. Reinforcement learning algorithms attempt to find the optimal policy by experiencing states and actions. Such algorithms

observe the set of possible states and attempt to maximize the reward by taking the best possible action (Xiong, et al., 2018).

Model. A model can be described as a simulation or a representation of an environment that can allow the behavior of the environment to be inferred. For instance, a model might take a pair of (*state, action*) and predict the reward and the next state (Sutton & Barto, 2018).

2.2 Reinforcement Learning Methods

At the beginning of this section, we cover fundamental concepts in RL and discuss exploration and exploitation. Then, we will explore the tabular methods and give a brief introduction to them. These methods are useful when the state and action space sizes are sufficiently small, i.e., fit in memory, so that the value function approximation can be represented as *tables*, a tabular method can be utilized. Often, an optimal value function and an optimal policy can be found.

In most real-life scenarios, especially for algorithmic trading, however, the state size is either continuous or too large to fit in memory. In the case of algorithmic trading, the states are prices or derivatives of them. Therefore, an approximation of the value function should be used. Approximation methods are introduced later in this section.

The following subsections will lay out the foundation of RL and how the methods evolve to tackle small-scale rudimentary problems to more sophisticated methods that deal with continuous spaces with the long-term goal in mind. Later in this study, the role of Neural Networks is introduced, notably, we establish in what ways they can be incorporated in solving RL problems, and how we use them in the experiments.

2.2.1 Exploration vs. Exploitation

The *Explore/Exploit dilemma* is often brought up in the *Multi-armed bandit* problem in the existing literature (Espinosa-Leal, et al., 2020). The training process of reinforcement learning requires exploration of the action space to collect data and to update the reward. As we iterate and collect data to learn about the environment, we gain experience about actions and their expected rewards. This may result in choosing the next actions based on

our experience from the past which means exploiting our knowledge. However, our knowledge of the environment is not complete, especially when it comes to a highly dynamic environment like stock trading. Therefore, we need to keep exploring during the training process for as long as possible.

To combat the Explore/Exploitation dilemma several methods can be utilized. A few of the most common methods may include (Sutton & Barto, 2018):

- ε-greedy and its variations,
- Upper Confidence Bound,
- Gradient Bandit Algorithms,
- Thompson Sampling.

Only ε-greedy and its variations are utilized in the learning process.

ε-greedy. ε-greedy is a simple approach to choose between exploration and exploitation. In each iteration of the learning process at time t, the algorithm takes a random action with the probability of ε, $0 \leq \varepsilon \leq 1$. This means, the probability of exploration, taking a random action a from the action space, is $P(a) = \varepsilon$ and the probability of exploitation is $1 - \varepsilon$.

Two extensions of ε-greedy are Decaying Epsilon, and Optimistic Initial Values.

Decaying Epsilon is a variation of ε-greedy where ε is not constant. Instead, the value decreases as we proceed with the learning. The only difference here is that in each iteration at time t, we take a random action a with probability of $f(t)$ which is a decaying function. For instance, a popular choice is $f(t) = \frac{1}{t}$.

Optimistic Initial Values method only contains the greedy part but not the epsilon part of the regular ε-greedy. The mean rewards of each action in this method are optimistically initialized to encourage exploration. In each step the action with the highest mean reward is selected, i.e., $A_t = \arg\max Q_t(a)$. The reward is then calculated and updates the mean reward. The means will eventually converge to the true values of the rewards (Sutton & Barto, 2018).

2.2.2 Rewards, Returns, and Episodes

We have already introduced the concept of rewards in an RL setting. A "reward" is the signal that the environment returns to the agent after taking an action. The goal of the agent is to maximize the expected cumulative value of the rewards in the long run (Cumming, 2015). The cumulative sum of rewards is called "Return", denoted as G_t, at time t (Sutton & Barto, 2018):

$$G_t \doteq R_{t+1} + R_{t+2} + \ldots + R_T$$

where R_t is reward at time t and T is the final time step, also referred to as the *terminal state*.

An "episode" is an agent's interaction (Szepesvari, 2009) or a sequence of actions (Cumming, 2015) with the environment that always starts from the initial state and ends in the terminal state.

The transition from state to next state during an episode is depicted in Figure 3.

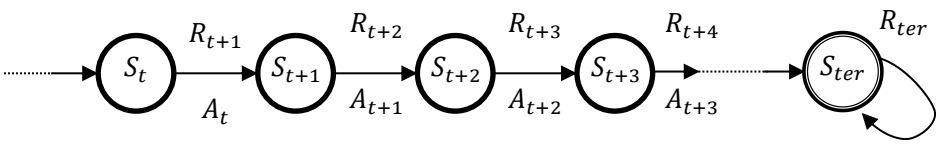


Figure 3. Episode's transitions

2.2.3 Markov Decision Process

Markov Decision Problems (MDPs) can be traced back all the way to the fifties to the work of Richard Bellman in stochastic control theory (Bellman, 1957). They are sequential decision-making processes where an *action* has to be chosen in each state by the system (Kanwar, 2019).

Sutton and Barto (2018) (see also Kanwar (2019)) define an MDP as a 5-tuple, $\langle S, A, P, R, \rho_0 \rangle$, where:

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

- $P: S \times A \to \mathcal{P}(S)$ is the transition probability function, which assigns a probability measure to each state-action pair $(s, a) \in S \times A$ that the next state, s', is valid and $s' \in S$.
- $R: S \times A \times S \to \mathbb{R}$ is the reward function, with $r_t = R(s_t, a_t, s_{t+1})$,
- and ρ_0 is the starting state distribution.

Figure 4 shows a 3-state MDP with (a, p, r) on each arc where a is the action taken in state s, with probability p, that gives the reward r (Gosavi, 2009). For instance, in Figure 4 there is a 20% probability that action 2 would be taken on state "Snowy" which would end up in state "Sunny" and would yield 5 as the reward.

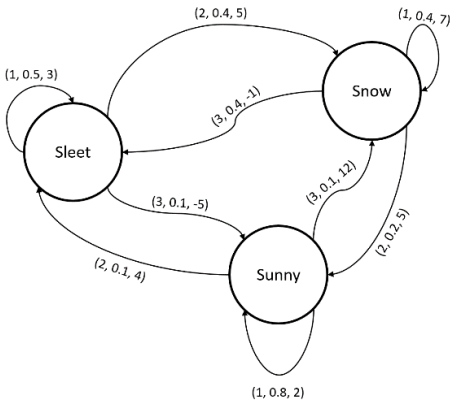


Figure 4. A 3-state MDP with (action, probability, reward)

In an MDP, the state and the reward at step $t + 1$ only depend on the state at step t and the action taken then and the previously observed states, taken actions and received rewards have no effect on what the state and the resulting reward would be in the following step. An MDP on a finite action space and a finite state space should satisfy the following two definitions (Cumming, 2015):

Transition Probabilities: the transition probabilities ($\mathcal{P}_{ss'}^{a}$) of a finite MDP give the probability of transitioning from the initial state s_t with action a_t to the subsequent state s_{t+1} (Cumming, 2015):

$$\mathcal{P}_{ss'}^{a} = \mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a)$$

Expected Reward. Expected reward for state s_t, action a_t, subsequent state s_{t+1} is the expectation of the reward function given the triplet of (a_t, s_t, s_{t+1}) (Cumming, 2015):

$$\mathcal{R}^{a}_{ss'} = E(r_{t+1}|s_{t+1} = s', s_t = s, a_t = a)$$

2.2.4 Policy and Value Function

Sutton and Barto in the "Reinforcement Learning: An Introduction" book (Sutton & Barto, 2018) define Value Functions as "functions of states (or of state-action pairs)" which estimates how much is a state worth (or how much is a state worth given an action). In other words, the expected reward is going to be given a state (or an state-action pair).

A policy is the strategy of performing actions given states. In other words, it is a mapping from states to probabilities of selecting each possible action as illustrated in Figure 4. Reinforcement learning algorithms attempt to find the optimal policy by experiencing states and actions. Such algorithms observe the set of possible states and attempt to maximize the reward by taking the best possible action (Gosavi, 2009). A policy is what we ultimately want to "learn" and then apply this learning to solve similar problems.

$v_\pi(s)$, referred to as *value function* of a state *s*, is the expected reward when starting in state s and following policy π. Generally, the function v_π is referred to as *state-value function for policy* π. For a terminal state, also called the final state, this value is zero. Correspondingly, the *action-value function* $q_\pi(s, a)$, is defined as the value of taking action a in a state s under policy π (Sutton & Barto, 2018).

2.2.5 Dynamic Programing

Dynamic Programing (DP) refers to a series of algorithms and methods to derive optimal policy for an environment using MDP (Sutton & Barto, 2018). DP assumes it can gain knowledge of the entire environment and can compute an exact, deterministic solution (Jansen, 2020). This assumption implies that the environment, its state, action, and reward sets are finite (Sutton & Barto, 2018).

The Bellman equation (Bellman, 1957) is used to recursively update the value functions of each state by starting at the terminal state. Values of states are calculated based on the expectation of the reward and the estimation of the subsequent state's value. When the value function is close to the optimal one, the policy can be improved by making it greedy

with respect to the value function (Cumming, 2015). Then the state (or state-action) value functions are re-evaluated until the policy converges closer to the optimal policy.

A major drawback of a DP is that the entire environment must be known and be finite, so it is not performant when it comes to tackling environments with large or continuous actin space.

2.2.6 Monte Carlo

Unlike DP, Monte Carlo (MC) method does not require complete knowledge of the environment; instead, only *experiencing* sequences of states, actions, and rewards are required (Sutton & Barto, 2018). MC does not need to know the complete probability distribution of all states, actions, and rewards as stated in Figure 4. This method learns the rewards of different actions solely by sampling the sequences of state-action-reward (Jansen, 2020).

Monte Carlo methods estimate the state value function v_π and state-action value function q_π for state s by following a policy π and averaging the returns of the subsequent states. This average will ultimately converge to the state's value, denoted as $v_\pi(s)$. Similarly, state-action value function $q_\pi(s, a)$ is estimated by taking averages for each action taken in each of the states (Sutton & Barto, 2018).

Monte Carlo methods are relevant for episodic tasks. Once a sample run (episode) is completed and a terminal state is reached, we trace back, calculate the returns for each state, and update the value functions until converged (Cumming, 2015).

2.2.7 On-Policy and Off-Policy Methods

There are two approaches to ensure that an agent visits all the states in an environment often enough and continues to select them which are called *on-policy* and *off-policy* methods. An on-policy method attempts to update the policy based on the data that lead to taking the latest action; whereas an off-policy method makes the update to the policy using the data obtained at any point (Kanwar, 2019).

On-policy methods are in general soft with respect to policy improvement. For example, when they follow an ε-greedy policy, the probability of an action being chosen at random

is ε. However, an on-policy method continually moves towards a more deterministic optimal policy (Sutton & Barto, 2018).

Off-policy methods are in general harder than the on-policy counterparts to implement, because off-policy methods usually need further concepts, models, and data. Besides, they are slower to converge and may have greater variance. Off-policy methods have a *target policy*, which is the one that is being learned, and a *behavior policy*, which is the policy that generates behavior. The target policy is what becomes the optimal policy, and the behavior policy explores the environment to generate the behavior (Sutton & Barto, 2018). This makes them more general and usually more effective. Also, on-policy methods become a special case of off-policy methods when both the behavior and the target policies are the same.

2.2.8 Temporal-Difference (TD) and *SARSA*

Temporal-Difference (TD) learning can be described as a combination of merits of Monte Carlo and Dynamic Programming. TD utilizes bootstrapping like Monte Carlo methods to learn directly from interacting with the environment and gaining experience. And, similar to DP methods, TD uses the learned experiences to update estimates of the value functions (Cumming, 2015), (Sutton & Barto, 2018).

Monte Carlo methods gain experience during an entire episode and only after reaching the terminal state, do they evaluate the increment of the value function, $V(S_t)$. TD methods have an advantage over MC methods as they can update the $V(S_t)$ after each step of the episode. At time $t+1$ upon observing reward R_{t+1} and the value $V(S_{t+1})$ estimate that receives the transition to S_{t+1}, the following update is made (Sutton & Barto, 2018):

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

where α is step-size constant and γ is the learning rate. This particular case updates the value function at step *t* after observing step *t+1*, therefore, is called *one-step* TD or TD(0). This method can be developed to *n-step* or *TD(λ)* (Sutton & Barto, 2018). The procedure is outlined in Procedure 1.

SARSA. algorithm uses the current State, current Action, next Reward, next State, and next Action $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. It is an on-policy extension to *TD(λ)* algorithms that, rather than learning the state-value function, it learns the action-value function. In

TD and generally for Monte Carlo methods, values of states are learnt, whereas, here values of state-action pairs are considered. Like TD algorithm, the action-value function for each non-terminal state is estimated as: (Sutton & Barto, 2018)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \quad (2)$$

and $Q(S_t, A_t) = 0$ for a terminal state. Procedure 2 outlines this approach.

Tabular $TD(0)$ for estimating v_π
Input: the policy π to be evaluated Algorithm parameter: step size $\alpha \in (0,1]$ Initialize $V(s)$: arbitrarily for all $s \in S$, for $V(S_{terminal}) = 0$ For each episode: Initialize S For each step: $a \leftarrow$ action given by π for s $R, s' \leftarrow$ take action a $V(s) \leftarrow V(s) + \alpha[R + \gamma V(s') + V(s)]$ $s \leftarrow s'$ until s is terminal

Procedure 1. Tabular TD(0) (Sutton & Barto, 2018)

SARSA (on-policy TD control)
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$ Initialize $Q(s, a)$: arbitrarily for all $s \in S, a \in A$ $Q(S_{terminal}, .) = 0$ For each episode: Initialize S If $random < \varepsilon$: $a \leftarrow$ Randomly from A Else: $a \leftarrow$ from s using policy derived from $Q(s, a)$ For each step: $R, s' \leftarrow$ take action a If $random < \varepsilon$: $a \leftarrow$ Randomly from A Else: $a \leftarrow$ from s using policy derived from $Q(s, a)$ $Q(s, a) \leftarrow Q(s, a) + \alpha[R + \gamma Q(s', a') - Q(s, a)]$

$s \leftarrow s'; a \leftarrow a'$
$\varepsilon \leftarrow$ update ε e.g. ε-greedy
until s is terminal

Procedure 2. on-policy SARSA (Sutton & Barto, 2018)

2.2.9 Q-Learning algorithm

An action-value function under policy π is denoted as $Q_\pi(s, a)$. The result is the expected value at state s by taking action a under the policy π (Xiong, et al., 2018).

Q-learning is an off-policy method that updates at any point during the training process without considering the previous states and the exploration method. For each state-action pair $(s, a) \in S \times A$ the *Q*-learning algorithm stores an estimate $Q_t(s, a)$ of the optimal policy $Q^*(s, a)$ (Szepesvari, 2009). $Q_t(s, a)$ is the value function at step (time) t of action a with state s. Given a discount factor γ and observing the transition from (S_t, A_t) to state $s' = S_{t+1}$ with reward R_{t+1}, $Q(S_t, A_t)$ can be estimated using the following equation (Sutton & Barto, 2018):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, A_t) - Q(S_t, A_t)\right] \quad (3)$$

Q-learning procedure

$\alpha \in (0,1], \varepsilon > 0$
Arbitrarily initialize $Q(S, A), s \in S, a \in A$
For each episode:
$s \leftarrow$Initial state
For each step until s is terminal:
If ε (e.g. ε-greedy) then:
$a \leftarrow$ Random action $\in A$
Else:
$a \leftarrow \underset{s \in S}{argmax}\, Q(s, A)$
$s', R \leftarrow Act(a)$
$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(s', a) - Q(S, A)\right]$
$s \leftarrow s'$

Procedure 3. Q-learning pseudocode (Sutton & Barto, 2018)

Q-learning is a case of TD learning which allows the use of sampling arbitrarily during the training process (Watkins, 1989). That is given all state-action pairs are visited and

updated often (Szepesvari, 2009). The goal of *Q*-learning is to find an action that would maximize the return given the set of current states (Procedure 3).

2.2.10 Actor-Critic methods

Actor-only methods are in **policy iteration** family where the policy function is improved in each step and each episode. On the other hand, a critic-only method estimates and optimizes the **value function** which tries to learn the Bellman equation. The combination of the strengths of these two methods is the **actor-critic** algorithm, where the critic part approximates a value function using simulation and approximation, which in turn is used to update and improve the policy (Konda & Tsitsiklis, 2003). The interaction between the actor and critic of a reinforcement learning environment is depicted in Figure 5.

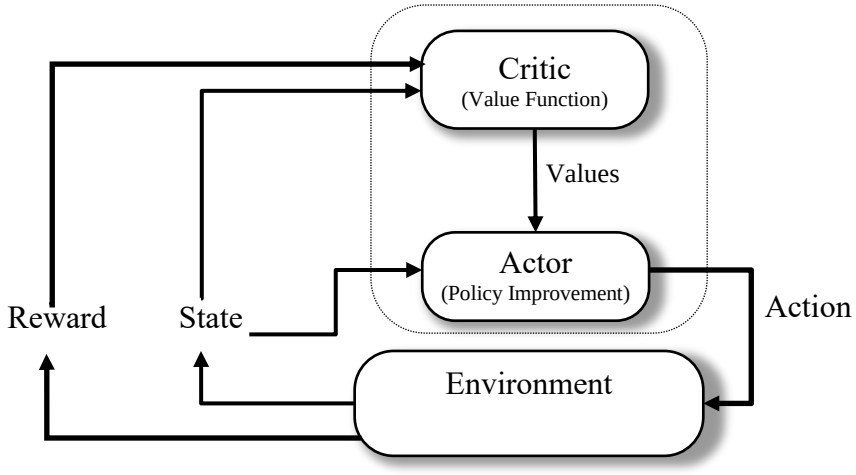


Figure 5. An Actor-Critic environment

Critic is the value function estimator of the *actor*'s target policy. It is a modified value function estimator that instead of evaluating an action's value, it directly estimates the value of actions (Szepesvari, 2009). The critic is essentially a form of SARSA algorithm where the estimated trajectory is used by the actor to update the estimate of the policy.

Szepesvari (Szepesvari, 2009) proposes two approaches to implement the policy improvement of the actor. First approach utilizes the critic's output to push the current policy to the estimated action-value function; and the second approach implements a gradient ascent on the parameters of the policy's performance.

Procedure 4 shows an actor-critic process where the state-value function, $\hat{v}$, is paired with semi-gradient *TD(0)*, α is the step size, π is the policy, ∇ is the partial derivatives, θ is the target policy vector, δ is the temporal difference, γ is the discount rate, w is the weights vector of the approximate value function, and I is the return.

Episodic Actor-Critic

Input: a policy parameterization $\pi(a|s,\theta)$
Input: a state-value function parameterization $\hat{v}(s,w)$
Parameters: step sizes $\alpha^{\theta} > 0, \alpha^{w} > 0$
Initialize policy parameter $\theta \in \mathbb{R}^{d}$ and state-value weights $w \in \mathbb{R}^{d}$ (e.g., to 0)
For each episode:
    Initialize s (first state of episode, i.e. *env.reset()*)
    $I \leftarrow 1$
    For each time step (until S is terminal):
        $a \leftarrow \pi(.|s,\theta)$
        $s', r \leftarrow$Take action a
        If s' is not terminal:
            $\delta \leftarrow r + \gamma\hat{v}(s',w) - \hat{v}(s,w)$
        Else:
            $\hat{v}(s,w) = 0$
        $w \leftarrow w + \alpha^{w}\delta\nabla\hat{v}(s,w)$
        $\theta \leftarrow \theta + \alpha^{\theta} I\delta\nabla \ln\pi(a|s,\theta)$
        $I \leftarrow \gamma I$
        $s \leftarrow s'$

Procedure 4. Episodic one-step actor-critic procedure (Sutton & Barto, 2018)

2.3 Financial Terms and Concepts

Before getting into the technicalities and the implementation, a few relevant, commonly used financial terms and concepts are described as follows.

Asset. An asset is any item of economic value such as cash, stock, share, bond, etc. In this project, the term refers to stock or cash.

Stock. A stock is fractional ownership of a corporation's assets that can be purchased by members of the public. It is also known as *share* or *equity* which entitles an individual who owns the shares to ownership of that proportion of the issuing corporation's earnings and assets.

Stockholder. A stockholder is a member of the public who owns one or several stocks (shares or equities) of an issuing corporation. A stockholder could be an individual or an organizational entity.

Stock transaction. The act of trading stocks, or in other words buying and selling, is a stock transaction. This action usually takes place in a stock exchange market, though it could also occur privately.

Portfolio. A portfolio consists of a group of various financial assets such as stock, equities, securities, currencies, bonds, real estate, or anything of a financial value.

A portfolio can be represented in a vector form, where each element refers to a stock or an asset such as:

$$V = p_1 \times h_1 + p_2 \times h_2 + \cdots p_n \times h_n$$

where, for $1 \leq i \leq n$, $p_i \in \mathbb{Z}^+$ represents the count of stocks and $h_i \in \mathbb{R}^+$represents the value of each stock.

Stock price resolution. The price of a stock is a volatile value that continuously fluctuates; hence, the precise evaluation at any given time is difficult to obtain. The trading price per stock at the time of submitting a buy or sell request may differ from the actual traded price. Therefore, oftentimes algorithmic trading four values of *open, low, high,* and *close* at any given time interval are used to obtain a better picture. The *interval* can be as small as one minute (or even smaller) or as large as one day. An example of price variation within a time interval is illustrated in Figure 6.

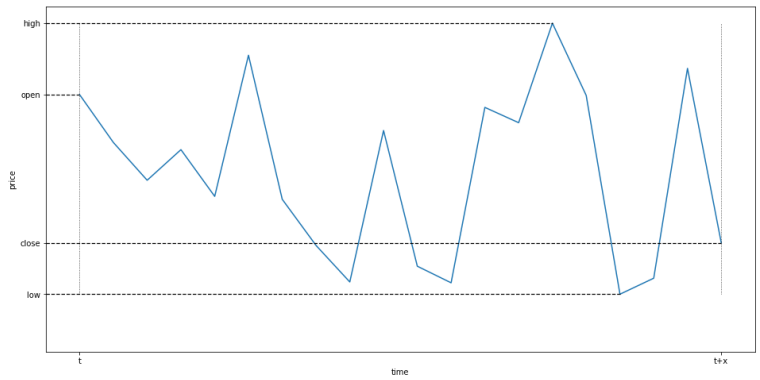


Figure 6. Stock prices in at time t with interval of length x

3 RELATED WORK

Reinforcement Learning has proven to be a valuable addition to the toolbox. The agent explores the historical data and makes decisions as it moves forward. Each decision is then evaluated, and a reward is calculated and fed back to the agent. The agent interacts with the environment, that is the market's price data, and continuously receives reward/penalty as feedback and adjusts the behavior for the next decisions accordingly. Through the constant interaction and feedback loop, the agent "learns" to adjust its judgment optimally to the environment.

This study aims to explore the area of Reinforcement Learning by investigating a good setup for the environment, learning agent, reward function, and the feedback loop. Several research projects have been done in this area using solely Machine Learning methods, Reinforcement Learning, or a mix of various techniques. Some examples include "A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem" by Zhengyao Jiang, et al (2017), "An Investigation into the Use of Reinforcement Learning Techniques within the Algorithmic Trading Domain" by James Cumming (2015), or "An automated FX trading system using adaptive reinforcement learning" by M.A.H. Dempster and V. Leemans (2006).

In this chapter, two of the main inspirations to this research are covered that are both based on variants of Neural Networks. In the first part, algorithms based on Convolutional Neural Networks, Recurring Neural Networks, and Long-Term-Short Memory are introduced. State and action spaces as well as the CNN algorithm from the work of (Jiang, et al., 2017), motivated the implementations in this study.

The second part explores several policy-based methods. The main takeaway from (Laing, et al., 2018) for this study is the actor-critic algorithm and how it approaches a stock trading environment.

3.1 Deep Reinforcement Learning with CNN, RNN, and LSTM

In the study "A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem" (Jiang, et al., 2017), three instances of a Convolutional Neural Network (CNN), a Recurrent Neural Network (RNN), and a Long-Short-Term Memory

are implemented. In this research, the trading was done on a cryptocurrency market, with Bitcoin being the most well-known example.

In their study, Jiang and colleagues (2017) preselected 11 cryptocurrencies, including Bitcoin. This makes the size of the portfolio $m + 1 = 12$. Each asset in the portfolio (referred to as "coin"), like other financial instruments, has 4 prices in each tick: open, low, high, and close; each of which is for a 30-minute period. All of these are fed to the algorithm to train and make decisions. For the most parts, the environment, the actions, the reward function, and the explore-exploit are similar to ours as it will be presented in chapter 4. Method.

3.1.1 Network Topologies of Policy Function

Figure 7 shows the CNN implementation and Figure 8 displays the topology of an RNN or LSTM network. The input to the network in all cases is the price vector and the output (as displayed in the figures) is the portfolio vector ω_t which contains the asset allocation of our portfolio at time *t*.

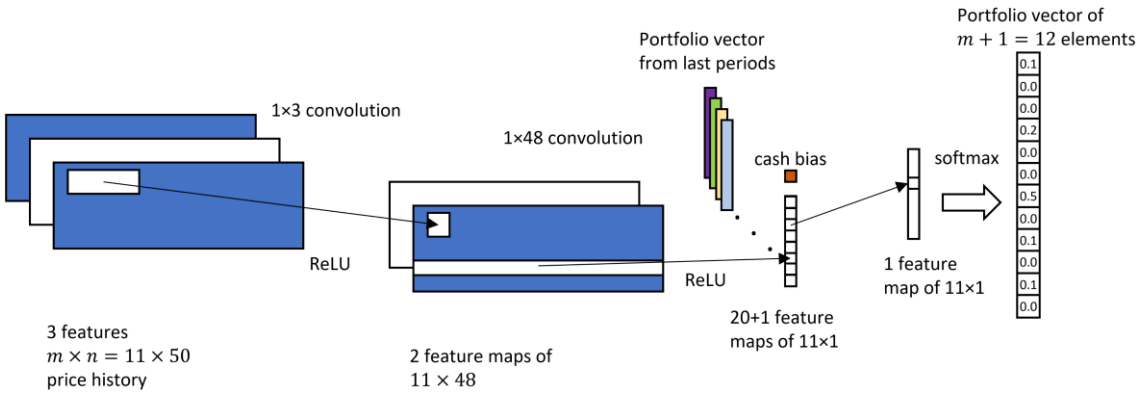


Figure 7. CNN Implementation

An important feature of this study's implementation is that for each of the *m* assets in the portfolio, networks are identical, but flow independently. They only connect at the last point where *Softmax* function is applied. These streams are formally called Identical Independent Evaluators (IIE). The Ensemble of IIE (EIIE) greatly improves the performance. The authors also describe three factors as the advantages of such EIIE approach:

1. Scalability of asset number: training time is roughly linear due to the ensemble nature of the algorithm.

2. Data-usage efficiency: for an interval of price history an IIE can be trained *m* times across assets which shares the accumulated experience in both time and asset dimensions.
3. Asset collection plasticity: because an IIE is not restricted to an asset, the size of the portfolio can change on the go without having to re-train the entire network.

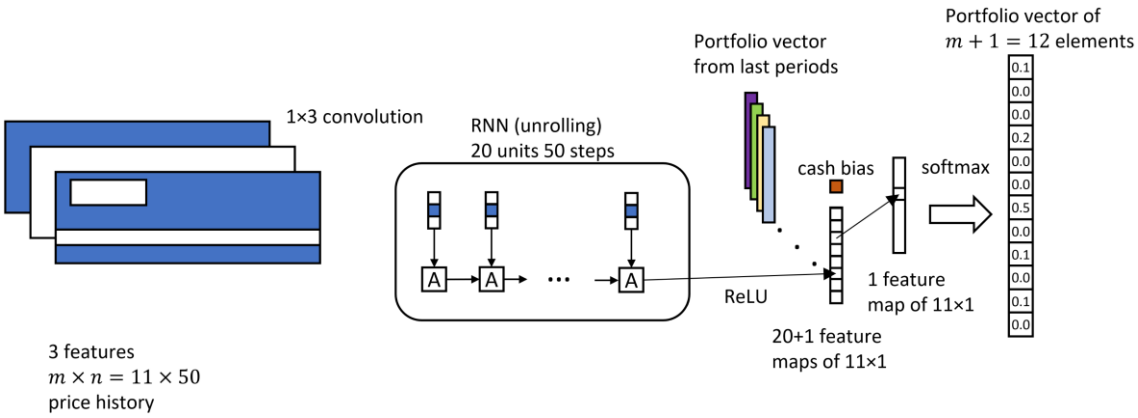


Figure 8. Basic RNN or LSTM Implementation

3.2 Adversarial Deep Reinforcement Learning − Actor-Critic

In the paper "Adversarial Deep Reinforcement Learning in Portfolio Management" (Laing, et al., 2018), Deep Deterministic Policy Gradient (DDPG), Proximal Policy Optimization (PPO), and Adversarial Policy Gradient (PG) approaches are studied on datasets of Chinese stock market.

Deep Deterministic Policy Gradient (DDPG) is a combination of Q-learning and Policy Gradient. The DDPG approach consists of an actor to output actions and a critic to evaluate and improve the Policy. The actor function is the Policy Gradient function and the critic is the Q-Value function. Four networks are instantiated in this method: online actor, target actor, online critic, and target critic. Online actor outputs an optimal action, online critic evaluates the action based on which the online actor is updated. Then the online actor and the online critic, update the target actor and target critic softly as described in subsection 4.3.4 below.

Proximal Policy Optimization (PPO) is based on Trust Region Policy Optimization (TRPO) (Schulman, et al., 2017). TRPO is a theoretically justified algorithm to improve

the Policy that is obtained by minimizing a certain surrogate objective function (Schulman, et al., 2015). PPO still attains the data efficiency as well as the performance of TRPO, but only uses the first-order optimization. An objective function with clipped probability ratios is introduced to form a lower bound of the performance of the policy. Then the policy is optimized by alternating between sampling data and performing several epochs of policy optimization based on the sampled data (Schulman, et al., 2015).

Adversarial PG is used in the paper due to the unsatisfactory performance of the two other methods. To increase the robustness of the actor-critic algorithm is devised to estimate the gradient and update the policy parameters in the ascent direction. This is performed while the convergence of the implemented algorithms is established to locally risk-sensitive optimal policies (Laing, et al., 2018). An adversarial training by adding $N(0, 0.002)$ noise in the data is utilized. Their network structure is, otherwise, similar to those of Jiang et al (2017).

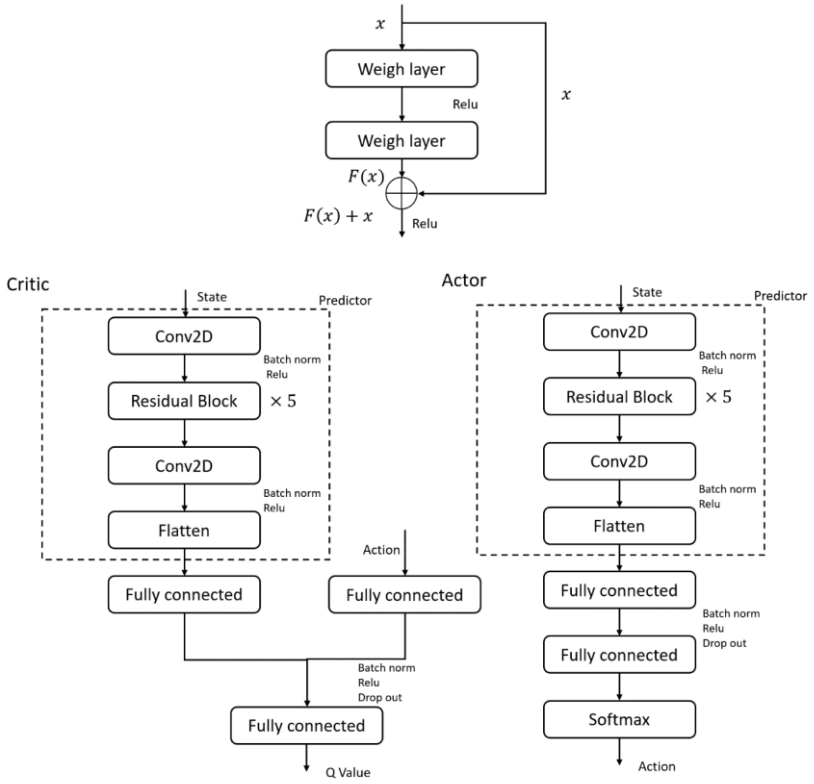


Figure 9. Residual block and the DDPG network structure of the experiment

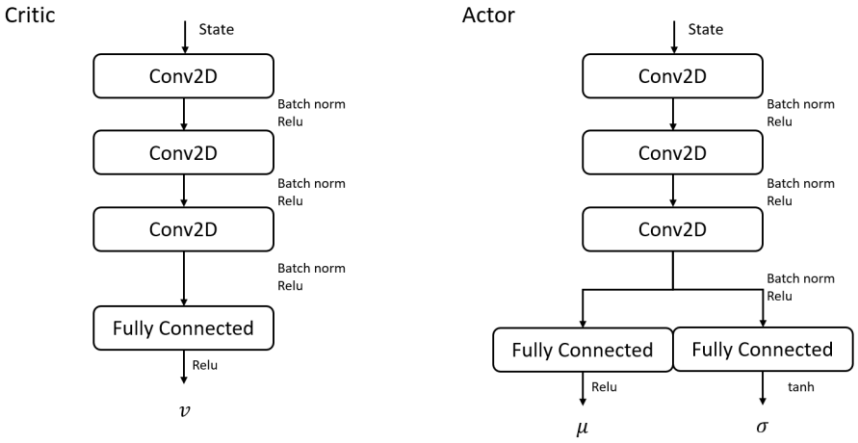


Figure 10. PPO Network Structure in the experiments

4 METHOD

In this chapter, the main components of *states*, *actions*, *environment*, *policy*, *reward*, and *value function* in reinforcement learning are described. Some of the components are quite restricted by the environment while some other ones are left free. For example, there is very little freedom in defining the action space, whereas the design and implementation of the policy's model are entirely up to us.

Another topic to cover is the available options for the decision-making process and handling the explore exploit dilemma. We introduce the approach taken in this study.

The data, and the format, are of high importance as they have a key role in defining the general environment as well as the state space.

4.1 Tools

A large part of this study consists of writing code for training, interacting with the environment, and evaluating results. All the code has been written in Python[1] programing language using a variety of libraries. The PC used for code execution runs on Intel® Core™ i7-8750H, 8th generation, with 6 cores (equals to 12 virtual cores with hyperthreading) at the clock speed of 2.2-4.1GHz that utilizes 16GB of DDR4 RAM[2] at 2666MHz with dual-channel capabilities. The storage in the PC is 256GB SSD M.2 PCIe. Optionally, it is

[1] https://www.python.org/
[2] Random Access Memory

possible to utilize an NVIDIA® GeForce™ GTX 1060 GPU with 6GB of DDR5 RAM. Using the GPU is advantageous when the ANN size is large (network with several large hidden layers) or the number of input features (states) is sizable.

4.1.1 Pandas

Pandas[1] is an open-source library in Python programing language for data analysis. It is powerful, high-performance, and user-friendly that makes structuring, analyzing, and manipulating data easy for a programmer or an analyst (Pandas, n.d.). In this study, it is used for data preparation and data handling tasks. The source files, where the price histories are located, are read using Pandas, and data from different assets can be merged into one data structure which makes the handling job simpler. Additionally, all feature engineering tasks as well as handling missing data are all accomplished using this library.

4.1.2 NumPy

NumPy[2] is a Python open-source library for numerical computation. It is highly optimized and makes a variety of scientific routines accessible to a programmer. In this project, a suite of features for multidimensional array objects of NumPy is used (Numpy, n.d.). Random choices, matrix and vector operations, and some statistical operations are some of the use cases of this powerful library in our project.

4.1.3 Keras

Keras[3] is an open-source platform in Python that provides an interface for the TensorFlow[4] library and simplifies the implementation of artificial neural networks. It provides APIs[5] that reduce the amount of work required by the programmer for common use cases. At the same time, the Keras library is flexible and adjustable, yet remains efficient due to its deep integration with low-level TensorFlow (Keras, n.d.).

[1] https://pandas.pydata.org/
[2] https://numpy.org/
[3] https://keras.io/
[4] https://tensorflow.org/
[5] Application Programing Interface

In this project, all the neural networks, including convolutional neural networks, are implemented using the Keras framework. Because of its flexibility, various topologies of networks can simply be passed to a network building function as a variable. Besides, hyperparameter tuning becomes simpler and the implementation is mostly reduced to providing the network building function, subsequently the Keras layers, with specified hyperparameters.

4.2 Setting the Elements of Reinforcement Learning

4.2.1 Data

We consider *n* stock options in our portfolio. Each of the stocks is in a similar time series as other ones. They all contain at least Open, Close, High, and Low prices in each timeframe. Optionally historical information of the traded volume could bring in more information and therefore could be used in our algorithms.

The datasets are acquired from public sources as provided by TickStory (Tickstory, n.d.). These datasets contain the historical stock trading prices of some intervals. The intervals may be as small as minutes, or as large as a day. There are 4 figures in each interval that are of most importance: *open*, *high*, *low*, and *close* prices. The datasets may also contain the trade *volume* of each time interval. The datasets are provided as time series in chronological order where trading of the stock was open. For instance, trading is closed during weekends and bank holidays therefore no entry is usually provided for those periods.

Frequency or Resolution

As stated previously, technological advances have resulted in trades happening at a high rate that is near real-time. The frequency is so high that one may presume the time-series as almost continuous. Time can be broken down into discrete intervals where all of them are of the same length. The length of time can for example be in minutes, quarter-hourly, hourly, bi-hourly, daily, or weekly. The value of the very first trade is “Open” and the last one’s value is “Close”. The lowest trading value within the interval is “Low” and the highest one’s value is “High”. “Volume” refers to the number of the traded financial instrument within the time interval.

Resolution or Frequency refers to the time interval where the snapshot of trades and the indicators are taken. In this study, we mainly focus on the daily resolution of data.

Financial Instruments in the Experiment

Euro (€) is our reference currency where other currencies are valued against it. We have chosen the following currencies in our experiment (see Table 1):

Currency	Abbreviation	Symbol
United States Dollar	USD	US$, $
Canadian Dollar	CAD	CA$
Swiss Franc	CHF	
Pound Sterling	GBP	£
Japanese Yen	JPY	¥

Table 1. Currencies used in the experiment

Adjusting and Synthesizing Data

The main goal of this project is to implement and model an agent in a reinforcement learning environment where the modeled agent could experiment and maximize the gain independent of the origin of the input data. As long as the dataset resembles a stock trading environment, the agent must be able to converge to an optimal (or a sub-optimal) policy and perform trading actions.

A challenge we faced during the data preparation and analysis phase was the instability of exchange rates during the COVID-19 pandemic. This global challenge and the fact that different countries got involved at different times wreaked havoc on the currency exchange rates. This issue presented a challenge and an opportunity. Furthermore, the Euro currency (€) remained one of the only mainstream currencies whose value has appreciated[1] against other currencies. Here, modeling would be challenging due to the increased

[1] Value increase of a currency compared to another currency. When one unit of currency A is worth more units of currency B compared to a previous point in time, currency A has appreciated relative to currency B (Nasdaq, n.d.)

volatility, but at the same time, the method will be more promising if we build a model that could still return a profit in spite of challenges.

To alleviate the challenges, we take the exchange rates datasets for the aforementioned currencies where each presented as matrix $D = [O, L, H, C, V]$ where $O = [o_1, o_2, \dots, o_n]$, $L = [l_1, l_2, \dots, l_n], H = [h_1, h_2, \dots, h_n], C = [c_1, c_2, \dots, c_n],\ V = [v_1, v_2, \dots, v_n]$ are open, low, high, close, and volume vectors. To obtain the adjusted matrix $D' = [O', L', H', C', V']$ the following steps are performed:

1. $o'_i = o_i^{-1}, l'_i = h_i^{-1}, h'_i = l_i^{-1}, c'_i = o_i^{-1}, v'_i = v_i, 1 \leq i \leq n$. Low and High prices are swapped here due to the inversion.
2. $D' = [O', L', H', C', V']$ are scaled using the Standard Scaler, also known as Z-Score Normalization, which uses the arithmetic mean (μ) and the standard deviation (σ) of the provided dataset (Jain, et al., 2005). The scaled value of x is calculated as $x' = \frac{x-\mu}{\sigma}$.

This dataset generally has a somewhat more positive trend in comparison to using the alternative target currency to Euro exchange rates. Otherwise, the agent is more likely to converge to a strategy, where holding the cash is preferred to performing trades. On the other hand, using this semi-synthesized dataset will not influence the core aim of modeling agents that perform trades and return a profit in a negative way.

4.2.2 States

The main indicator of stock *X* is the *closing* price at time *t*. The basic components of our states are the open, close, high, low, and volume information. We refer these as basic fields. However, there are some additional indicators that can bring more value and reveal more insights from the historical prices. In the following subsection, we discuss some of these technical indicators.

For the n stocks in our trade portfolio, **state** is defined as $S = [p, i, h, c]$ where $p \in \mathbb{R}_+^{5\times n}$ is the stock prices, $i \in \mathbb{R}_+^{m\times n}$ is the m calculated technical indicators, $h \in \mathbb{Z}_+^n$ is the holding of each stock, and $c \in \mathbb{R}_+$ is the amount of available cash.

Technical Indicators

Technical Indicators also known as charting, are mathematical formulae to reflect hidden information based on price history (Qiu & Song, 2016). These indicators may reveal hidden information about the stock price, its performance, and where it is headed.

The following indicators have been used as states in addition to basic *open, low, high, close* prices in this project:

1. Stochastic $\%K = 100 \times \frac{C_t - L \cdot L_n}{H \cdot H_n - L \cdot L_n}$
2. Stochastic $\%D = \frac{\sum_{i=0}^{n-1} \%K_{t-i}}{n}$
3. Larry Williams' $\%R = 100 \times \frac{H_n - C_t}{H_n - L_n}$
4. Momentum $M = C_t - C_{t-4}$
5. $MA_n = \frac{\sum_{i=1}^{n} C_{t-i+1}}{n}$
6. $SY_t = (ln\, C_t - ln\, C_{t-1}) \times 100$
7. Price oscillator $OSCP = MA_5 - \frac{MA_{10}}{MA_5}$
8. $M_t = \frac{C_t + L_t + H_t}{3}$
9. $SM_t = \frac{\sum_{i=1}^{n} M_{t-i+1}}{n}$
10. $D_t = \frac{\sum_{i=1}^{n} M_{t-i+1} - SM_t}{n}$
11. $ASY_n = \frac{\sum_{i=1}^{n} SY_{t-i+1}}{n}$

where C_t, L_t, H_t are the closing price, the lowest price, and the highest price at time t, respectively. L_n and H_n are the lowest price and the highest price in the past n days.

The basic data price data contains five fields of *open, low, high, close,* and *volume*. For each currency, on the other hand, 11 technical indicators are added. With 5 currencies used in this study (subsection 4.2.1 and Table 1), the state size would be $|p| + |i| + |h| + |c| = 5 \times 5 + (5 \times 12) + 5 + 1 = 91$. Alternatively, when no Technical Indicators are used, the state size would be $(5 \times 5) + (5 \times 0) + 5 + 1 = 31$.

4.2.3 Dataset Preparation

For each of the currencies, rows belonging to the same timestamp are merged to create one row of data. Some executions of the experiment are run with Technical Indicators, described in 4.2.2, added to the set of states. Technical indicators are calculated for each of the currencies. Figure 11 illustrates how values of currencies and their technical indicators (Figure 11. b) are merged together by timestamp.

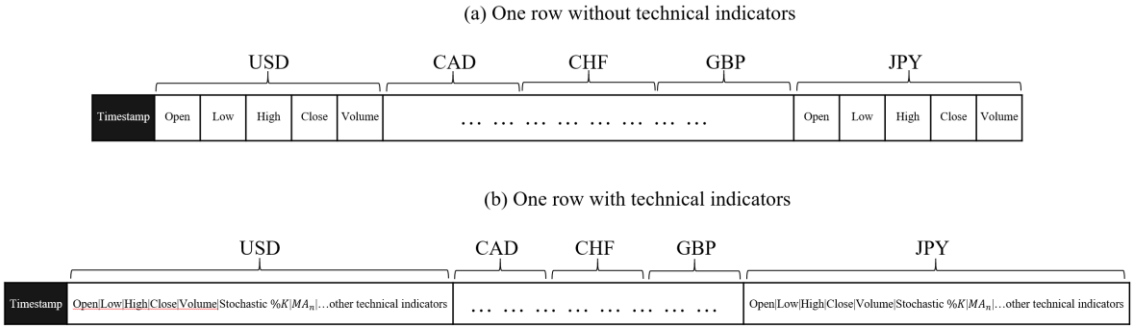


Figure 11. One row of data assembled consisting of several currencies.

Handling missing data

Missing data is referred to some fields, for example, the "Open" value on a given timestamp does not exist. A missing value is displayed as *Null* in SQL[1] or JSON[2], as *None* in Python programing language, or as *NaN*[3] in Pandas dataframe. There are 2 main scenarios that missing values can exist or be introduced to the data. Though improbable, it is possible that some values for some timestamps do not exist. Some of the technical indicators, especially those that require data from previous timestamps, naturally cannot produce results for the first timestamps where there is not enough information. This causes the first rows to contain missing values.

The best and most practical solution in our case to fix the issue with missing data is just to simply pretend they do not exist or in other words "drop" them. We use the `drop()` function in Pandas to eliminate such rows from the dataset.

Scaling

The exchange rates of currencies are not in the same ranges. For example, in our dataset, Euro to Sterling Pound exchange rate is the dataset lies between 0.64 and 0.95 while Euro

[1] Structured Query Language
[2] JavaScript Object Notation
[3] Not a Number

to Japanese Yen exchange rate is between 109.48 and 149.79. Therefore, they must be scaled to the same range to be used comparably.

One problem is that when training a model, we do not access to all possible values and in what ranges they could be. Unfortunately, there is no proper solution to this. Hence, we can devise a technique to minimize having data out of range. We can make random observations of our environment by taking random steps to collect states. By collecting a large number of scales, we can apply a scaler function to standardize the values of states. We use `StandardScaler` from Scikit-Learn library. It scales the values to unit variance (Scikit Learn, n.d.). For x the scaled value is calculated as $x' = \frac{x-\mu}{\sigma}$, where μ is the mean and σ is the standard deviation of the observations.

Train-Test Split

The datasets for the currencies were downloaded using the Tickstory application as mentioned in subsection 4.2.1. They are provided in minutes and daily resolutions. The daily resolution dataset contains data from January 1st, 2014 all the way through February 28th, 2021 which sums up to 1886 rows of data. Data from the beginning through December 31st, 2019 is dedicated for training, and for testing we use the remaining which starts on January 1st, 2020.

4.2.4 Actions

The basic actions that can be performed on a stock market are to *buy*, *sell*, or *hold* the shares. For the n stocks in the portfolio, the possible actions are defined as

$$A = [a_1, a_2, \dots, a_n], a_i \in \{buy, sell, hold\}$$

where a_i is the action for stock with the index of i in the portfolio.

For the n stocks there can be various combinations of actions. For example, for three stocks one action could be [buy, hold, hold]; which means buy stock 1, hold stock 2, and hold stock 3. Similarly, the action [hold, sell, sell] means to hold stock 1, to sell stock 2, and to sell stock 3. As you can see in the example, for n stocks in the portfolio there can be 3^n combinations of actions. In our study, as mentioned in subsection 4.2.1, with the five currencies, there will be $3^5 = 243$ possible actions.

A question that remains here is that what do sell and buy mean, that how many of the shares of each stock should be sold or bought. Deciding on the number of shares to be included in the trade is a whole new decision-making problem. Therefore, for the sake of this research and simplifying the approach, the *sell* decision for stock *n* means to sell all the available shares. In the same way, the *buy* decision for stock *m* means to buy as many shares as there is cash left.

As we manage a portfolio, there is more than one stock available to trade. This means that an action might mean selling some of the stocks and buying some other ones so additionally, we make two more adjustments. First, we perform the "sell" action. This will provide the agent with more cash in hand to perform buy action(s), if any. Second, all the stocks in the buy list, are valued equally. This means that we iterate through the buy list, buy 1 share of the first stock, 1 of the second one, and so on; then we start from the top and buy one share of a stock at a time until there is not enough cash in hand is left.

Furthermore, for the purpose of this study, we only assume that the trade commission fee is a percentage of the trade and there is no floor or ceiling defined for it. The commission fee is deducted from the cash in hand as soon as the trade, buy, or sell, is made. We also avoid over-drawing from the available cash balance. Before a "buy" action is performed, we ensure there will be enough cash left to pay the commission fee.

4.2.5 Reward Function

At the beginning, at time $t = 0$ the initial total value, *V*, of the portfolio of *n* assets is:

$$V_0 = p_1 \times h_1 + p_2 \times h_2 + \cdots + p_n \times h_n + c \tag{3.1}$$

where, p_i is the value per share of stock i at time t, h_i is the quantity or weight of stock i in the portfolio for $1 \leq i \leq n$, and c is the amount of available cash. For simplicity we can conclude:

$$v_i = p_i \times h_i, 1 \leq i \leq n$$
$$V_0 = v_1 + v_2 + \cdots + v_n + c \tag{3.2}$$

In the very beginning at $t = 0$ the quantity of all assets are zero, therefore, the value of the portfolio equals the amount of available cash $V_0 = c$. The goal of the agent is to maximize the total value of the portfolio of n assets at time t. The initial portfolio, V_0 is enforced externally and the agent does not have control over it. Additionally, the trading

process does not end so that the length of the session is unknown. Based on descriptions of (Jiang, et al., 2017) the goal is equivalent to maximizing the average accumulated return R:

$$R(s_1, a_1, \dots, s_t, a_t, s_{t+1} = \frac{1}{t} . \frac{V_t}{V_0} = \frac{1}{t} \sum_{i=1}^{t+1} V_i \tag{3.3}$$

$$= \frac{1}{t} \sum_{i=1}^{t+1} r_i \tag{3.4}$$

where V_i is the value of the portfolio at time i.

The immediate reward of an individual episode is r_t/t. To ensure the fairness of the reward function the denominator t is considered. It is worth mentioning that the transaction cost is taken into account while performing a trade.

An alternative option for obtaining the cumulative reward is to use the accumulated portfolio value (Moody, et al., 1998). The profit here is $P_t = V_{t+1} - V_t$ that can be considered as the immediate reward of action a (See Figure 12 for an illustration).

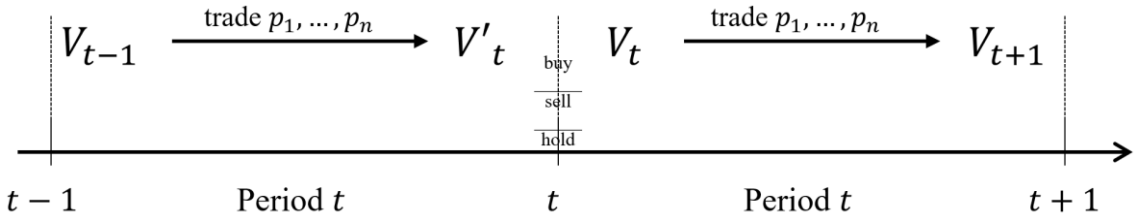


Figure 12. Performing a trade at time t and the effect of the trade on the value of the portfolio

4.2.6 Exploration versus Exploitation

This study uses a combination of two methods - decaying epsilon and ε-greedy approach. The algorithm utilizes a global ε value and another independent one within the episodic iterations. The process initially starts with $\varepsilon = 1.0$, so episode 1 begins with this value. ε-greedy is utilized within the episode thus the value of ε decreases after each step. Once episode 1 terminates, the value of global ε has not been influenced by the episodic ε, thus $\varepsilon = 1.0$. Before the next episode starts, the global ε is decayed and passed into the episodic run.

The process starts with ε_0 and decays to ε_m after iterating through m episodes. e is the total number of episodes intended to run for the training process where $e > m$. Therefore, episode i starts with the value:

$$\begin{cases} \varepsilon_i = \varepsilon_0 - \dfrac{(\varepsilon_0 - \varepsilon_m) \times i}{m} & , 0 \leq i < m \\ \varepsilon_m & , m \leq i \leq e \end{cases}$$

where $0 \leq \varepsilon_0 \leq 1$ and $0 \leq \varepsilon_m < \varepsilon_0$.

This setup will give us the freedom to experiment in different scenarios. We can disable the ε-decay by setting $\varepsilon_m = \varepsilon_0$ and m to any number. It is also possible to deactivate the ε-greedy of each episode by setting $\varepsilon_g = 1$ so the ε value within an episode's run will remain the same.

4.3 Approximation Model

The prices of the assets that comprise the states reflect the information that an agent needs in order to make decisions and take actions (Lo, et al., 2000). However, the prices are not discrete. The number of states involved in our environment can also quickly grow by increasing the portfolio size or adding prices of the previous timeframes to the current state. Consequently, tabular methods cannot be applied to our problem and the need for approximation methods arise to generalize our knowledge.

In this study, we implemented three variations to approximate the optimal policy using *Q*-learning. Before getting into the details of the implementations, a short explanation of *Q* modeling is given. Next, we provide an overview of the process. Further, in subsection 4.3.2 the $Q(s, a)$ function is approximated using ANNs. Then in 4.3.3 additional layers for Convolutional Neural Networks are added. CNNs help to re-arrange the state and change the dimensionality that may help capture additional information about the relations between features. Finally, we explain a somewhat different approach of the Actor-Critic method of RL in 4.3.4 by approximating actor and critic functions. All approximators will still rely on ANNs.

4.3.1 Process

The process starts by reading the data. Technical indicators are added in the pre-processing phase if they are required. Next, a model of the environment is created. The environment object keeps track of the portfolio, how many items of each asset there is, what the portfolio is worth, and what the current *state* is. Given an *action*, a utility function in the environment can take a step. When a step is taken, possibly a trade or multiple trades can take place. The details of how actions are performed and trades are handled are discussed at 4.2.4. When a *step* is taken, reward, next state, and a flag that indicates if the agent has reached the terminal state are returned.

We have also added a replay buffer memory feature (Lapan, 2018) where the last n states, actions, rewards, next states, and terminal state flag are kept in a memory object. As the agent takes steps and progresses in the environment, after each step, the initial state, the taken action, the received reward of the action, the resulting state, and the terminal state flag are added to the replay buffer memory. This is an optional feature and can be used during the training process. As our method is off-policy, we can take a sample of size $m \leq n$ and use those for training the policy.

The *agent* learns the environment where the model(s) of the policy is kept and trained based on the interactions the agent has with the environment. In the beginning, the agent starts at the initial state of the environment. The process ensures this by setting the *reset* function of the environment. The set of states are passed to the agent where it is decided what action should be taken. The resulting action, indeed, can be random during the training process depending on the explore-exploit strategy. The *environment* then takes a step based on the resulting *action* is passed. The process continues until the last possible step is taken in the environment. This is indicated by the *done* flag returned from the *step* utility function of the environment.

In the following, more details on the inner makings and architecture of the agent's policy learning are given.

4.3.2 Artificial Neural Network

Q-learning is the main method that is utilized in this work. Given a set of states our Q function should output an optimal action that is expected to maximize the return. This means that states are fed into the "model" as features and actions are the "targets" (Lapan, 2018). Figure 13 depicts an example of such modeling. You can find the code that implements the ANN in Appendix A: Artificial Neural Network in Keras.

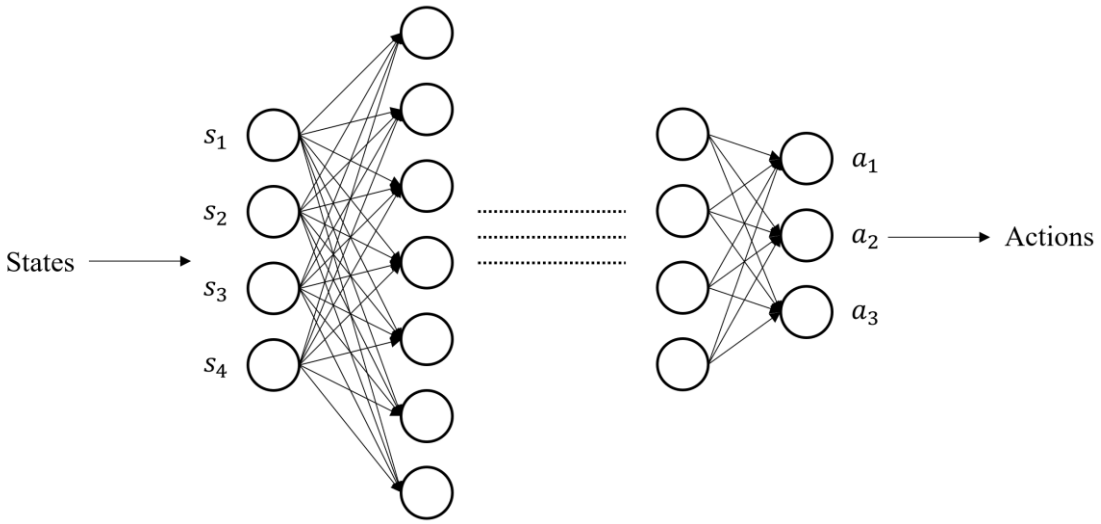


Figure 13. An Artificial Neural Network for the Q-function

However, *actions* are not the targets. The target is in fact the expected return of the taken action. Procedure 3 introduced a solution for this problem. By choosing actions at random with the probability of ε, the algorithm is expected to visit as many actions as possible given a state. This especially holds true when the number of episodes is sufficiently large.

When an action is chosen and a step is taken, environment returns a new set of state and the reward for the action. The network (Q function) performs an update by setting the target as (Mnih, et al., 2015):

$$target = R + \gamma \max_a Q(s', a) - Q(S, A)$$

Those actions that led to higher rewards receive positive signals and are strengthened in the neural network.

To retrieve a decision from the Q function model, the set of state is provided to the network as a feature vector input and the model returns the estimated value of "return" for

each action (Figure 13). Therefore, the best action to take at this stage is the one with the highest expected return:

$$a = arg\max_{a} Q(s, A)$$

where A is set of all possible actions.

4.3.3 Convolutional Neural Networks

In this study, the effect of adding a CNN to our network which learns the Q function is investigated. A CNN network could find hidden trends or patterns as it convolutes the feature vector and changes the dimensionality.

As illustrated in Figure 14 a batch of states are provided as the input to the CNN, a *Conv1D* layer is added and a *MaxPool1D* layer takes the output of the Conv1D layer activated with the *Relu* function. In the next step, the whole layer is *flattened* using *Soft-max* activation to be used as the input to an ANN. In other aspects, the way this CNN implementation works is similar to how the ANN is established. The code that creates the CNN used in this study can be found in Appendix B: Convolutional Neural Network in Keras.

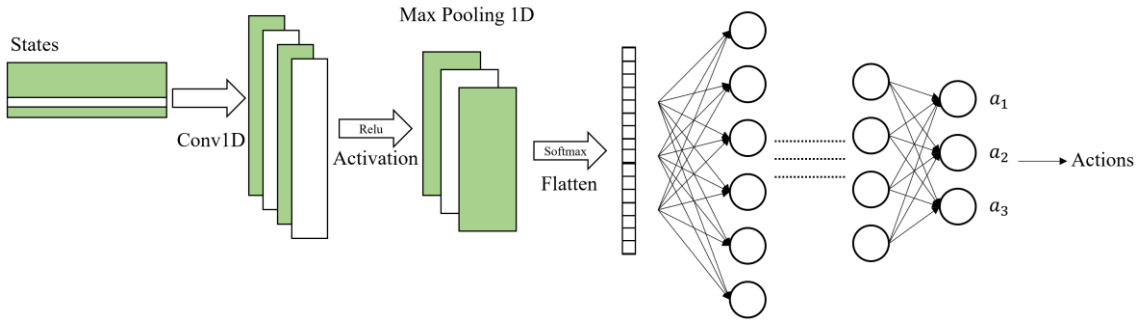


Figure 14. Convolutional Neural Network used for the Q-function modeling

Another difference to an ANN implementation in this project is during training the network. When a sample of states is taken either as a sample from the replay memory buffer or from the past "b =*batch size*" states in the dataset, it is ensured that the input to the CNN network is a matrix size "*batch size*" by "*state size*".

4.3.4 Actor – Critic Neural Networks

In this study, a deep deterministic policy gradient is implemented that consists of four components: local actor, local critic, target actor, and target critic. This method combines the state-action function of Q (critic) with the policy's value function (actor) (Laing, et al., 2018).

The *local actor* returns the best possible action based on the state of the environment. Then the *local critic* evaluates the local actor's proposed action as a Q function would. The output provided by the local critic directs the local actor towards its gradient (Saito, et al., 2018). Similarly, the *target actor* provides the optimal *next action* based on the *next state* and the *target critic* evaluates the provided *next action*. The target value from the target critic function is used to push the local critic function towards its gradient.

Finally, the *target actor* and the *target critic* are softly updated from the *local actor* and the *local critic* (Laing, et al., 2018). Figure 15 depicts the networks and how they are connected to each other.

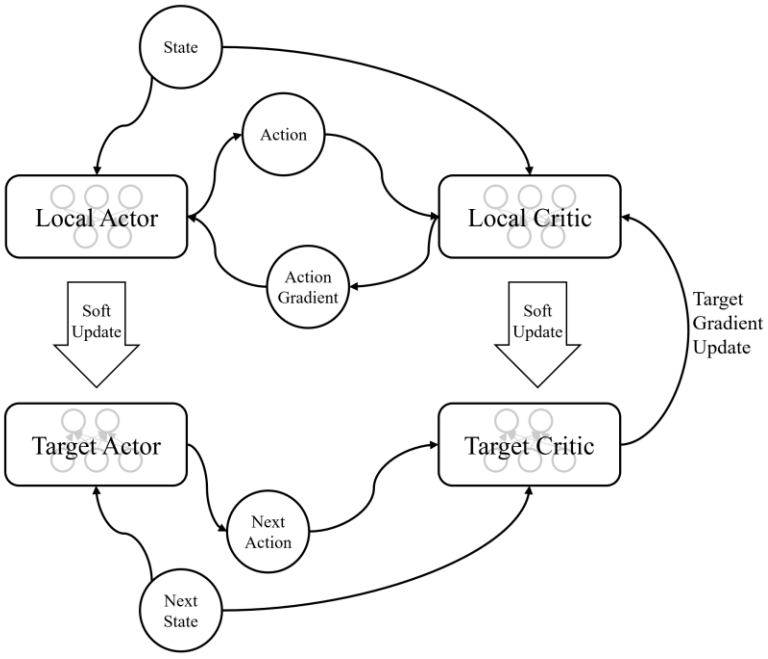


Figure 15. The architecture of an Actor-Critic reinforcement learning with local actor, local critic, target actor, and target critic.

To prevent drastic changes in the target models, the weights in the target networks (w') are updated based on the weights (w) in the local network and the τ parameter ($0 \leq \tau \leq 1$) (Saito, et al., 2018):

$$w' \leftarrow \tau . w + (1 - \tau) . w'$$

The implemented code used in this study can be found in Appendix C: Actor-Critic Networks in Keras.

4.4 Evaluation

Although a challenge of evaluating algorithmic trading and in general algorithms that deal with financial data is that many factors are involved in measuring how successful an algorithm is, measuring the profit remains one of the most important indicators. We will measure the profit in three ways:

1. Gross Profit: if the algorithm starts with an initial investment of X € in cash and complete trading with the portfolio value of $V_{terminal}$ €, the gross profit is $\frac{V_{terminal} - X}{X}$.
2. Profit against inflation: it is the same as gross profit with the difference that the inflation is taken into account. In a similar manner, for the inflation rate of I %, the profit against inflation is calculated as $\frac{V_{terminal} - X}{X + (X \times I\%)}$.

   According to Statistics Finland (OSF, n.d.), the year-on-year inflation in Finland in March of 2021 was calculated as 1.3%. Therefore, the starting investment of e.g., 10000.00 € at the beginning of 2020 would approximately be equivalent to 10130.00 € in March 2021.
3. Growth against "buy and hold": starting with an initial investment of X € in cash and having n currencies (stocks) in our portfolio, we divide the initial investment cash into n equal portions ($\frac{X}{n}$) we buy as many shares as possible with the given portion. This is equivalent to taking the "action" [buy, buy, …, buy] for all n available currencies (stocks) in our portfolio. We perform no other actions anymore, or in terms of our environment we take the "action" [hold, hold, …, hold] until the terminal state is reached (end of data). Based on the value vector definition from subsection 4.2.5 the value of our portfolio at the end is

$$V_{terminal} = p_1 \times h_1 + p_2 \times h_2 + \cdots + p_n \times h_n + c$$

Now we compare this value against the final value of our portfolio Y € that is the result of running our RL algorithm. The growth against buy and hold is, therefore:

$$growth = \frac{V_{terminal} - Y}{Y}$$

Following this assumption, starting with 10000.00 € investment, buying an equal number of stocks, which would be 80 of each of the currencies, holding until the last ticker, and selling all of them at the last step, would yield 10631.84 €.

In the following chapter, we compare all approaches using the three aforementioned metrics and present the results.

5 RESULTS

In this chapter, we present the performance of three approaches of modeling using ANNs, CNNs, and Actor-Critic networks. The performance metrics of all methods are presented as well as other considerations such as the episode's run time or processor and memory utilization. The experiments have been run with variations of the following hyperparameters on the main network types introduced in section 4.3:

1. Topologies of the neural network.
2. **Activation functions** are applied to the output of a layer to transform the values that will be fed into the next layer (Sharma, et al., 2020). Three functions have been used in the hyperparameter study for all network types:
    a. **ReLU** (Rectified Linear Unit) is a non-linear activation function $f(x) = \max(0, x)$ that deactivates the neurons in the network when the transformation value is smaller than 0 (Sharma, et al., 2020).
    b. **Sigmoid** is a non-linear function $f(x) = \frac{1}{1+e^{-x}}$ which transforms values to (0, 1) range (Sharma, et al., 2020). For small values, e.g., $x < -5$, a value close to 0 is returned (Keras, n.d.).
    c. **Softmax** can be interpreted as a combination of several sigmoid functions. It returns the probability of a data point being in a class out of K possible classes $\sigma(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$ (Sharma, et al., 2020).

3. **Optimizers** provide feedback on how far the predictions are to the real values during the training process. They optimize the process to approach the objective $f(x)$ function (Zeiler, 2012). The following three optimizers have been used in this study:
    a. **Adam** (Adaptive Moment Estimation) combines advantages from AdaGrad and RMSProp and exponentially decays the average of past gradients and at the same time moves slowly towards a minimum of errors (Kingma & Ba, 2017).
    b. **Nadam** (Nesterov-accelerated Adaptive Moment Estimation) is an Adam optimizer with Nesterov Momentum (Keras, n.d.).
    c. **Adadelta** is based on the Stochastic Gradient Descent (SGD) method that tries to follow the steepest descent. It takes steps towards the negative gradient on each training sample batch (Zeiler, 2012).
4. **Loss functions** are for the purpose of computing values that a model should minimize (Keras, n.d.). The following functions have been applied to the models in our study:
    a. **MSE** calculates the Mean Squared Error of the predictions to the real values.
    b. **MAPE** computes the Mean Absolute Percentage Error between the predictions and the actual target values.
    c. **MSLE** returns the Mean Squared Logarithmic Error between the predicted values and the target values.
    d. **Categorical Cross-Entropy** is used when several target classes are available, and the objective is to predict the likelihood of each of the available classes to be the desired target class. This loss function works on a one-hot encoded representation of target classes. Alternatively, **Sparse Categorical Cross-Entropy** works in a similar manner on class labels provided as integer values.
5. **States**: with or without technical indicators[1] (subsection 4.2.2).

The experiments trained on 1200 to 2000 episodes where they all started with epsilon value of $\varepsilon = 1.0$. After each episode, the value of ε linearly decreases to the minimum of

[1] TI is the abbreviation for Technical Indicators used in figures.

0.1 on the 100th episode before training finishes and stays the same for the rest of the training process. In each episode, the provided value of epsilon would decay at the rate of 0.99999 after each step within an episode.

In some cases, during the hyperparameter study, convergence did not happen. The algorithm either made arbitrary decisions or took the same action regardless of the provided state during the test process. This issue could happen for several reasons. First, the network topology and the hyperparameters might have been wrong or incompatible with what the desired output was. Another possibility is that some of the initializations, such as states or initial neuron weights, were not optimized. It is stated at (Lazy Programmer Inc.) in some cases, the convergence of the model is not guaranteed. Unfortunately, the investigation and probing the reasons in such cases are outside the scope of this study. Therefore, since they do not add merits to the project, we have decided to remove them from the results and discussions.

5.1 Hyperparameters Study for Artificial Neural Network

A variety of hyperparameters were utilized in the ANN implementation of this project. In Table 2 you can find the results of all tests with the provided hyperparameters and what returns they yielded at the end of the trading period. Models trained using the same optimizer, activation, and loss function are grouped together.

Figure 16 displays the changes of the portfolio value over time and how the values have changed from the beginning of the trading period until the end. The graphs, additionally, provide a baseline comparison of the performance of the agent against the index of the trading market, referred to here as Buy & Hold, and the inflation rate in Finland. Only those agents that performed better than Buy & Hold strategy can be considered as profitable, the higher above the Buy & Hold line the better. The same presentation logic is applied on graphs for CNN and Actor-Critic agents in the following sections.

Activation	Optimizer	Loss	Technical Indicators	Layers	Return (%)	Return against Buy & Hold (%)	Return against Inflation (%)
ReLU	Nadam	MAPE	Yes	128	-1.3	-7.16	-2.56
			Yes	128×64×32	7.46	1.08	6.08
			No	128	4.83	-1.4	3.49
			No	128×64×32	-3.77	-9.49	-5.01
Sigmoid	Adadelta	MAPE	Yes	128	9.1	2.61	7.7
			Yes	128×64×32	-4.46	-10.14	-5.69
			Yes	64×32	-2.42	-8.22	-3.68
			No	128	7.05	0.69	5.68
Softmax	Adam	Categorical Cross-Entropy	Yes	128×192×256	-5.81	-11.41	-7.02
			Yes	128×256	1.87	-4.18	0.56
			Yes	128×64	-0.66	-6.56	-1.94
			No	128×256	-8.44	-13.88	-9.62
		MSE	Yes	128×64	11.93	5.28	10.5
			No	128×256	-2.3	-8.1	-3.55

Table 2. Effects of hyperparameters on Artificial Neural Networks model performance

Most of the tests did not perform well here. This can be attributed to the lower or the lack of ability of an ANN to capture intricate and subtle signals from the state set. Also, the ANN method produced 16 models that did not converge.

It was expected that the Categorical Cross-Entropy loss function would improve the performance of the agent. However, this was not the case and in nearly all instances of testing with this loss function, no profit was yielded. This issue can be attributed to the lack of sophistication of the network or perhaps, more episodes needed to run for the ANN to capture the nuances of the states.

Figure 16. Portfolio value changes across time with ANN models.
X-axis: days, Y-axis: Relative growth ($^{V}/_{Investment}$).
Dash-dot line: Constant Buy & Hold growth. Dotted line: Constant inflation growth.

On the other hand, three networks performed well: 1) Sigmoid, Adadelta, MAPE with 128 hidden layers using all technical indicators, 2) Sigmoid, Adadelta, MAPE with 128 hidden layers without using technical indicators, and 3) Softmax, Adam, MSE with 128×64 hidden layers using technical indicators. The first agent's performance is somewhat unexpected. The simple topology of the network could have been the hindrance to capturing the essence of states, however, this simplicity and having a quite optimized size between the input vector size of 91 and output vector size of 243 might have been an advantage. For the second agent, the unexpected performance, given the fact that the state size is small with no technical indicators to carry more information, could be due to some randomness in the initialization, or the actions taken by the agent, are in fact not optimal, but in this case, they resulted in a profit. This can be investigated and tested on more data in a future research. The third case, however, provides a good sign as the combination of activation, optimizer, and loss functions has been shown to perform well on RL methods.

5.2 Hyperparameters Study for Convolutional Neural Networks

Table 3 shows how different combination of hyperparameters influence the performance of the agent as well as the returns. For the following parameters in all CNNs used in this study, either default values were used, or constant values were set (Keras, n.d.):

- In `Conv1D` layer object:
    - *Filters* = 16 is applied to the input to output a feature map with the set dimensionality.
    - *Kernel size* = 8 specifies the length of the convolution window.
    - *Kernel initializer* = 'uniform'.
    - Default values for s*trides* = 1, *padding* = 'valid', *data format* = 'channels_last', *dilation rate* = 1, *use bias*= true, *bias initializer* = 'zeros'.
- In `MaxPooling1D` operation:
    - *Pool size* = 5 is the max pooling window size.
    - Default values for *strides*: same as *pool size*, *padding* = 'valid' which means no padding and *data format* = 'channels_last'.

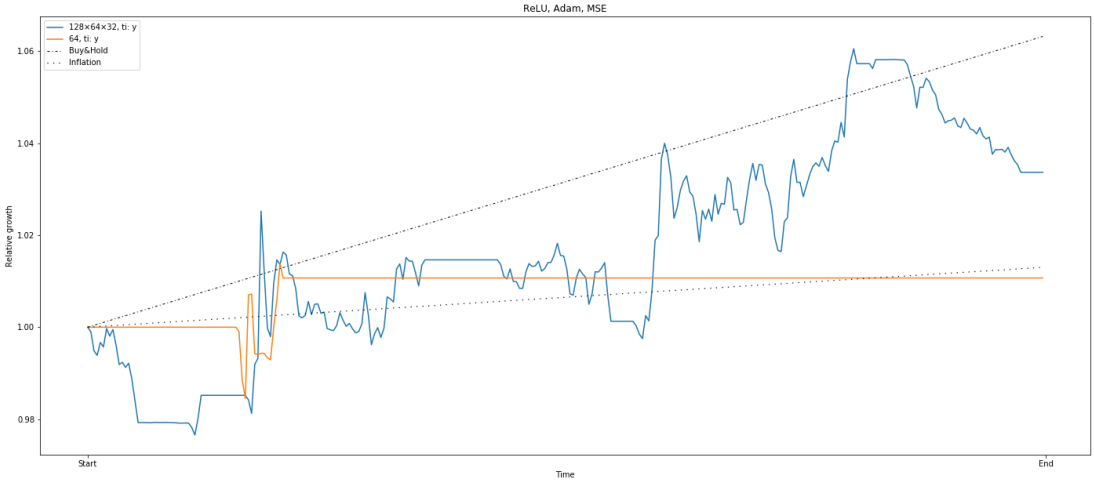
Activation	Optimizer	Loss	Technical Indicators	Layers	Return (%)	Return against Buy & Hold (%)	Return against Inflation (%)
ReLU	Adam	MSE	Yes	128×64×32	3.37	-2.78	2.04
			Yes	64	1.07	-4.94	-0.23
	Nadam	MAPE	Yes	128	0.4	-5.57	-0.89
			No	128	5.51	-0.76	4.15
			No	64×32	-2.63	-8.42	-3.88
Sigmoid	Adadelta	MAPE	Yes	128	8.83	2.36	7.43
			Yes	128×64×32	-5.09	-10.73	-6.31
			Yes	64×32	-0.58	-6.49	-1.86
			No	128	7.46	1.07	6.08
			No	128×64×32	0.96	-5.04	-0.34
			No	64×32	-1.98	-7.81	-3.24
Softmax	Adam	Categorical Cross-Entropy	Yes	128×192×256	-4.5	-10.18	-5.73
			Yes	128×256	2.46	-3.63	1.14
			No	128×192×256	0.65	-5.33	-0.64
		MSE	Yes	128×256	0.23	-5.72	-1.05
			Yes	128×64	10.21	3.66	8.8
			No	128×256	-0.48	-6.39	-1.75
		MSLE	Yes	128×64×32	-1.65	-7.49	-2.91
			Yes	64×32	2.64	-3.46	1.32

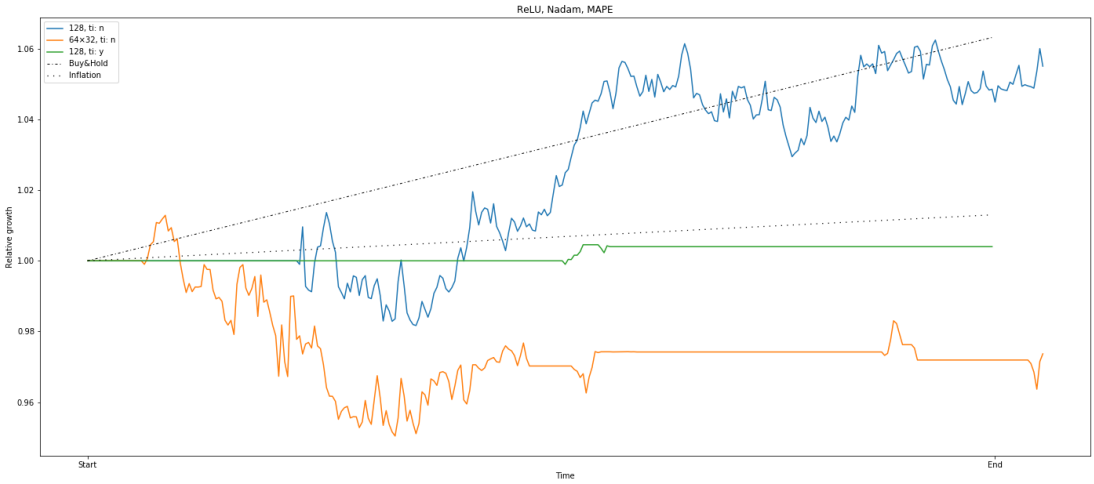
Table 3. Effects of hyperparameters on Convolutional Neural Networks model performance

Similar to the performance and behavior of ANNs, agents based on CNNs in most of the studied cases did not yield positive returns. Though, only 10 networks did not converge to obtain a policy.

In the same way as for ANNs, three of the CNN-based agents performed well. All these three agents are with the same hyperparameters as those for ANNs. Besides, more interestingly, the yields of CNN-based agents are well in the same range as those of the ANN-based agents. These results could be good indicators of what activation, optimizer, and loss functions, as well as optimal network topology would yield good results.

Convolutional Neural Networks Hyper-parameters





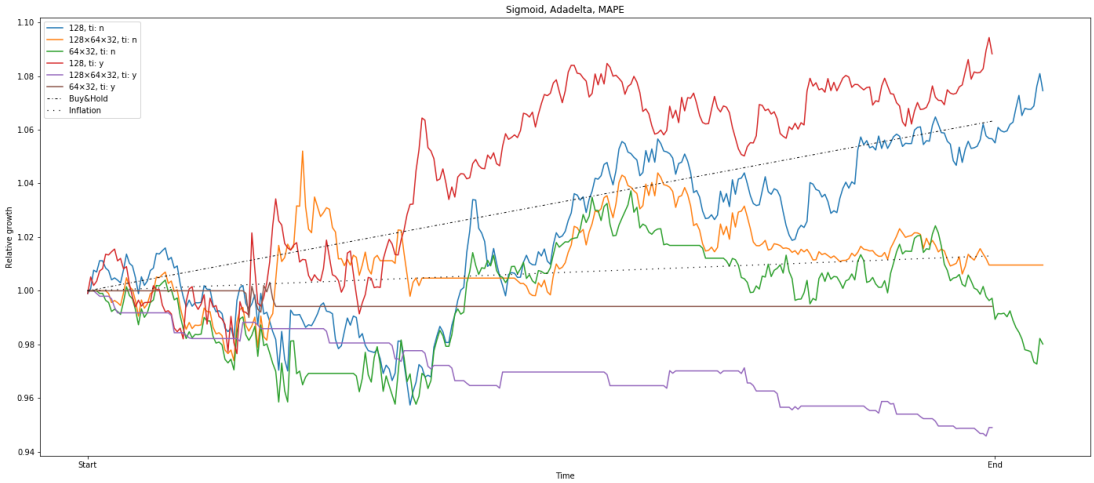


Figure 17. Portfolio value changes across time with CNN models.
X-axis: days, Y-axis: Relative growth ($^{V}/_{Investment}$).
Dash-dot line: Constant Buy & Hold growth. Dotted line: Constant inflation growth.

Figure 17 displays the portfolio values over time. Some of the plots demonstrate that some networks initially perform well, but later during the trading period, the value of the portfolios significantly decreases. These cases represent promising avenues for future research that can investigate the mechanisms driving such behaviors.

5.3 Hyperparameters Study for Actor-Critic Networks

As stated in subsection 4.3.4, an agent-based on actor-critic (A2C) consists of two ANNs: the actor and the critic. In this study, to simplify the network implementation, hyperparameters are used in both networks. The results of hyperparameter tuning for A2C networks are presented in Table 4.

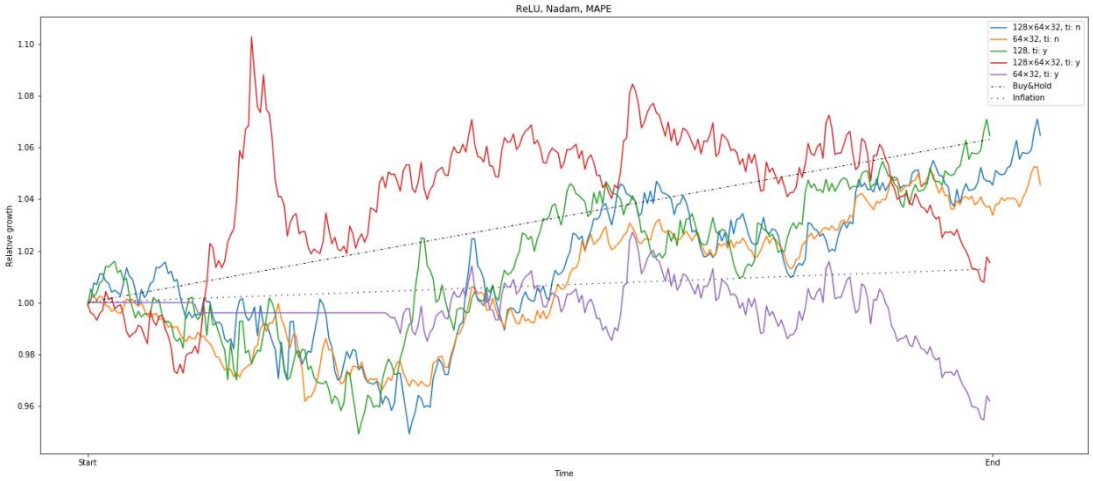
Activation	Optimizer	Loss	Technical Indicators	Layers	Return (%)	Return against Buy & Hold (%)	Return against Inflation (%)
ReLU	Adam	MSE	Yes	128×64×32	-1.42	-7.27	-2.68
			Yes	32	4.99	-1.25	3.64
			Yes	64	2.82	-3.29	1.5
			Yes	64×32	-1.5	-7.36	-2.77
	Nadam	MAPE	Yes	128	6.47	0.15	5.11
			Yes	128×64×32	1.55	-4.48	0.25
			Yes	64×32	-3.82	-9.53	-5.05
			No	128×64×32	6.47	0.14	5.11
			No	64×32	4.56	-1.66	3.22
Sigmoid	Adadelta	MAPE	Yes	128	-2.8	-8.57	-4.04
			Yes	64×32	-4.23	-9.92	-5.45
			No	128×64×32	8.71	2.25	7.32
			No	64×32	-4.16	-9.86	-5.39
Softmax	Adam	Categorical Cross-Entropy	Yes	128×256	10.45	3.88	9.03
			Yes	128×64	6.5	0.17	5.13
			No	128×256	6.12	-0.18	4.76
			No	128×64	6.51	0.18	5.14
		MSE	Yes	128×256	8.1	1.68	6.71
			Yes	128×64	5.41	-0.85	4.06
			No	128×64	6.44	0.12	5.08
		MSLE	Yes	128×64×32	6.51	0.18	5.14
			Yes	64×32	5.74	-0.54	4.38
			No	128×64×32	2.78	-3.33	1.46
			No	64×32	-4.53	-10.2	-5.75

Table 4. Effects of hyperparameters on Actor-Critic model performance

A2C-based agents seem to be more stable than ANN and CNN counterparts as the number of models that did not converge dropped to 4. The performance improvement of none of the A2C models indicates a significant advantage over other methods. However, the fact that in general, more agents yielded better results on average compared to ANN and CNN-based agents points out a promising direction to study these networks in the future.

Figure 18 depicts the overall performance of A2C agents over time. Similar to ANN and CNN agents, the combination *Sigmoid* activation function, *Adadelta* optimizer, and *MAPE* loss function as well as *Softmax* activation function, *Adam* optimizer, and *MSE* loss function for A2C networks, have contributed to training some models that perform the best. However, the network topologies of the best performing A2C models with similar hyperparameters are different from what ANN and CNN models have. Furthermore, an A2C model with *Softmax* activation function, *Adam* optimizer, and *Categorical Cross-Entropy* loss function performed much better than the other models. The network topology of 128×256 could be an indicator of what direction investigations should point to.

Actor-Critic Networks Hyper-parameters



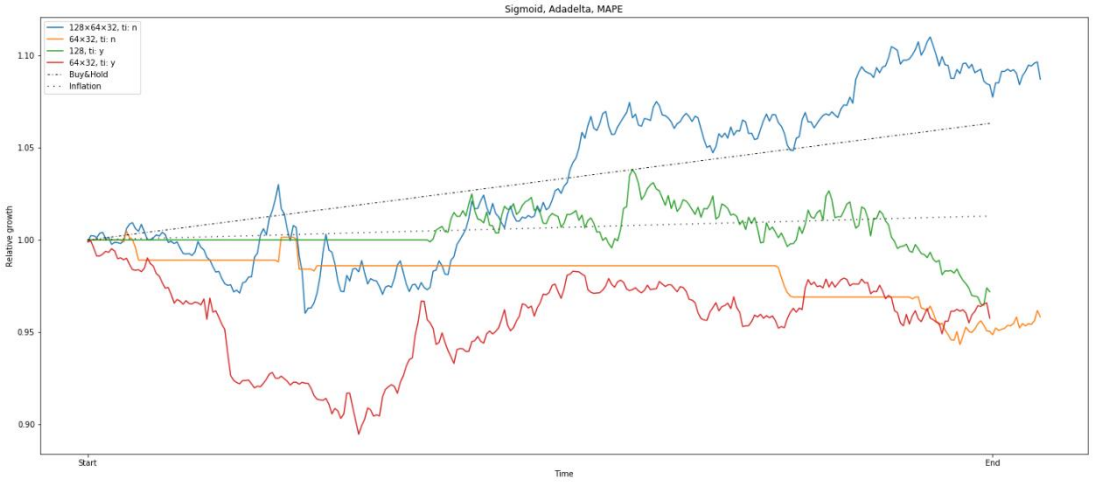


Figure 18. Portfolio value changes across time with Actor-Critic models.
X-axis: days, Y-axis: Relative growth ($^{V}/_{Investment}$).
Dash-dot line: Constant Buy & Hold growth. Dotted line: Constant inflation growth.

5.4 Runtime of Experiments

Specifications of the machine on which the experiments were carried out are explained in section 4.1. The average runtimes of episodes for the training process on the specified machine are presented in Table 5.

Agent type	Topology	Processor Cores	Average Episode Runtime (ms)
A2C	128	4.0 CPU(s)	43268
	128×256	1.0 CPU(s)	75535
	128×64×32	3.4 CPU(s)	49819
	128×64×32	6.0 CPU(s) + GPU	24397
	64×32	3.6 CPU(s)	44966
CNN	128	2.4 CPU(s)	23648
	128×192×256	1.0 CPU(s)	47934
	128×192×256	1.0 CPU(s) + GPU	32707
	128×256	1.0 CPU(s)	51001
	128×64×32	3.0 CPU(s)	22404
	128×64×32	1.0 CPU(s) + GPU	22667
	64×32	3.0 CPU(s)	16305
ANN	128	2.0 CPU(s)	19020
	128×192×256	1.0 CPU(s)	44106
	128×256	1.0 CPU(s)	43152
	128×64	1.0 CPU(s)	41580
	128×64×32	2.0 CPU(s)	27451
	128×64×32	1.0 CPU(s) + GPU	15165

Table 5. Average episode runtime of experiments

Interestingly, although the running on a GPU improves the performance, the difference is not significant. This is due to the nature of our process that is not optimized. The training of the model occurs on GPU in batches of usually size 32. But the environment's operations mostly occur on the CPU and the main memory (RAM). This back-and-forth switch between the CPU and GPU has contributed to a considerable amount of time being spent on the tensor conversions. This process can be improved by optimizing the environment operation's code and attempt to utilize GPU operations more. Although it is outside of the scope of this study, this optimization can be examined further in a future work.

6 CONCLUSION

6.1 Implementation

The major focus of this study was to understand how Reinforcement Learning can be utilized in the financial trading realm. We introduced and studied RL frameworks and their connections to the process of finding optimal actions for trading. We found that RL-based methods provide an advantage over alternative approaches. Using alternative approaches, such as growth or value prediction leaves the decision-making process an open question. On the other hand, RL methods take in the state of the environment, such as historical prices, and they output an optimal action that can be performed.

An interesting finding of running the experiments is that the convergence to an optimal policy does not easily happen; models are relatively insensitive to the state of the environment when unseen test data is fed into the algorithms. On the other hand, there are indications that the topology of the networks, especially the size of the hidden layers, should approximately match the state (input) and action (output, label, or target) space sizes. For instance, for the state dimension size of 91 and action space size of 243, hidden layers should represent those values more closely.

6.2 Comparison of Approaches

We covered three network types to model the decision-making based on the expected value of our portfolio in the future. ANN and CNN implementations all have an important factor in common: They are both based on one model that output actions based on states. A2C networks are quite different as they consist of two sets of networks, where the critic network evaluates and improves the actor network's suggested action. This additional step for A2Cs adds more complexity to the model and increases the time and memory consumption of such an approach. Nevertheless, these come with several added benefits. First, overall, A2C methods are more likely to yield more returns compared to the ANN and CNN counterparts. Second, they are more likely to converge to an optimal policy than the other two approaches are.

It is worth mentioning that evidently, CNNs have a better tendency to converge compared to ANNs'. This could perhaps be explained by the fact that the filtering and pooling processes in CNNs could possibly capture hidden trends that otherwise might go unnoticed by the ANN methods.

6.3 Effects of Hyperparameters

Surprisingly we found that using technical indicators is not a guarantee that the model would perform better compared to when they are not used. This might look counterintuitive as the technical indicators carry information on the previous data points that can be used to predict the next data points or the direction of prices. Though, this could possibly be due to the limited data and the limited number of episodes during the training process.

Activation functions, optimizers, and loss functions play an important role in the efficiency of the models. We showed that generally *Softmax* and *Sigmoid* activation functions perform better than the other options. *Adam* and *Adadelta* optimizers clearly improve the model performance. *MSE* and *MAPE* loss functions are also preferred choices.

Running over many episodes to "learn" the optimal policy makes the training process of RL methods very time-consuming. They need a large amount of data to train on and require several passes over them (episodes) to find all hidden paths. The fact that in most experiments, the obtained policy either is suboptimal or not converged close to an optimal policy, could be explained by the relatively small amount of data as well as the number of episodic runs.

The currency prices in the dataset over a long period of time are quite sporadic. They can vary significantly from one point to another point several months later. On the other hand, as the daily resolution is used, the difference between open, low, high, and close, could be relatively large and significant which might skew the state set. Using higher resolution price data, such as hour-by-hour or minute-by-minute ones, and on relatively shorter periods could alleviate this issue. This tactic also provides more data points where one point is closer to another one, therefore, eliminating the challenge of dealing with sporadic data. This issue will be discussed in the Future Work chapter.

6.4 Final Recommendations

Based on the results and discussion, we argue that A2C methods, in general, work better than ANN or CNN-based methods. Though, if the model complexity is of concern where A2C is difficult to implement and maintain, a CNN method is preferred to a more common ANN one.

As stated earlier, we would recommend using a more balanced network topology with at least two and a maximum of four hidden layers. The size of the first hidden layers should be closer to the state dimension size and the number of units in the last hidden layers should match the action space size closer. There could be more in-depth studies done on the topology of the network to find out what sizes of hidden layers would work better see discussion in the Future Work chapter below.

Relatively better choices for activation functions would be *Softmax* and *Sigmoid*. *Adam* and *Adadelta* would be recommended for optimizers. *MSE* and *MAPE* also seem to be good choices for the loss function. However, as the RL task here is in nature a classification problem, *Categorical Cross-Entropy* or its variants could be good choices as well, although, more experimentation and study into using it is needed.

7 FUTURE WORK

This study is an attempt at investigating the advantages, challenges, and limitations of Reinforcement Learning in financial portfolio management and trading. In no way is this research a comprehensive analysis of methods or algorithms that can be used in live financial markets. Nonetheless, the findings from this work can be used as an instrument to step in the right direction and towards building sophisticated systems that can make trading recommendations or even carry out trades.

In this chapter, we go through some points to expand upon the findings and propose potential future research extension. We sketch out a baseline upon which the current study can be improved and outline what other experiments can be investigated further.

7.1 Further Studies on Networks

7.1.1 Topology Study

The network topologies we experimented with are quite limited. We have only utilized networks with the maximum of three hidden layers. One reason for this is the amount of data we have available and the limited amount of time to run 1000-2000 episodes per experiment. We believe that a more complex network can be studied should we have a larger dataset and more time to run longer experiments. To increase efficiency and to avoid overfitting, we could have dropouts in the networks.

The number of units per hidden layer we experimented with changes by factors of 2^n from one layer to an adjacent layer. Other unit numbers per hidden layer can be studied further to evaluate what effects they could have.

7.1.2 Hyperparameter Tuning Study

The experiments with optimizers, activation and loss functions we performed do not include all well-studied and well-tested ones. In a future study other options can be researched and evaluated.

There are several hyperparameters that were left as default in the experiments. For instance, on layers kernel initializer, bias initializer, kernel regularizer, bias regularizer, activity regularizer parameters, or on model metrics and loss weights may have room for tuning. We chose the batch size of 32 for training that was due to the size of our training set. With a larger dataset, various batch sizes can still be examined.

Convolutional Neural Networks offer more parameters to experiment on in order to achieve better results. In this study, only one CNN layer was used. Adding more layers consecutively could help with capturing hidden trends in data. Furthermore, several other hyperparameters of CNN can still be tuned. We believe that tuning the number of filters, kernel size, strides, padding, and pool size parameters could improve the performance of the model.

7.1.3 Other Networks

In section 3.1 we mentioned the use of Recurrent Neural Networks and Long Short-Term Memory methods and how they can be applied to Reinforcement Learning for financial trading. Studying these strategies was not in the scope of this study since such techniques are more complex to implement. Yet, the power and the benefit they provide cannot be underrated.

Prior research shows that RNNs and LSTMs have good track records of predicting the direction of stock. In a future study, the output of such a network can be added to the environment's state which is subsequently provided to the agent's model. More elaboration can be made on the prediction. For instance, the RNN can predict the next several price points which are then added to the state. This insight will most probably impact the performance of the agent positively.

7.1.4 Improving Stability

Because there is a high correlation between the states and the actions, the neural network that is modeling the value function is not guaranteed to converge to the optimal policy. On the contrary, it could oscillate or, worse, even diverge. (Lapan, 2018) points out a strategy to improve the training stability by creating a main network and a target network. The target network is used for training on the $Q(s, a)$ values of the Bellman equation. Then the target network is only synchronized with the main network occasionally after N iterations, where N, depending on the size of the dataset, is usually set to a rather large number.

Asynchronous Advantage Actor-Critic (A3C) is an extension of A2C methods by parallelizing several child processes across multiple machines. Each agent in a child process uses local training data to calculate the gradient. The central process is responsible for collecting the gradients from child processes and performing a Stochastic Gradient Descent on the shared network model, which propagates the new weights to the agents to ensure a shared policy is maintained across all processes (Babaeizadeh, et al., 2017).

7.2 Further Studies on Other Datasets

Using a higher resolution of data - for example, data points for every minute - and performing trades on them is considered closer to real-life use cases for portfolio management. In a future study, a model can be trained for a relatively shorter period of time, for example, 24 to 72 hours' worth of data. This length of data would still have a large number of data points, given the higher resolution. In turn, there would typically not be a problem with larger differences between price points compared to those of daily trading data.

Such high resolution might have the disadvantage that the model would become inaccurate or irrelevant after only a few hours. We could set up a process where a model is trained on the past X hours of data and the online trading agent would use this model. While the model is online and performs actions, another process would train a new model with a more up-to-date dataset. Once the new model is trained and validated would the online and older model be replaced by the newly trained model.

REFERENCES

Agarwal, A., Hazan, E., Kale, S. & Schapire, R., 2006. *Algorithms for portfolio management based on the Newton method.* s.l., s.n., pp. 9-16.

Arulkumaran, K., Deisenroth, M. P., Brundage, M. & Bharath, A. A., 2017. A Brief Survey of Deep Reinforcement Learning. *CoRR.*

Babaeizadeh, M. et al., 2017. *GA3C: GPU-based A3C for Deep Reinforcement Learning.* Barcelona, Spain, s.n.

Bellman, R., 1957. A Markovian decision process.. Volume 6, pp. 679-684.

Bengio, Y., Goodfellow, I. & Courville, A., 2016. *Deep Learning.* s.l.:MIT Press.

Cumming, J., 2015. *An Investigation into the Use of Reinforcement Learning Techniques within the Algorithmic Trading Domain,* London: Imperial College London.

Dempster, M. A. H. & Leemans, V., 2006. An automated FX trading system using adaptive reinforcement learning. *Expert Systems with Applications,* pp. 543-552.

Dixon, M. F., 2017. A High Frequency Trade Execution Model for Supervised Learning. December.

Espinosa-Leal, L., Chapman, A. & Westerlund, M., 2020. Autonomous Industrial Management via Reinforcement Learning. *Journal of Intelligent & Fuzzy Systems,* December, Volume 39, pp. 8427-8439.

Farmer, D. & Skouras, S., 2012. *Review of the benefits of a continuous market vs. randomised stop auctions and of alternative Priority Rules (policy options 7 and 12),* s.l.: UK Government Office for Science.

Ganesh, P. & Rakheja, P., 2018. Deep Reinforcement Learning in High Frequency Trading.

Gosavi, A., 2009. Reinforcement Learning: A Tutorial Survey and Recent Advances. *INFORMS Journal on Computing,* 5.pp. 178-192.

Haugen, R. A., 1986. *Modern investment theory.* s.l.:Prentice Hall.

Jain, A., Nandakumar, K. & Ross, A., 2005. Score normalization in multimodal biometric systems. *Pattern Recognition,* 38(12), pp. 2270-2285.

Jansen, S., 2020. *Machine Learning for Algorithmic Trading.* 2nd ed. Birmingham: Packt Publishing Ltd..

Jiang, Z., Xu, D. & Liang, J., 2017. *A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem,* s.l.: s.n.

Kanwar, N., 2019. *Deep Reinforcement Learning-based Portfolio Management,* Arlington, Texas: The University of Texas at Arlington.

Kelly, J. L., 1956. A new interpretation of information rate. *The Bell System Technical Journal,* Volume 35, pp. 917-926.

Keras, n.d. [Online] Available at: https://keras.io/ [Accessed 2021].

Kingma, D. p. & Ba, J., 2017. Adam: A Method for Stochastic Optimization.

Konda, V. R. & Tsitsiklis, J. N., 2003. On Actor-Critic Algorithms. *SIAM Journal on Control and Optimization,* pp. 1143-1166.

Laing, Z. et al., 2018. Adversarial Deep Reinforcement Learning in Porfolio Management.

Lapan, M., 2018. *Deep Reinforcement Learning Hands-On.* Birmingham: Packt Publishing Ltd..

Lazy Programmer Inc., 2018. *Artificial Intelligence: Reinforcement Learning in Python.* s.l.:Udemy Inc..

Li, B. & Hoi, S. C. H., 2014. Online Portfolio Selection: A Survey. *ACM Computing Survey,* 46(3).

Li, X. et al., 2014. Empirical analysis: stock market prediction via extreme learning machine. *Neural Computing and Applications,* Volume 27, pp. 67-78.

Lo, A. W., Mamaysky, H. & Wang, J., 2000. Foundations of technical analysis: Computational. *The Journal of Finance,* Volume 55.

Mnih, V. et al., 2015. Human-level control through deep reinforcement learning. *Nature.*

Moody, J., Wu, L., Liao, Y. & Saffell, M., 1998. Performance functions and reinforcement learning for trading systems and portfolios. *Journal of Forecasting,* Volume 17, pp. 441-470.

Nasdaq, n.d. *https://www.nasdaq.com.* [Online]
Available at: https://www.nasdaq.com/glossary/c/currency-appreciation
[Accessed May 2021].

Nino, J. et al., 2017. *Algorithmic Trading Using Deep Neural Networks on High Frequency Data.* s.l., s.n., pp. 144-155.

Numpy, n.d. [Online] Available at: https://numpy.org/ [Accessed 2021].

Nunno, L., 2014. *Stock Market Price Prediction Using Linear and Polynomial Regression Models.* s.l., s.n.

OSF, n.d. *Consumer price index.* [Online] Available at:
http://www.stat.fi/til/khi/index_en.html [Accessed 1 May 2021].

Pandas, n.d. *Pandas.* [Online] Available at: https://pandas.pydata.org/ [Accessed 2021].

Ponomarev, E. S., Oseledets, I. V. & Cichocki, A. S., 2019. Using Reinforcement Learning in the Algorithmic Trading Problem. *Journal of Communications Technology and Electronics,* 64(12), p. 1450–1457.

Pring, M. J., 2014. *Technical Analysis Explained- The Successful Investor's Guide to Spotting Investment Trends and Turning Point.* New York: McGraw-Hill Education.

Qiu, M. & Song, Y., 2016. Predicting the Direction of Stock Market Index Movement Using an Optimized Artificial Neural Network Model. *PLOS One.*

Quang-Vinh, D., 2019. Reinforcement Learning in Stock Trading. *HAL Archives,* Oct.

Saito, S., Wenzhuo, Y., Shanmugamani, R. & Saito, S., 2018. *Python Reinforcement Learning Projects.* Birmingham: Packt Publishing Ltd..

Schulman, J. et al., 2015. *Trust Region Policy Optimization.* s.l., PMLR, pp. 1889-1897.

Schulman, J. et al., 2017. Proximal Policy Optimization Algorithms. *CoRR.*

Scikit Learn, n.d. *Scikit Learn API Reference.* [Online]
Available at: https://scikit-learn.org/stable/modules/classes.html

Sezer, O. B., Ozbayoglu, A. M. & Dogdu, E., 2017. An Artificial Neural Network-based Stock Trading System Using Technical Analysis and Big Data Framework. *Proceedings of the SouthEast Conference.*

Sharma, S., Sharma, S. & Athaiya, A., 2020. Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology,* 4(12), pp. 310-316.

Sutton, R. S. & Barto, A. G., 2018. *Reinforcement Learning: An Introduction.* Second ed. s.l.:The MIT Press.

Szepesvari, C., 2009. Algorithms for Reinforcement Learning. In: *Synthesis Lectures on Artificial Intelligence and Machine Learning.* s.l.:Morgan & Claypool Publishers.

Tickstory, n.d. *Tickstory.* [Online]
Available at: https://tickstory.com/
[Accessed 2021].

Wang, X., 2013. Theoretical Analysis of Financial Portfolio Model. *iBusiness,* Volume 5, pp. 69-73.

Watkins, C., 1989. *Learning From Delayed Rewards.* Cambridge, UK: King's College.

Xiong, Z. et al., 2018. *Practical Deep Reinforcement Learning Approach for Stock Trading.* s.l.:s.n.

Zeiler, M. D., 2012. ADADELTA: An Adaptive Learning Rate Method.

APPENDICES

Appendix A: Artificial Neural Network in Keras

```
def ann_model(state_size, action_size, hidden_dims=None,
              activation=None, activation_last_layer=None,
              loss=None, optimizer=None):
    model = Sequential()
    model.add(Dense(units=hidden_dims[0],
                    kernel_initializer='uniform',
                    activation=activation, input_dim=state_size,
                    name='input'))
    for i in range(1, len(hidden_dims)):
        model.add(Dense(units=hidden_dims[i],
                        kernel_initializer='uniform',
                        activation='relu', name=f'layer_{i}'))

    model.add(Dense(units=action_size,
                    kernel_initializer='uniform',
                    activation=activation_last_layer,
                    name='output'))

    metrics = None
    if isinstance(loss, str) and \
            (loss.lower() == 'categorical_crossentropy' or \
             loss.lower() == 'sparse_categorical_crossentropy'):
        metrics = ['accuracy']

    model.compile(loss=loss, optimizer=optimizer, metrics=metrics)

    return model
```

Appendix B: Convolutional Neural Network in Keras

```
def cnn_model(state_size, action_size, hidden_dims=None,
              batch_size=32, activation=None,
              activation_last_layer=None,
              loss=None, optimizer=None):

    model = Sequential()
    model.add(Conv1D(filters=int(batch_size/2), kernel_size=8,
                     input_shape=(state_size, 1),
                     kernel_initializer='uniform',
                     activation=activation, name='conv_input'))

    model.add(Activation(activation, name='activation'))
    model.add(MaxPool1D(pool_size=5, name='max_pooling'))
    model.add(Flatten())

    model.add(Dense(units=hidden_dims[0],
                    kernel_initializer='uniform',
                    name='hidden_in', activation=activation))
```

```
    for i in range(1, len(hidden_dims)):
        model.add(Dense(units=hidden_dims[i],
                        kernel_initializer='uniform',
                        activation=activation, name=f'layer_{i}'))

    model.add(Dense(units=action_size, kernel_initializer='uniform',
                    activation=activation_last_layer, name='output'))
    metrics = None
    if isinstance(loss, str) and \
            (loss.lower() == 'categorical_crossentropy' or \
             loss.lower() == 'sparse_categorical_crossentropy'):
        metrics = ['accuracy']

    model.compile(loss=loss, optimizer=optimizer, metrics=metrics)

    return model
```

Appendix C: Actor-Critic Networks in Keras

```
def get_network(input_size, hidden_dims=None, name='',
                activation=None):
    net_in = Input(shape=(input_size,), name=f'{name}_input_states')

    net = net_in
    for i in range(len(hidden_dims)):
        net = Dense(units=hidden_dims[i], kernel_regularizer=l2(1e-6),
                    name=f'{name}_dense_layer_{i + 1}')(net)
        net = BatchNormalization(name=f'{name}_batch_normalization_{i
+ 1}')(net)
        net = Activation(activation, name=f'{name}_activation_{i +
1}')(net)

    return net_in, net


def actor_model(state_size, action_size, hidden_dims=None,
                activation=None, loss=None, optimizer=None):
    states, net = get_network(state_size, hidden_dims, name='actor',
                              activation=activation)

    actions = Dense(units=action_size, activation=activation,
                    name='actor_actions')(net)
    model = Model(inputs=states, outputs=actions, name='actor')

    action_gradients = Input(shape=(action_size,),
                             name='action_gradient')
    loss = K.mean(-action_gradients * actions)

    if isinstance(optimizer, str) and \
       optimizer.lower() == 'adam':
        optimizer = Adam(lr=.00001)
    elif isinstance(optimizer, str) and \
         optimizer.lower() == 'nadam':
        optimizer = Nadam(lr=.00002)
    elif isinstance(optimizer, str) and \
         optimizer.lower() == 'adadelta':
        optimizer = Adadelta(lr=0.9)
```

```
    updates_op = optimizer.get_updates(params=model.trainable_weights,
                                       loss=loss)
    train_fn = K.function(
        inputs=[model.input, action_gradients, K.learning_phase()],
        outputs=[],
        updates=updates_op)

    return model, train_fn


def critic_model(state_size, action_size, hidden_dims=None,
                 activation=None, loss=None, optimizer=None):
    actions = Input(shape=(action_size,), name='critic_input_actions')

    states, net_states = get_network(state_size, hidden_dims[0:-1],
                                     name='critic',
                                     activation=activation)

    net_states = Dense(units=hidden_dims[-1],
                       kernel_regularizer=l2(1e-6),
                       name=f'states_dense_layer_'
                       f'{len(hidden_dims)}')(net_states)

    net_actions = Dense(units=hidden_dims[-1],
                        kernel_regularizer=l2(1e-6),
                        name='actions_dense_layer_1')(actions)

    net = Add(name='critic_add')([net_states, net_actions])
    net = Activation(activation, name='critic_activation')(net)

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

    model = Model(inputs=[states, actions],
                  outputs=q_values, name='critic_model')

    model.compile(optimizer=optimizer, loss=loss)

    action_gradients = K.gradients(q_values, actions)

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

    return model, get_action_gradients
```