APPLICATION OF STATE OF THE ART DEEP REINFORCEMENT LEARNING ALGORITHMS TO THE CRYPTOCURRENCY TRADING PROBLEM

DANIEL EKWUAZI

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF ENGINEERING
IN COMPUTATIONAL INTELLIGENCE SYSTEMS
SCHOOL OF ENGINEERING
KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG
YEAR 2023
KMITL-2023-EN-M-047-045

THESIS TITLE	Application of State of the Art Deep Reinforcement Learning Algorithms To The Cryptocurrency Trading Problem.
STUDENT NAME	Mr. Daniel Ekwuazi
STUDENT ID	61610024
DEGREE	Master of Engineering
PROGRAM	Computational Intelligence Systems
YEAR	2023
ADVISOR	Prof. Dr. Worapong Tangsrirat
CO-ADVISOR	Asst. Prof. Dr. Ratchanee Kulyanon

Abstract

The cryptocurrency market has been known to be volatile, which presents an opportunity to profit from swings. Although buy-and-hold agent has shown to be the simplest yet powerful trading strategy for a long-term period, yielding an outstanding ROI, there are efforts to automate the trading process. The most traditional approach is using statistical indicators to signal trading actions, sometimes assisted by a simple ML as a price predictor. Another new-age approach that is the topic of interest of this research is trading using reinforcement learning (RL), an application of ML which incorporates learning process by exploring states and searching the best action based on the states. A good action is granted a positive reward, vice-versa. In this research, multiple RL algorithms in continuous and discrete action spaces are used to perform backtrading in the Bitcoin spot market. Trained on a hour-level crab market, the models' performance are tested in bull, crab, and bear markets in hour and minute levels. The results show that RL agents acting on discrete spaces would give higher returns and relatively low MDD compared to continuous agents. The fastest RL training is attained by Discrete A2C, yet, it yielded benchmark-like results. This research contributes as the primary research in comparing different RL algorithms' performance in a backtrading environment based on algorithm and action space types. In future works, this research can be expanded by incorporating dynamic hyperparameter tuning and more complete RL design to enable the agent to act in a live trading environment.

Keywords: portfolio management, automated trading, reinforcement learning.

Acknowledgments

I appreciate all parties who have made the completion of this thesis possible. I would like to express my gratitude to my supervisor Prof. Dr. Worapong Tangsrirat and Asst. Prof. Dr. Rutchanee Gullayanon who with their patience has given me facilitation and guidance to allow me to finish this thesis. I would also thank Asst. Prof. Dr. Chaiwat Nuthong as my first, former supervisor who helped to get the ball rolling on my thesis. I also will not forget Asst. Prof. Dr. Ukrit. Watchareeruetai's teachings about machine learning and reinforcement learning which have deeply inspired me in conducting research in this field.

Furthermore, I am grateful of individuals who have produced excellent several teachings online that helped me reconstruct my thought flows, namely: Phil Tabor PhD. for his "Reinforcement Learning in Motion" masterclass and also Asst. Prof. Dr. Adam White and Asst. Prof. Dr. Martha White for their MOOC "Reinforcement Learning Specialization" course which helped me broaden my understanding of reinforcement learning and what materials to further read up on.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Cryptocurrency markets is often associated with price volatility. Its market cap has increased tremendously over the years to almost USD 3 trillion in late 2021[1]. Average investors are at risk of the fear of missing out (FOMO) and therefore prone to making investment decisions detrimental to their finances. When it comes to cryptocurrency, the market's popular action sentiment is to buy and hold. This has proven to be the best strategy in the long term, generating an astronomical return-of-investment (ROI), as can be seen with Bitcoin's return in 10 years[2]. Such tremendous return-of-investment ROI is exciting, but the buy and hold sentiment encourages that one perpetually holds yet without exit strategy; this makes investment returns to be suboptimal. Therefore, there is a need to equip a strategy-building framework to maximize returns, preferably with the assistance of automation.

Besides statistics and basic machine learning (ML) approaches, many switch to explore the field of Reinforcement Learning (RL) for trading automation: a ML branch which learns from rewards and punishments over exploring actions. Reinforcement Learning algorithms have achieved outstanding performances in various domains in recent history, with agents such as Alpha GO, Alpha Zero beating a human at the complex game of Go and Chess respectively. State of the art performance has also been achieved in Atari gameplays using Deep Q-Network (DQN) and in

[1] https://www.tradingview.com/symbols/CRYPTOCAP-TOTAL/
[2] https://www.tradingview.com/chart/?symbol=BTCUSD

robotics. Various DL algorithms have also been applied to stock and cryptocurrency trading bots.

1.2 Problem Description

Deciding when to buy, sell or hold, for investors who would like to take advantage of the cryptocurrency market swings and make gains in the short term, can be very challenging. Due to the high volatility and nonstationarity of cryptocurrency markets, sometimes traditional ML approaches do not perform as efficiently as wished. RL improves ML by being able to take actions based on real-time states and previous feedbacks, in automation. Despite RL's similarity to unsupervised ML in terms of learning supervision, RL is more efficient than ML due to the fact that the number of action an RL agent can take is bounded and more exact. Since there are a vast selection of RL algorithms to choose from, therefore, it is necessary to examine each algorithm's advantages and disadvantages by comparing their performances against another.

1.3 Research Objectives

This research aims to evaluate and compare several RL algorithms and action spaces (DQN, PPO, SAC, A2C, and A3C) and traditional trading algorithms (HODL, MACD) in performing a Bitcoin trading (sell, buy, hold). The comparison is based on the following key metrics: ROI, MDD, average ROI over benchmark's ROI, and time to convergence. On top of that, this research compares between continuous action space against discrete action space for the problem setting.

1.4 Research Scope

The scope of this research is as per the following:

- Trading decisions are done and calculated through backtesting.
- Trading data consists of training and testing data, comes from only one trading pair, which is `BTCUSD`:
  - Training data: 11,144 data points of a crab market in a hour-level timeframe.

  - Testing data: 2,000 data points in minute and hour levels, both in bull/crab and bear trends.

- The actions are limited to buying, holding, and selling Bitcoin in the spot market. No trading fees.

- The performance are measured using the following metrics: portfolio return, maximum drawdown (MDD), Sortino Ratio (deprecated), average ROI over benchmark's ROI, time to convergence, the number of trades taken, and the total trading value.

1.5 Contributions

This research serves as one of the first to scrutinize how different RL algorithms perform in the application of cryptocurrency trading, according to the type of the algorithm (policy-based, value-based, and combined) and the type of action space (discrete, continuous). From the results, this work will summarize the advantages and disadvantages of each algorithm in the application of trading in the spot market, and suggest future researchers on how to pick models based on specific cases.

Chapter 2

Literature Review

2.1 Introduction

Before looking into several researches which compares between reinforcement learning (RL) algorithms' advantages and disadvantages, preliminary lectures regarding RL implementations in trading are presented, to illustrate RL implementation in trading applications.

The first practical lecture [4] aims to design an RL agent to perform spot trading. Here, the process starts with deciding how the data will be read and learned by an agent: either by reading the whole dataset, per-trade data chunks, per-window candles, or per-instrument. The next step is designing rewards. Reward and punishment design is deemed to be the core problem in trading and can be measured by different factors, e.g., profit and loss (PnL) on exit, per-period PnL, trend detection, and long hold prevention. It is also important to choose the features to extract from the data, for example, technical indicators, sentiment data; using different time resolutions are recommended. From there, the system is ready to be tested, using several possible instruments like trend curves, random walks, autocorrelation, etc. Lastly, a deep learning algorithm (see section 3.3.1 is added to the system to improve learning capacity.

Results from this lecture show that RL may require huge amount of sample and is prone to overfitting, therefore results can vary among agents despite using the same code. Moreover, designing reward function is difficult: even with the assistance of trading indicators, it is difficult to avoid sticking to local optima.

Another supporting lecture [5] performs cryptocurrencies trading with the following qualities

to measure: return-on-investment (ROI) and performance over market benchmarks. This research compares RL training in stationary and non-stationary data. Stationary data means that value distribution (mean, variance) at any time point is constant: one time point corresponds to one sample. Non-stationary data is the opposite: a series of data is considered as one individual sample. Since asset prices are non-stationary, the work mentions that conventional statistics and machine learning are not necessarily enough to tackle this problem. Therefore, the work uses REINFORCE, a policy gradient RL method that estimates the optimal policy by adjusting the policy that maximizes total reward. However, this algorithm is remarkably not sample efficient: old data are thrown away. Moreover, network parameters are prone to violently change, that explains the huge variability in training results.

2.2 Deployment of Deep Reinforcement Learning and Market Sentiment Aware Strategies in Automated Stock Market Prediction

Sagiraju and Mogalla [1] compare several RL algorithms in predicting stock market in an automatic manner using both market price data and Twitter sentiment as the environment. The proposed framework is as shown in Fig. 2.1.

Various algorithms are compared: Deep Q-Networks (DQN), Deep Deterministic Policy Gradient (DDPG), Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C), and their proposed AutoEncoder-LSTM network combined with Reinforcement Learning (AE-LSTM+RL).

The daily closing data of Dow Jones Industrial Average (DJIA) and S&P 500 data from 2006–2016 are used as training data, and are tested against the real data from 2016–2021. These data are combined with tweet sentiments in the range of [-1,1] calculated from engagement data. Yet, to align with the focus of this thesis, sentiment in this research is not discussed here.

Three benchmarks are used in evaluating the algorithms' performance:

1. Sharpe Ratio: $\frac{R_P - R_B}{\sigma_P}$, where R_P = portfolio return, R_B = benchmark rate, and σ_P = standard deviation of portfolio. Penalizes any return below benchmark. Higher is better.

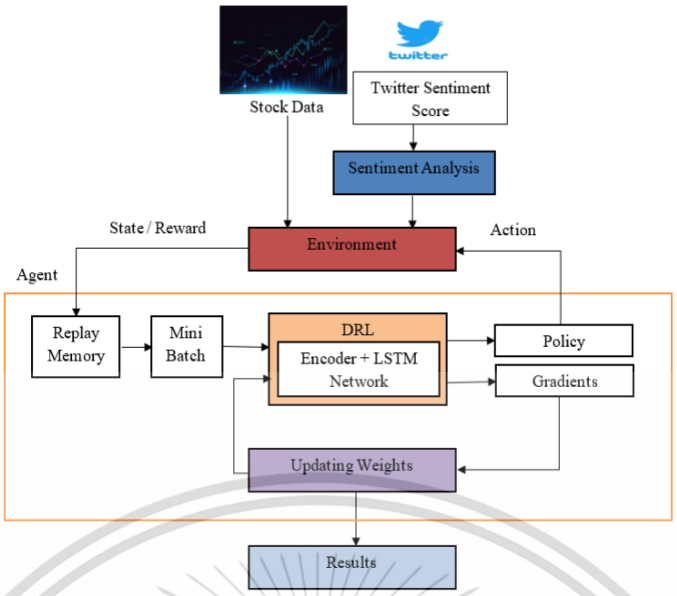

Figure 2.1: Proposed framework by [1]

2. Sortino Ratio: $\frac{R_P - R_B}{\sigma_D}$, where R_P and R_B are identical to above, and σ_D = standard deviation of downside. Punishes returns below benchmark within a specific threshold. Higher is better.

3. Maximum Drawdown (MDD): the maximum percentage of drawdown from the latest portfolio all-time high (ATH). Measures downside risk. Values closer to zero is better.

4. Annual and cumulative portfolio return (ROI). Higher is better.

The results show that, from the two assets' evaluation, AE-LSTM+RL returns the highest Sharpe Ratio and almost the best Sortino Ratio, annual ROI, and cumulative ROI, yet it causes the highest MDD. A2C performs slightly better or slightly worse to AE-LSTM+RL. PPO's results are often the worst in many benchmarks except it performs best at MDD. DQN and DDPG show similar results to A2C. Results can differ depending on the tested asset or time period, for example, PPO gives the highest annual DJIA return but by far the lowest cumulative return. Hence, supporting [5]'s claim regarding the high variability between results given by RL tests, comparing and deciding which conventional algorithm performs the best is not a simple task.

2.3 Deep Reinforcement Learning for Trading

Similar to [1] but excluding sentiments data and with different assets, Zhang et al. [6] compare DQN, Policy Gradients (PG), A2C, with baseline algorithms Long Only, Sign(R), and Moving Average Convergence Divergence (MACD) using nine different metrics to maximize trading portfolio of five asset categories: commodity, equity index, fixed income, foreign exchange, and combined. From the results, DQN yields the best results in most metrics in every investment class except equity index. PG and A2C comparably perform worse than DQN but better than baseline algorithms, with A2C performs better than PG more often.

2.4 A2C versus A3C

Espeholt et al. [7] presents one comparison between A2C and A3C as a part of a larger comparison comprising of a proposed novel actor-critic algorithm, batched A2C, and A3C. The focus of the research is not related to trading but to find the most efficient algorithm to learn to complete two puzzles, measured by the training speed and learning outcome. The comparison between A2C and A3C is only found in the single-machine, GPU-less experiment, where A3C with 32 workers and 64 CPUs can process puzzle 1 and puzzle 2 at 6.5K and 9K frames per second (FPS). Meanwhile, batched A2C, 48 CPUs, with step synchronization, can reach 9K and 5K FPS, and batched A2C with trajectory synchronization yields 16K and 17.5K FPS. Again, this comparison is done using different settings and not mentioned in other parts of the study, hence cannot be easily concluded if A2C has any advantage over A3C.

Therefore, a qualitative comparison by Sewak [8] between A2C and A3C will be included here. A3C is performed by distributing learning to multiple agents to be done in parallel. Each newly created agent copies the parameters from a centralized network parameter server and trains its data over the parameters. At one time, there will be a merge event where agents combine their own parameters with the global parameter, then replaces each's own parameters with the new global parameter to retrain. However, A3C's asynchronousness can be disadvantageous: agents may have different knowledge of the global state since not every agent pull the global parameters at one time. This is particularly bad if an agent fails or refuses to synchronize with the global parameters for

a long time, causing training instability and slower convergence. A2C improves A3C by forcing agents to synchronize at specific times or conditions together at one time. Though, if an agent is unreachable or not responsive, this may cause synchronization delay.

2.5 Verdict

An extensive search of research repositories show that there has not been many, if any, research papers that explicitly compare between conventional algorithms, ML algorithms, and RL algorithms or between A2C against Asynchronous Advantage Actor-Critic (A3C), especially in the applications of trading and portfolio management. Moreover, the comparative results from different existing, or even within the same research, may wildly vary. With similar experiment settings and comparisons, the results can be different, see [1] which shows that A2C yields the better results versus [6] which clearly signifies that DQN is superior.

Chapter 3

Background Knowledge

This chapter introduces concepts and algorithms that are used in this research.

3.1 Genetic Algorithm

Before discussing about genetic algorithms, we visit its hypernym: evolutionary algorithms. Evolutionary algorithms mimic the biological evolution that living beings use to maintain the existence of their species [9]: how each generation's lifetime starts and ends and how optimized it performs its "tasks," compared to other individuals and its parents [10]. In evolutary algorithms, "organisms" or a model would naturally survive if it has a stronger gene, for example, when it can perform more optimal problem solving capability [11]. The keyword "gene" here introduces us to the concept of "genetic algorithms."

Genetic algorithms train individual models in a generation to solve a problem: the outcome being trained models and their respective properties (genes), forming a "gene pool." Each "genotype", consists of a set of genes, is represented by a binary string. Only those with higher fitness value can survive and can reproduce in the next training generation. For each iteration, genes from different qualified individuals can recombine with each other or mutate, just like biological gene crossover or mutation [11, 12]. The generational trainings come into when a stop condition has been met.

A classic pseudocode of genetic algorithms is as follows.

```
obtain initial population;
calculate each individual's quality;
while not completed do                                  // produce new generation
    for population size/2 do                            // recombination cycle
        choose two individuals to recombine, prioritizing higher quality candidates;
        recombine the selected two to produce two offspring;
        calculate each offspring's quality;
        append the produced individuals into the new generation/iteration;
    end
    if population count converges then
        completed ← TRUE
    end
end
```

Algorithm 1: A Classic Process of Genetic Algorithms [13]

3.2 Artificial Neural Networks (ANN)

Artificial Neural Networks (ANN), or in short neural nets (NN), is an interconnected layers of "neurons" that act as processing elements. Connection strength between neurons, which denotes the processing ability of the network, is represented by "weights" [2]. ANN is based on the working of living creatures' nervous systems, especially the brain. As concisely aforementioned, a simple NN consists of layers, neurons, and weights, see Fig. 3.1. In its simplest form, an NN consists of a single input layer and a single output layer. An input layer can have one or multiple neurons with different values x_k representing the inputted data to the neural network. A weight w_k connects an input neuron w_k and an output neuron y and acts as a multiplier, i.e. the output neuron y will receive the value $w_k x_k$ from this connection. A neuron can be a source of multiple weights to the next layer and also a destination of multiple weights from the previous layer. The value of a output layer neuron in Fig. 3.1 is equal to $y = f(\sum_{i=0}^{n-1} w_i x_i - \theta)$, where f is an activation function, e.g. Sigmoid, normalization, etc. and θ an internal threshold or offset. Lastly, the output layer, after receiving its final value, becomes the output of the NN. An output layer is useful for classification problem: it shows which neurons have a value that lies within a specific classification threshold.

The correctness or quality of an ANN can be determined using a loss function, which indicates the disparity between the expected output and the prediction produced by the ANN. Loss function may also be used to adjust other hyperparameters, such as weights [14].

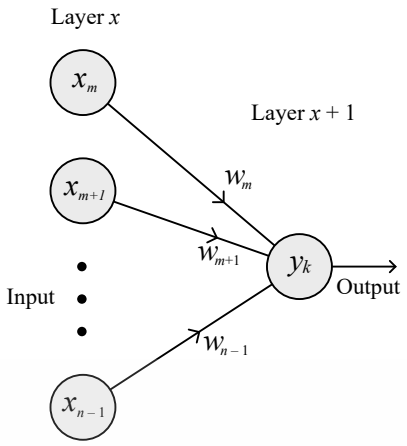

Figure 3.1: Structure of an ANN [2]

3.2.1 Deep Learning

Deep learning improves ANN in a way that it is multi-layered: each layer processes different levels of features. Deep learning algorithms have intermediary layers connecting input and output layers which in a progression extract useful input patterns and pass them to the subsequent layer [15]. According to how inputs are presented or obtained, deep learning can be divided into multiple types, most prominently (1) unsupervised learning, (2) supervised learning, (3) reinforcement learning (RL), or (4) combination between any of (1), (2), or (3).

Supervised learning problems has the following characteristics. First, data is presented in the form of $X \mapsto Y$, where X is a representation data labeled with a label Y. Next, the learning agent will abstractly learn the relations between X and Y as a knowledge base to predict the label of a new input. Unsupervised learning lacks Y part such that the learning agent needs to infer data classification from the feature distribution of input data. Agents in reinforcement learning learns independently, assisted from rewards and punishments from performing an action in one environment state.

3.3 Analytic Trading

Trading, in this sense, refers to an activity of buying, selling, or holding a commodity, stock, currency, or other derivative assets in their corresponding finacial markets. The ultimate goal of

trading is to gain profit from price difference and minimizing transactional cost. Analytic trading optimizes profit-making decisions by predicting price actions based on fundamental analysis plus chart features such as price trend, support and resistance levels, trading volume, trading indicators, price patterns, etc. [16]. Fig. 3.2 shows Bitstamp's Bitcoin-USD pair chart taken from TradingView on 5 August 2022, at 9:01 AM UTC. It contains different indicators: Volume (Vol), Moving average with window size equals 28 bars (MA(28)), MACD long/short strategy, and Bollinger Bands with window size 20 and band deviation of 2 sigma (BB(20,2)).

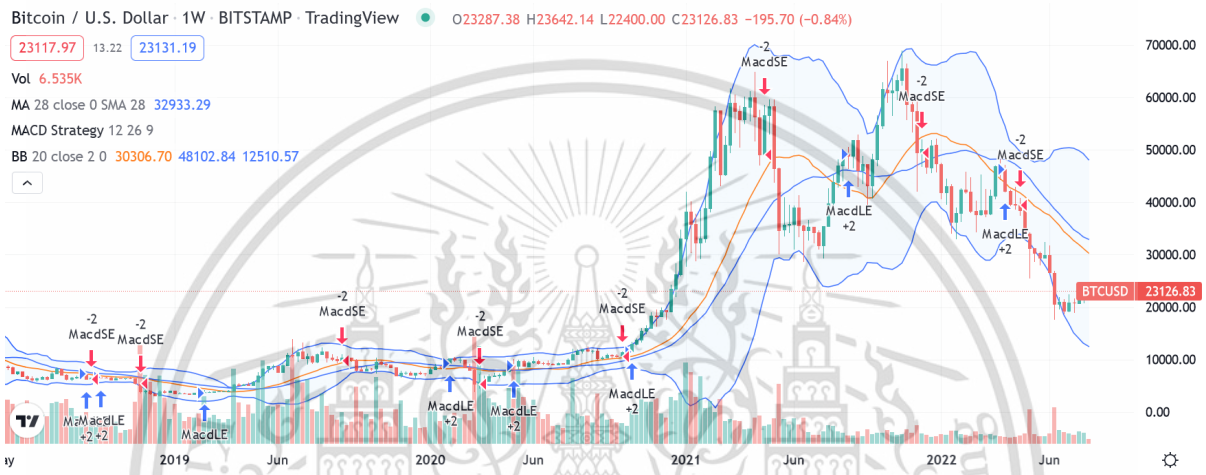

Figure 3.2: An example of trading chart with several indicators [3]

There are three important vocabularies that can describe the trend of an asset price: bear, bull, and crab [17]. A ""bear" market defines a downward price trend; a "bull market" defines an upward price trend. Finally, the term "crab" market is used to describe a relatively stagnant, trendless market, alternatively, a price movement with both major bull and bear markets contained in it. Consult Figure 3.3.

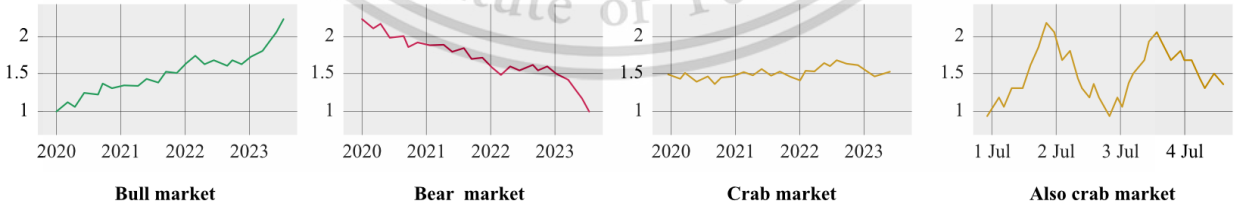

Figure 3.3: Bear, bull, and crab markets

Typically, the input data of a price prediction problem is in a form of a time series data [18]. Values from a time window is then calculated and analyzed to obtain various indicators to assist

trading decision. More recent approaches automate trading decisions in a form of algorithmic (algo) trading, which can generate decisions in (near-)real time. Sometimes, algo trading is empowered by artificial intelligence and ANN to improve the prediction performance as well [19].

Many fund managers nowadays have shifted their interest to trade cryptocurrencies due to their profit-inducing volatility [20] and rapid development [21]. Bitcoin, being the first invented and the most mature cryptocurrency, has become the cryptocurrency standard in various researches and studies [20].

There are three basic actions in spot trading: `Buy`(x) or `Long`, `Sell`(x) or `Short`, and `Hold`. `Buy`(x) tells the trading bot to buy x units of asset at the specified time, `Sell`(x) works similarly but for selling, and `Hold` is to neither buy nor sell the asset at the specified time point. Decision to choose one of the three depends on indicators, or in ANN-enhanced decision, the output of the ANN.

This research uses two popular indicators: exponential moving average (EMA) and Moving Average Convergence/Divergence (MACD), a derivative of EMA. $EMA(x)$ is defined as

$$EMA(x) = p \cdot k + EMA_p \cdot (1-k), \tag{3.1}$$

where $n = \frac{2}{x+1}$, x is the number of time frames considered in the EMA, p is the asset price at the current timeframe, and EMA_p is the value of EMA of the previous day [22]. Since the function is recursive, the base case of the formula, for the first datapoint, is [23]

$$EMA_0(x) = \frac{\sum_{i=0}^{t-1} p_i(1-\alpha)^i}{\sum_{i=0}^{t-1}(1-\alpha)^i}, \tag{3.2}$$

where α is the discounting factor of past values, typically is 0.18.

MACD, on the other hand, uses EMA in its calculation, and is defined as: $MACD(f,s) = EMA(f) - EMA(s)$. f stands for “fast" moving average, typically is fixed at 12. s is “slow" moving average, typically is equal to 26. On top of that, $EMA(l)$ of $MACD$ is called the “signal line", equals to $EMA(l) - MACD(f,s)$ [23]. Signal line (hereafter notated by σ) is useful to trigger buy and sell signals. When $MACD - \sigma$ turns from negative to positive, a buy signal is issued; when it turns from positive to negative, a sell signal is issued. Otherwise, no trade is done.

3.3.1 Deep Learning for Trading

Long-Short Term Memory (LSTM), a subset of Recurrent Neural Network (RNN), is a popular supervised deep learning algorithm for trading bots since its nature fits with predicting future values from past data. This class of RNN uses either sigmoid or hyperbolic tangent (tanh) activation function and SoftMax. Back propagation in RNN, when seeing a non-optimal result, may redirect the outputs of the network to the hidden, intermediary layer, in a hope to improve the prediction result [24].

LSTM improved long-term forecasting by adding more memory into RNN. Each LSTM 'cell' (Fig. 3.4 [25]) has two states: hidden state (h, to process input and output and short-term memory) and cell state (c, for data flow and long-term memory). There are several steps in building an LSTM network [26]. First, forgetting unused information from previous step (h_{t_1}) and input X_t using a sigmoid function (σ) through the forget gate (f_t, a vector whose values are between 0 and 1). The next stage is remembering new information from X_t through the memory gate (i_t to accept or discard new information multiplied by $\tilde{C}_t$ to give an importance level to the input). Finally, the output of the cell h_t is obtained by multiplying $\sigma(h_{t_1})$ and c_t.

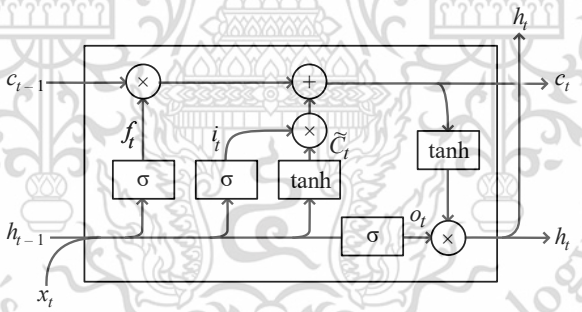

Figure 3.4: An LSTM cell

LSTM, in this regard, however, is used mostly for forecasting stock prices. Trading decisions are generated using another deep learning algorithm named Convolutional Neural Network (CNN). CNN, usually used for feature extraction in multidimensional spaces like images, learns stock movement patterns (trend, trajectory) and thus can decide when to buy or sell an asset [27]. Not many works indicate the ROI, but prediction accuracy. Yet, Chakole and Kurhekar's work [27] show that a plain CNN would return a lower ROI than other methods, not excluding the buy-and-hold method.

3.4 Markov Decision Process (MDP)

Markov Decision Process is central to the main concept of this research Reinforcement Learning (RL). MDP mimics the working of decisions in human life, focuses on choosing the better impact from decisions. MDP can be modeled after a finite state machine like in Fig. 3.5. An agent can choose to go from a state to another state (or back to its current state) via an action and receive a reward for each state propagation. In a more general definition, an agent can act upon the environment to move to another state and receive a reward as a consequence [28].

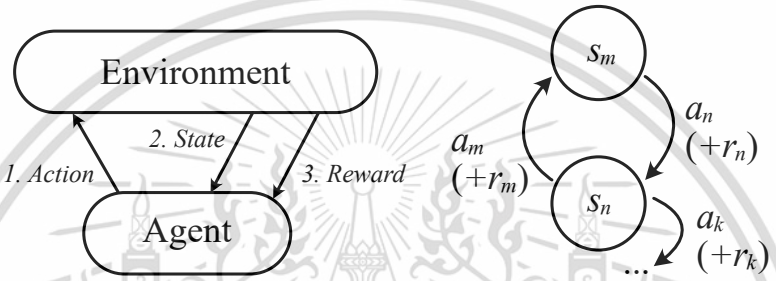

Figure 3.5: Markov Decision Process in two views.

The formulation of MDP is as follows [28]:

$$r_t(s,a) = \sum_{s' \in S} r_t(s,a,s') p_t(s'|s,a), \tag{3.3}$$

where r denotes reward, t is the current propagation instance, $r_t(x)$ is the reward of performing a propagation x, s is the current state, a indicates the action taken, s' is the next state after performing a on s, and $p(x)$ is the probability of propagation with condition x happening.

According to Feinberg and Schwartz [29], there are two kinds of impacts: (1) cost minimization and profit maximalization and (2) impacts on future states after the decision. MDP looks for a decision that is profitable in long-term period by choosing optimal policies. MDP is shown to be useful in various implementations. In finance and investment, [30] uses MDP to decide when to sell or hold a financial asset to get faster gain. Saario [31] tries to solve a house-selling problem, that is, someone must decide to sell a house given an arbitrary offer or decline the offer and wait longer for a better offer. An MDP-like dynamic programming (DP) solution is used to maximize the accepted offer value.

3.4.1 Bellman Equation and Optimality

Bellman equation plays an important role in dynamic programming: it expects the total reward from the present state s to the end of the propagation horizon in an MDP. The equation [28] is as follows:

$$v_t^{\pi}(s) = \max_{a \in A(s)} \left\{ r_t(s,a) + \sum_{s' \in S} \gamma^t p(s'|s,a) v_{t+1}(s') \right\}, \tag{3.4}$$

where $v_t^{\pi}(s)$ is the expected present and future total rewards from the current state s. A discounting factor γ^t, $0 < \gamma < 1$ is sometimes added to diminish the effect of far-future rewards and to put priority to near-future rewards.

The best and most optimal possible $v_t^{\pi}(s_0)$ is denoted by $\max_{\pi} v^{\pi}(s) = v*(s)$. Bellman's Optimality Theorem states that [32] $v^*(s)$ must satisfy the Bellman Optimality Equation

$$v^*(s) = \max_a \left\{ r(s,a) + \gamma \sum_{s'} p(s'|s,a) v^*(s') \right\} \tag{3.5}$$

for all s.

3.5 Reinforcement Learning (RL)

Reinforcement Learning (RL), a branch of deep learning, is analogous to how living things learn to perceive and to interact with their environment: by doing something experimentally then receiving a reward or a punishment as a consequence [33]. Like so, RL iteratively learns causes and effects under little supervision to reach the goal of obtaining as much reward as possible, in a computational way.

Based on finite MDP, RL can be illustrated using Fig. 3.5 [33]. Five elements of RL are :

1. Agent: the entity which learns about the environment by taking actions and receiving rewards,

2. Environment: every entity other than the agent whom the agent interacts with,

3. State S_t: the current state or constellation of the environment,

4. Action A_t: a set of possible tasks or actions that are allowed to be taken by an agent at the current state to change the state of the environment, and

5. Reward R_t: the consequence that the agent receives upon completing an action.

3.5.1 Policy in RL

The output of RL is a policy π_t for each time step, that is, $\pi_t(a|s)$, a probability of choosing an action $A_t = a$ at the current state $S_t = s$. The objective is to produce a set of policies π_t for every A_t, S_t which maximizes R_t in the long run. In an equation [34],

$$\pi^* := \underset{x}{\operatorname{argmax}}\, \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_k \middle| \pi\right], \tag{3.6}$$

where π^* is a deterministic optimal policy, $\mathbb{E}[\cdot|\pi]$ is expectation based on policy π, γ is discount factor, which determines agent's consideration of possible future actions. π itself describes the policy under a state-action trajectory $s_0, a_0, s_1, a_1, ..., a_{k-1}, s_k$.

3.5.2 RL Taxonomy

The following list adopted from [35] shows the taxonomy of RL algorithms. Some of the sub-branches and algorithms will be discussed in the subsequent paragraphs.

1. Model-Based RL
   (a) Model given, e.g., `MCTS` (Monte Carlo tree search)
   (b) Model learned, e.g., `I2A` (Imagination-Augmented Agents), World Model
2. Model-Free RL
   (a) Value-Based: focus on state-action value
      i. On-Policy: policy determines learning
         - `Sarsa`
      ii. Off-Policy: learning is done randomly
         - `Q-Learning`

- DQN (Deep Q-Network), further classified into C51, Dueling DQN, Double DQN, QT-Opt, and DDPG (Deep Deterministic Policy Gradient).

(b) Policy-Based: focus on overall policy value

i. Gradient-free

- Cross-Entropy Method: QT-Opt (intertwined with DQN)
- Evolution Strategy: SAMUEL

ii. Gradient-based

- Actor-Critic
  - DDPG (Deep Deterministic Policy Gradient)
  - DDPG combined with DQN
    - SAC (Soft Actor Critic)
    - TD3 (Twin Delayed DDPG)
  - A2C (Advantage Actor Critic)
  - A3C (Asynchronous Advantage Actor Critic)
- Policy Gradient
- ACKTR (Actor Critic using Kronecker-Factored Trust Region)
- TRPO/PPO (Trust Region Policy Optimization/Proximal Policy Optimization)

Model-based and Model-free Reinforcement Learning

A model in RL refers to any data or knowledge that contains the state-action-reward prediction table for an agent to plan and follow. Model-based RL uses a model, learned or known, and approximates a policy function from learning [36]. Model-free RL, on the other hand, calculates the environment spontaneously using trial-and-error, while retaining no transition probability distribution arrays, hence learning how to act solely based on rewards [37]. Model-free algorithms are viewed to be labor-intensive but can adapt to new environment fast, while model-based algorithms are more lightweight but produce less optimal policies, hence are more suitable for deterministic use cases.

Actor-Critic Method

Konda and Tsitsiklis [38] observe that most RL algorithms are either actor-only or critic-only. Actor-only methods focus on parametrization of policies and parameter improvisation without learning process. An example of actor-only methods is REINFORCE. Critic-only methods use solely value approximation in learning to comply with the Bellman equation, but produce less optimal policy. An example of critic-only methods is Q-learning.

Thus, the actor-critic method is derived to fuse the advantages of actor-only and critic-only algorithms. Actor-critic mathods typically use two separate neural networks: a neural network for actor and a neural network for critic [39]. The critic part approximates and simulates learning to improve the actor segment's policy parameters.

Policy Gradient

The policy gradient theorem often plays a big role in Actor-critic algorithms [35]. However, Sutton et al. [40] claim that sole function approximation is not effective enough in realizing a policy convergence. One effort is to parametrize the policy instead of computing the policy from parametrized action-value functions. This way, one can track and plan the gradient descent of the overall policy parameter. Say that the policy parameter is denoted by θ, the gradient $\Delta\theta = \alpha\hat{\nabla}_\theta J(\theta)$ can be adjusted by setting α: step size and $J(\theta)$: RL performance of the policy that follows the parameter θ, either calculated by total reward or reward per step.

Gradient Descent Gradient descent itself is a method that iteratively minimize a function value. Imagine a function $y = f(x_1, x_2, ..., x_n)$. Gradient descent starts with choosing a random or predefined parameter vector $x_1, x_2, ..., x_n$ as the initial position. Obtain the gradient vector by observing the change of y when each of $x_1, x_2, ..., x_n$ is changed by a little. Next, adjust some x using a predefined step to get y with smaller value and repeat until a local minima for y is found. This process can be illustrated by Fig. 3.6.

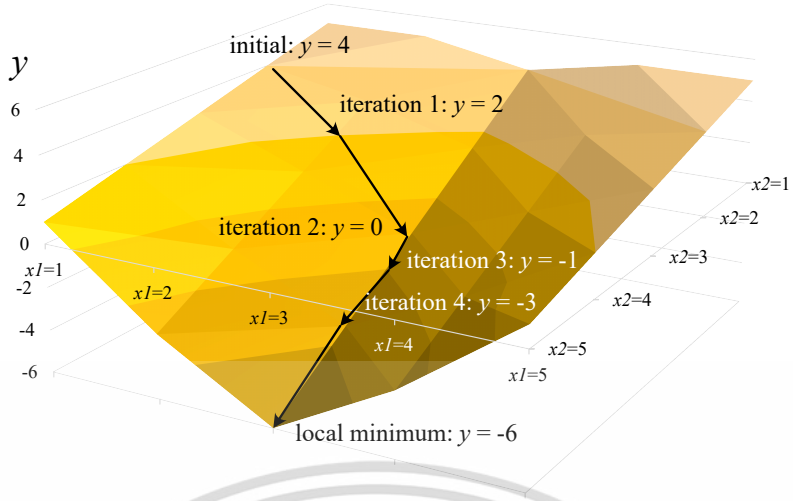

Figure 3.6: Gradient descent

Q-learning

Q-learning, falls under the model-free RL algorithms family, explores the maximum reward of all possible state-action propagations, updated for every propagation. The Q-function $Q(s,a)$ is defined as $r(s,a)+\gamma v^*(\delta(s,a)) = r(s,a)+\gamma \max_{a'} Q(\delta(s,a),a')$, where δ is a state transition function. Therefore, in Q-learning notation, $\pi^*(s) = \text{argmax}_\pi Q(s,a)$. The maximum current rewards of state-action pairs are recorded in an $n-$dimensional array, and for each iteration, Q-learning will choose the action with the highest reward, based on the current state. The algorithm is as follows [33]:

$Q(s,a), \forall s \in S, a \in A(s) \leftarrow$ an arbitrary value;
$Q(terminal\ state, \cdot) = 0$;
while *S is not terminal state* **do**
  S gets initial value;
  for *each propagation* **do**
    Based on π constructed in Q, select possible A for the current S;
    Perform A, get R and S';
    $Q(S,A) \leftarrow Q(S,A)+\alpha[R+\gamma \max_a Q(S',a)-Q(S,A)]\ S \leftarrow S'$
  end
end

Algorithm 2: Q-learning Algorithm

Q-learning with Deep Learning However, in a system with huge dimensions of A and S, calculating Q value for all S,A pairs consumes heavy resources which is unsustainable and inefficient. Mnih et al. [41] apply Deep Learning into Q-learning, called a deep Q-network (DQN), to convert high-dimensionality input into a quantized, pattern-matched inputs with lower dimensionality. The work gives an example scenario: an RL agent needs to observe a video game with 84 $\times$ 84 $\times$ 4 consecutive images $\times$ 256 grayscale levels and to act upon the state by "clicking" on one of many controller buttons. With a plain array-based Q-learning, each state change require the system to recursively calculate $256^{84\times84\times4} \times 18 \approx 10^{69971}$ Q values, assuming that there are at most 18 different actions that the agent can take. That value exceeds the number of atoms in the universe, which is $\approx 10^{82}$. With DQN, state-action-reward mappings are approximated using convolutional neural network on-the-go and per-request. This way, determining action would not require storing or generating much, and sometimes unnecessary, prior knowledge.

3.5.3 This Research's RL Algorithms in A Nutshell

Alongside with DQN which is introduced in Section 3.5.2, four RL algorithms are used: Proximal Policy Optimization (PPO), Advantage Actor Critic (A2C), Asynchronous Advantage Actor Critic (A3C), and Soft Actor-Critic (SAC).

Asynchronous Advantage Actor Critic (A3C) and Advantage Actor Critic (A2C)

A3C came from a research by Mnih et al. [42] that proposes an asynchronous deep RL framework which adds asynchronous, parallel actor-learners over One-step Q-learning, n-step Q-learning, Sarsa, and A2C. The framework is designed such that multiple RL actor-learners are run in different CPU threads, updating a centralized parameter server. The work finds that parallel workers expand the exploration horizon, hence maximizes knowledge procurement and thus learning speed. Among the asynchronized versions of the four base algorithms, asynchronous A2C (thus called A3C) is the fastest agent that reaches the higher score when tested on Atari 2600 games.

A2C, derived in the same work, is a version of A3C with only one agent. In short, A2C is an improvement of vanilla actor-critic method with an addition of *advantage* value: how advantageous it is to perform an action a_t compared to the average of all possible actions at the state s_t [43].

Proximal Policy Optimization (PPO)

OpenAI's Schulman et al. [44] derived a set of related algorithms called the Proximal Policy Optimization (PPO) algorithms, a subset of policy-based RL. PPO improves Trust Region Policy Optimization (TRPO) in terms of implementation, sample complexity, and execution time. For one eposide of policy update, PPO goes through multiple or batched small episodes of stochastic gradient policy ascent. Moreover, PPO focuses on the area of policy gradient change with high probability, to increase policy faster and more efficiently. Tested using benchmark RL tasks (`Hopper-v1`, `Walker2d-v1`, `HalfCheetah-v1`, etc.), PPO is claimed and shown to perform better than A2C, CEM, TRPO, A2C with Trust Region, and vanilla Policy Gradient method.

Soft Actor-Critic (SAC)

An example actor-critic algorithm that combines policy-based and value-based RL is Soft Actor-Critic (SAC) [45]. SAC optimizes policy function using an off-policy way by mimicking Q-Learning's approach to accomplish a task: by acting balancedly random (maximizing entropy) yet also enhancing expected reward. The original work [45] indicates that SAC performs better on benchmark RL tasks (`Hopper-v1`, `Walker2d-v1`, `Ant-v1`, etc.) compared to prior works such as TD3, DDPG, PPO, and Soft Q-Learning (SQL).

Chapter 4

Methodology

This research considers ten trading agent types to be compared:

- Buy-and-hold (`HODL`) method, as the benchmark or reference agent,
- MACD indicator method, as a representative of a traditional indicator-based agent,
- DQN, a value-based off-policy method with a discrete action space,
- PPO (discrete and continuous), a pure policy gradient-based method,
- SAC, a policy gradient-based method fused with DQN, with a continuous action space,
- A2C (discrete and continuous), another pure policy gradient-based method, and
- A3C (discrete and continuous), to be compared with A2C.

4.1 Dataset

All training and testing data `BTC` data in term of `USD`. A sole training dataset $\mathscr{D}_{train}$ is fetched from CryptoDataDownload [46], containing 11,144 per-hour data points from to 14 October 2020 22:00 UTC to 22 January 2022 05:00 UTC. The price started at `USD` 11401.81 per `BTC` and ended at `USD` 35467.32 per `BTC`. This date range is selected because it is a crab market with two bull markets and two bear markets. The graph is given in Fig. 4.1: the black line (left y-axis) shows the price and the red line (right y-axis) shows the hourly return.

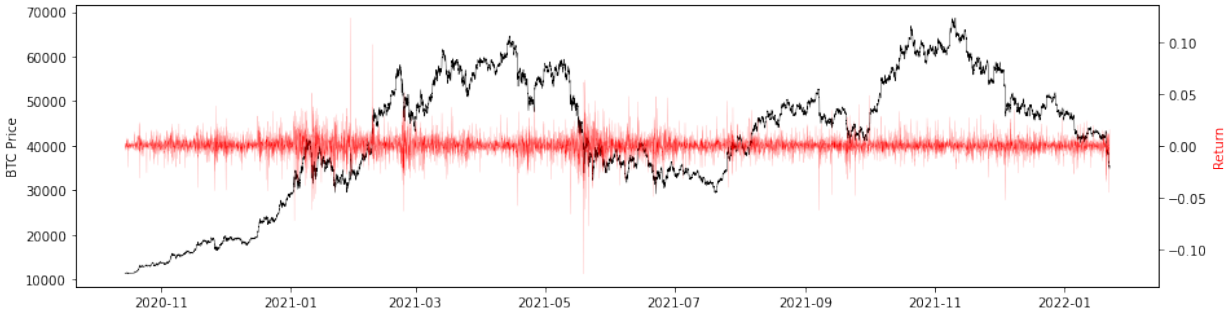

Figure 4.1: Training data graph

Test datasets $\mathscr{D}_{test}$ are provided by Cryptocompare[1] API, consists of two sets of timeframes. There are four datasets in use:

- Each-hour bull data (`HBu`): 2,000 points of data from 9 November 2022 06:00 UTC to 31 January 2023 14:00 UTC inclusive (Fig. 4.2). The price started at `USD` 18292.63 per `BTC` and ended at `USD` 23128.09 per `BTC`.
- Each-hour bear data (`HBr`): 2,000 points of data from 9 November 2021 06:00 UTC to 31 January 2022 14:00 UTC inclusive (Fig. 4.3). The price started at `USD` 68054.30 per `BTC` and ended at `USD` 37537.04 per `BTC`.
- Each-minute crab data (`MCr`): 2,000 points of data from 16 February 2023 04:40 UTC to 17 February 2023 14:00 UTC inclusive (Fig. 4.4). The price started at `USD` 24689.73 per `BTC` and ended at `USD` 23796.90 per `BTC`.
- Each-minute bull data (`MBu`): 2,000 points of data from 14 February 2023 04:40 UTC to 15 February 2023 14:00 UTC inclusive (Fig. 4.5). The price started at `USD` 21740.88 per `BTC` and ended at `USD` 22699.29 per `BTC`.

In each graph, like in the training data, the black line (left y-axis) shows the BTC price and the red line (right y-axis) shows the one-timeframe return. A small note for `MCr`: although the graph visually looks like a bear market, the graph contains extended sideways movement, a bull movement (around 16 February, 16:00), and a bear movement (around 17 February, 00:00).

[1] https://www.cryptocompare.com/

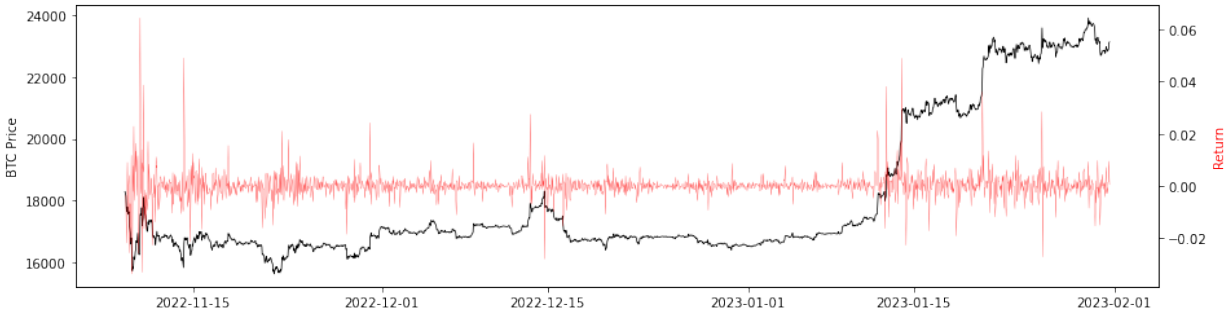

Figure 4.2: Each-hour bull test data

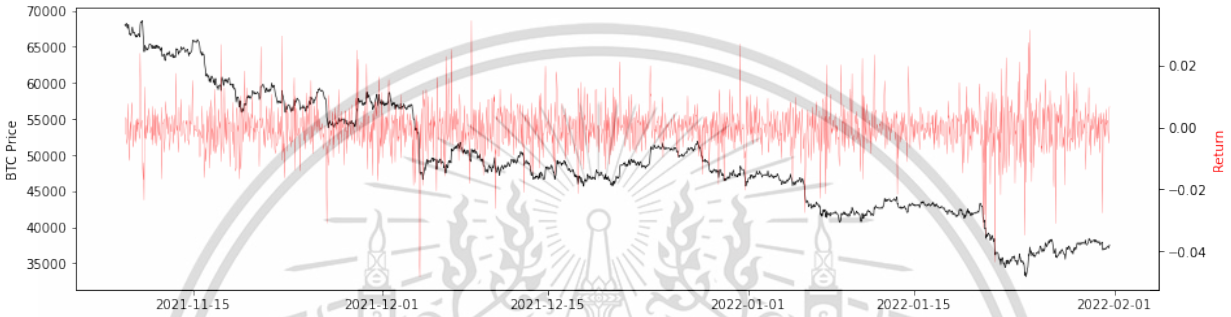

Figure 4.3: Each-hour bear test data

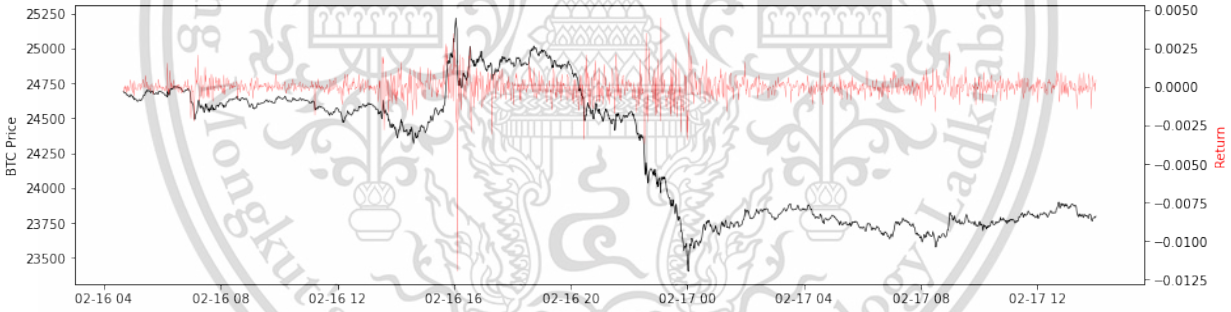

Figure 4.4: Each-minute crab test data

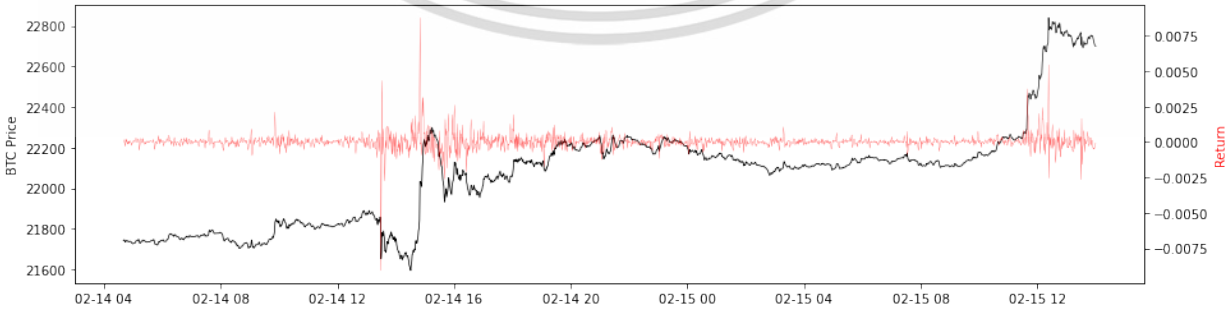

Figure 4.5: Each-minute bull test data

Each data point contains five information:

- `timeframe`: the time point where the price movement occured,
- `high`: the highest price logged in that timeframe,
- `low`: the lowest price logged in that timeframe,
- `open`: the price logged at the start of that timeframe,
- `close`: the price logged at the end of that timeframe.

The datasets go through a preprocessing step that adds three more custom columns to the original data, namely:

- `ewm`: the value of $EMA(28)$ of that timeframe,
- `macd_histo`: the values of $MACD(26,12,9) - SignalLine$ of that timeframe,
- `return`: the price movement since the last data point's close.

The reasons behind having completely independent train and test datasets are as follows:

- Avoids overfitting and introduces generalization. A market movement is not always chronically random: a price action at a timeframe can still retain causality with previous n timeframes.
- Trains the agent to see and examine what to do in all trends (bear, bull, and crab). The training dataset contains all of these trends, forming a huge crab market.
- Examines how the all-trend trained agent performs in each individual market directions. This is useful to know if it can perform well in which trends. An agent which generate profit in all trends would be performing well no matter what future trend will happen.

4.2 Problem Representation

Given a dataset $\mathscr{D}$ and an initial portfolio (ϕ) value of USD 10,000. At each data point d_t at timeframe t, a trading agent must decide to perform an action: either BUY to buy an asset, SELL to sell an asset, or HOLD to do no trade.

An example trading process is as follows. Suppose you have an initial capital ϕ_0 of USD 10,000 and the price of BTC is USD 20,000 per unit. You can perform a BUY trade, for example, using the entire portfolio, making your portfolio to be USD 0 and BTC 0.50000000, valued at USD 0 + BTC 0.50000000 $\times$ USD 20,000 per texttttBTC = USD 10,000. Imagine that the price of BTC rose to USD 25,000 per BTC and you decided to SELL half of your BTC holdings (0.25000000 BTC), getting you USD 6,250. Then, you would have a portfolio consisting of USD 6,250 and BTC 0.25000000, valued at USD 6,250 + BTC 0.25000000 $\times$ USD 25,000 per BTC = USD 12,500, a 25% profit (= +0.25 return). Lastly, on the next time period BTC price fell down to USD 15,000 per BTC and you decide to HOLD. Your portfolio, still at USD 6,250 and BTC 0.25000000, is now worth USD 6,250 + BTC 0.25000000 $\times$ USD 15,000 per BTC = USD 10,000, back to ϕ_0, giving the overall return of 0% (= +0 return).

Since the setting is a spot trading environment, the agent is not allowed to buy more asset than its current unspent capital and is not allowed to sell more than the value of Bitcoin it holds. All trades are executed without transaction fees, following Binance's move[2]. Finally, all agents are compared against another using the factors specified in Section 4.3.

4.3 Performance Metrics

Three performance metrics are used for final agent evaluation and also act as factors that determine reward[3]:

- The average of agent's portfolio return over benchmark's, denoted by δ. Let $\phi_\alpha(t)$ denote the portfolio return of agent α and $\phi_\odot(t)$ portfolio return of the benchmark agent. δ is

$$\frac{1}{T}\sum_{t=0}^{T}(\phi_\alpha(t)-\phi_\odot(t))$$

[2]https://www.binance.com/en/support/announcement/binance-launches-zero-fee-bitcoin-trading-10435147c55d4a40b64fcbf43cb46329

[3]however, see 4.5.4

, where T is the number of data points in the testing dataset, in this case, 2,000. Larger δ is better,

- Maximum drawdown at the final timeframe T (MDD_T): ranges from 0 to 1, closer to zero is better, and
- Non-annualized 100-timeframe Sortino ratio at the final timeframe T, larger is better.

To compare between agents, in addition to the above factors, two extra factors are included:

- Training speed (s), calculates the approximate time it takes for the training reward to converge in seconds, smaller is better. The number of episodes at the time point may follow.
- The number of trades taken (buy and sell). This factor is useful for future references, in researches that include trading fees.

4.4 Non-RL Agents

4.4.1 Buy-and-Hold Agent

The buy-and-hold agent is the bare basic agent as a global control variable, usually called as the “benchmark" in algo trading. The strategy is simple: buy at the first timeframe and sell at the last timeframe. There is no learning or calculation involved. Directly using testing datasets, the portfolio return (as a ratio) of this agent is equal to $\frac{p_{2000}-p_0}{p_0}$, in other words, the price movement relative to the initial price. An agent can be called a profitable agent if and only if it can perform better than this benchmark.

4.4.2 MACD Strategy Agent

The MACD strategy agent follows the signal from `macd_histo` to decide when to buy, when to hold, and when to sell. This agent acts as a second benchmark, representing simple statistical trading agents. The agent can only buy or sell the asset using all its capital: no order splitting is performed here. Therefore, directly using the four testing datasets, the portfolio return of the

MACD strategy agent is $\frac{\phi_{2000}}{\phi_0}$, where $\phi_t = \phi(USD)_t + rate_t \times \phi(BTC)_t$: the total portfolio in both USD and BTC.

4.5 RL Agents

This section describes the design of the trading agent: action space, state space, and the reward function. There are eight types of agents that share the same RL agent class, namely: (Discrete) DQN, Discrete A2C, Continuous A2C, Discrete PPO, Continuous PPO, (Continuous) SAC, Discrete A3C, and Continuous A3C.

4.5.1 Agent Characteristics

Table 4.1 shows the characteristics and hyperparameters used for each algorithm.

Table 4.1: Agent Characteristics and Hyperparameters

	DQN	**PPO**	**A2C**	**SAC**	**A3C**
Discount factor (γ)	0.95				
Learning rate	0.0001	0.0003	0.0007	0.0003	0.0001
Network type	2-layer fully-connected CNN				
Actor-Critic net	-	shared		separate	shared
Neurons per layer	64	64	64	256	64
Number of episodes	200	200	150	150	300
Action space	discrete	discrete, continuous	discrete, continuous	continuous	discrete, continuous
Library	Stable-Baselines3				N/A
Other factors	Exploration: initial rate = 1, decay rate = 0.1, final rate = 0.05	clip range = 0.2			2 threads

4.5.2 Action Space

There are two types of action state: continuous and discrete.

Continuous Action Space Trading with continuous action space allows the bot to perform a trade only using a certain amount of portfolio (in portion), e.g., buy BTC using 15% of available balance, etc. The continuous action space is a two-dimensional array consisting of `action_type` and `action_percentage`. The range of `action_type` is [0,3), where [0,1) triggers the `BUY` signal, [1,2) for `SELL` signal, else `HOLD`. And `action_percentage`, ranging at [0,1], indicates the amount of portfolio to trade (buy/sell) with. `action_percentage` has no effect on `HOLD` action.

There are some additional constraints to make: to reduce the number of buys and sells, the agent can only perform a `BUY` when the amount to buy is more than 0.001 BTC and there is at least USD 100 to spend. Moreover, the agent can only `SELL` when the trade size is over 0.001 BTC and the amount held is more than 0.0001 BTC.

Discrete Action Space Trading with discrete action space allows the bot to perform a trade only using the whole portfolio balance. The discrete action space is a scalar with three possible values: 0 for `BUY`, 1 for `SELL`, and 2 for `HOLD`. This mode also employs the exact same constraints like in the continuous action space.

4.5.3 State Space

The observable state space is in the continuous space, consists of the past five values of `return`, which is a stationarized data. An alternative case would be including the movement of `macd_histo` and `ewm`, but after several observations, the final reward would perform randomly, poorly, or never converge. Refer to Figs. 4.6 as a reference.

4.5.4 Reward Function

The reward function for both discrete and continuous action spaces is described using Algorithm 3.

There is no reward for buying. Multiplied by the holding amount, holding costs 0.0001 parts of profit or loss added with a flat -0.0001 points of reward to avoid lazy trading: a condition where the agent becomes hesitant to trade to avoid loss, plus a small amount of reduction when the unrealized profit is in the negative zone. When selling, two kinds of rewards are added together.

Figure 4.6: Rewards and returns (PPO) when using state space with three values. The declining pink line in `return_`, corresponds to the upper pink line in `reward`, appeared when Sortino is involved in reward calculation. The x-axis indicates the number of timesteps of learning.

Input: action type
Output: reward
if $action = BUY$ **then**
  reward $\leftarrow 0$;
else
  if $action = SELL$ **then**
    $reward\_profit \leftarrow$
     $sold\_amount \times (sell\_price - average\_held\_price)/average\_held\_price$;
    $reward\_mdd \leftarrow (latest\_mdd - current\_mdd) \times 0.5$;
    $reward \leftarrow reward\_profit + reward\_mdd$;
  else
    $reward\_profit \leftarrow hold\_amount \times (0.0001 * (net\_worth -$
     $initial\_balance)/initial\_balance) - 0.0001$;
  end
end
return *reward*

Algorithm 3: Reward Calculation Algorithm

The first component is profit reward, which is the difference between the selling price and the average holding price relative to the average holding price. The last component is how much MDD increases after the selling action times 0.5 to give it less priority than profit reward. The two components are multiplied by the selling amount.

Like determining state space and also mentioned in Fig. 4.6, Sortino was originally included in the reward calculation, but was removed due to poor performance.

4.5.5 Training And Testing Process

Each agent is trained 10 times through the training dataset $\mathscr{D}_{train}$. Since each training would likely to return distinct results, one which visually produces the highest return would be handpicked to be the representative of the algorithm. From each agent, a final model would be taken to be tested with all test datasets (`HBr`, `HBu`, `MCr`, `MBu`).

Chapter 5

Experimentation and Results

5.1 Development Environment

In the experiment, all cases share an identical set development environment:

- Programming environment: Python 3.9 (Google Colab)
- Deep learning library: PyTorch 1.13.1 (CPU)
- RL libraries: `Stable-Baselines3` for DQN, PPO, SAC, and A2C, code from scratch for A3C
- Processor: one single-core, double-threaded Xeon Processor @2.3 GHz (Google Colab CPU)

5.2 Results

As a rule of thumb, the term 'portfolio,' denoted by the red lines in the graphs, shows the portfolio value progression that is generated by the corresponding agent. The benchmark or `BTC` price movement is shown in black.

5.2.1 Buy-and-Hold Agent

The buy-and-hold agent's performance exactly follows BTC's price movement and is regarded as the performance benchmark. The statistics are shown in Table 5.1.

Table 5.1: Buy-and-hold agent statistics

	Return	**MDD**	**Sortino**	δ	**Figure**
HBu	0.2643	0.1174	0.7060	0	5.1
HBr	-0.4484	0.5199	-0.3982	0	5.2
MCr	-0.0361	0.0725	-0.2276	0	5.3
MBu	0.0441	0.0164	1.0024	0	5.4

Figures 5.1, 5.2, 5.3, and 5.4 show the full per-timeframe statistics of the recorded metrics. The upper part of the graphs shows two lines: (1) Red line: BTC closing price of each timeframe relative to the closing price of the first timeframe, (2) Blue line: maximum drawdown. The lower part of the graphs indicates the 100-timeframe Sortino ratio of each timeframe.

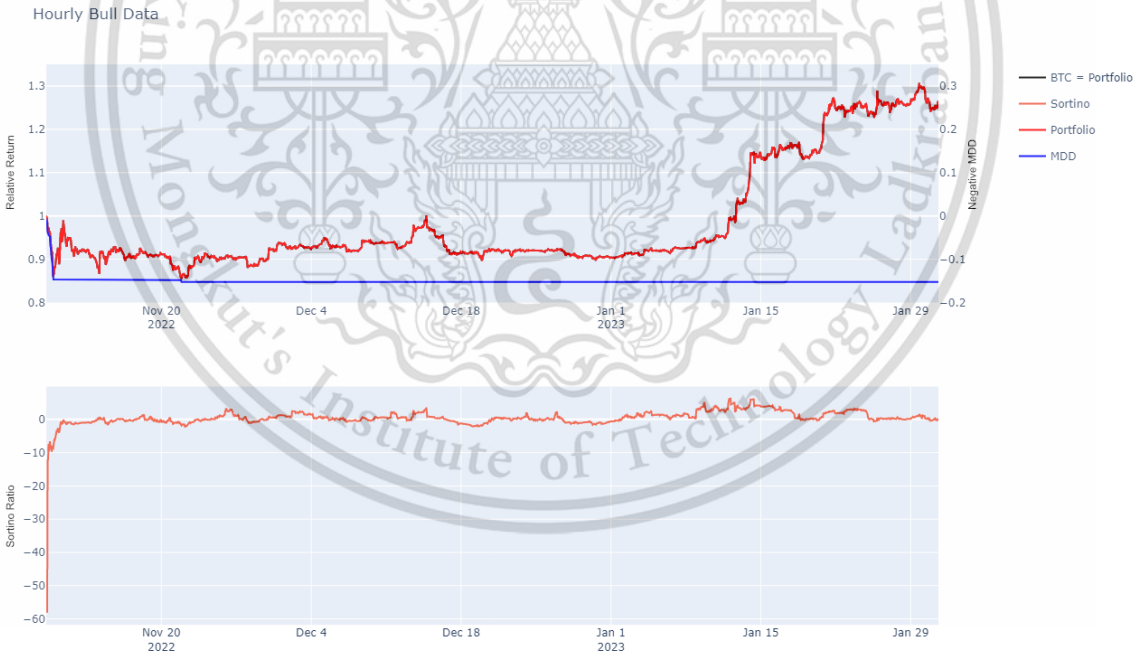

Figure 5.1: Buy-and-hold metrics for hourly bull data

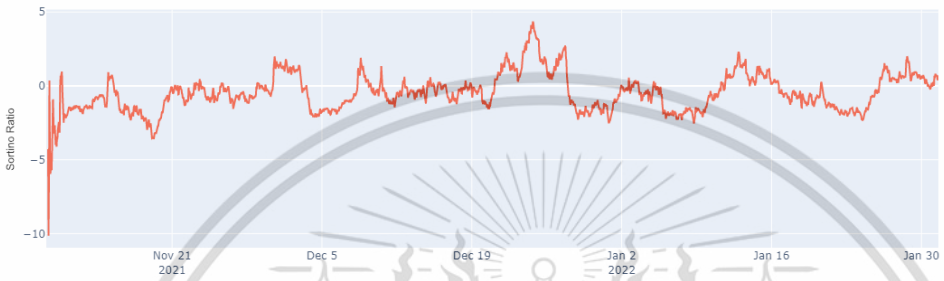

Figure 5.2: Buy-and-hold metrics for hourly bear data

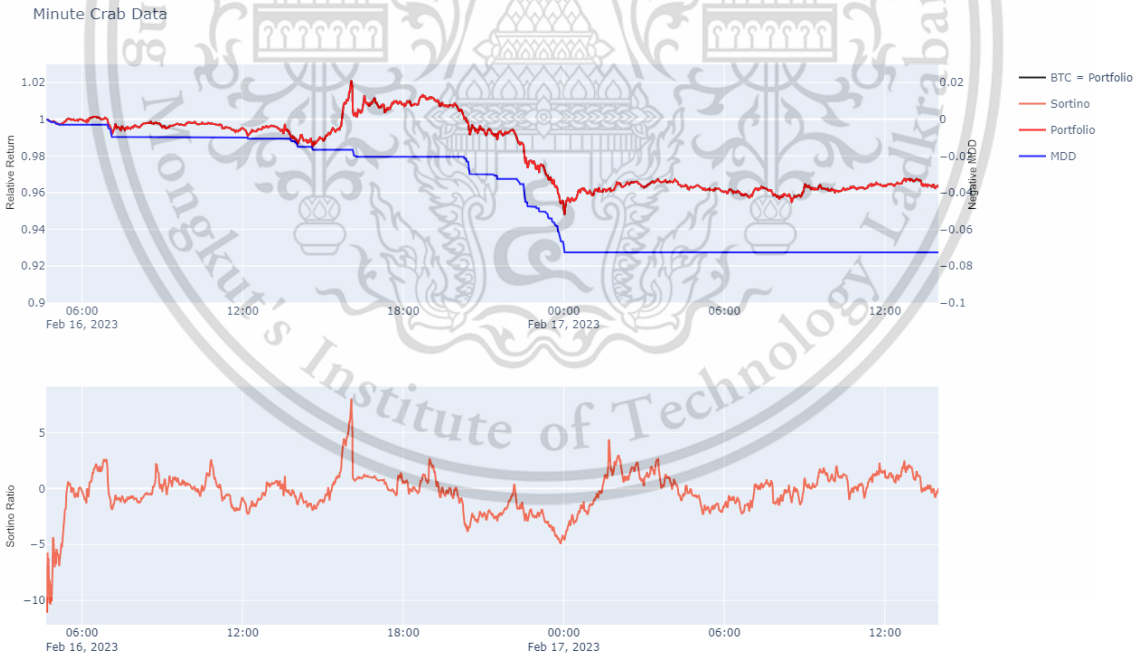

Figure 5.3: Buy-and-hold metrics for minute crab data

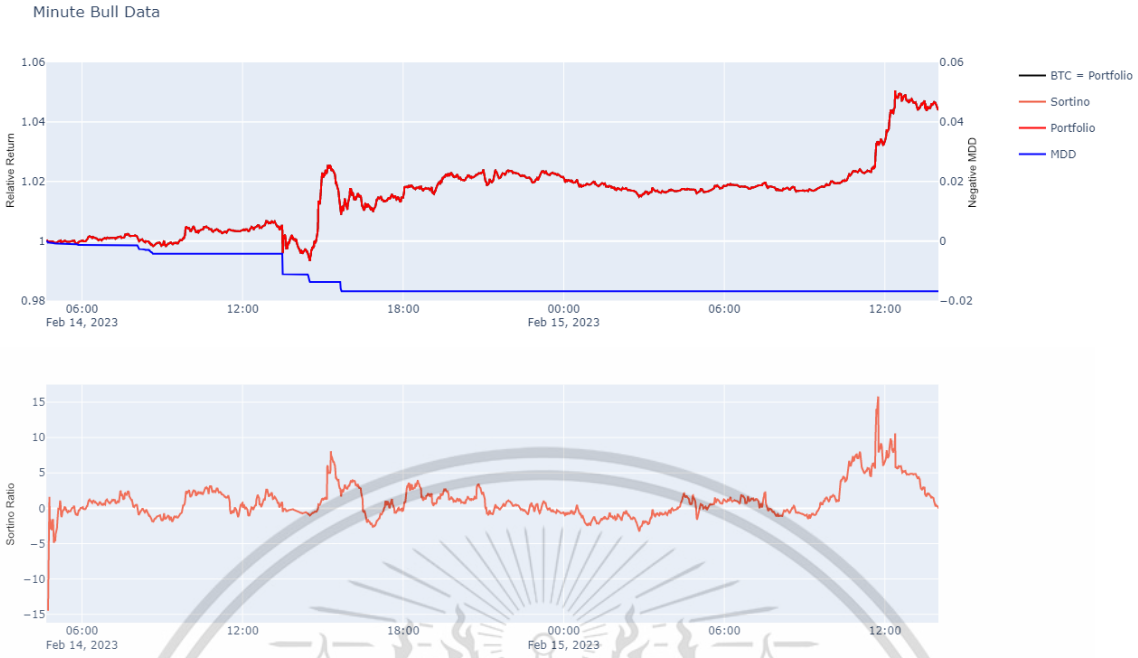

Figure 5.4: Buy-and-hold metrics for minute bull data

Since the buy-and-hold agent only buys at the first timestamp and sells at the last timestamp, then for all cases, the number of BUY is 1 with an trade value and an average of $initial\_capital/d_0$. That also applies to SELL. See Table 5.2.

Table 5.2: Trading actions statistics for buy-and-hold agent. Value and average trade values in BTC.

	BUY			SELL			HOLD
	trades	**value**	**average trade**	**trades**	**value**	**average trade**	
HBu	1	0.54666824	0.54666824	1	0.54666824	0.54666824	1998
HBr	1	0.14694213	0.14694213	1	0.14694213	0.14694213	1998
MCr	1	0.40502670	0.40502670	1	0.40502670	0.40502670	1998
MBu	1	0.45996298	0.45996298	1	0.45996298	0.45996298	1998

5.2.2 MACD Strategy Agent

The MACD strategy agent's performance statistics are shown in Table 5.3. Since this agent is not a learning agent, the values of s, τ, and ι remain undefined.

Table 5.3: MACD strategy agent statistics

$\mathscr{D}$	Agent			Benchmark			δ	Fig.
	Return	**MDD**	***Sortino***	**Return**	**MDD**	***Sortino***		
HBu	0.2161	**0.1159**	*0.0421*	**0.2643**	0.1174	*0.7060*	0.0620	5.5
HBr	**-0.3389**	**0.3570**	*0.6351*	-0.4484	0.5199	*-0.3982*	0.1363	5.6
MCr	**-0.0148**	**0.0427**	*0.4893*	-0.0361	0.0725	*-0.2276*	0.0118	5.7
MBu	**0.0364**	**0.0124**	*-0.1592*	0.0441	0.0164	*1.0024*	-0.0008	5.8

Observe that this agent dampens the drawdown and loss in downtrend markets (figs. 5.6, 5.7), yet it also limits the profits for some cases and some periods in bull markets (figs. 5.5, 5.8). Moreover, MACD strategy shows that it can reduce MDD in all four cases. As for MACD strategy's Sortino ratio, it performs better than benchmark in bear markets, but never in bull or crab markets.

Figures 5.6, 5.5, 5.7, and 5.8 show the full per-timeframe statistics of the recorded metrics. The upper part of the graphs shows three lines: (1) Black line: BTC closing price of each timeframe relative to the closing price of the first timeframe, (2) Red line: portfolio value of each timeframe relative to the initial investment, (3) Blue line: maximum drawdown of the portfolio. The lower part of the graphs contains two information: (1) Green line: 100-timeframe Sortino ratio of each timeframe, (2) Bar chart: shows the value of `macd_histo` at each timeframe.

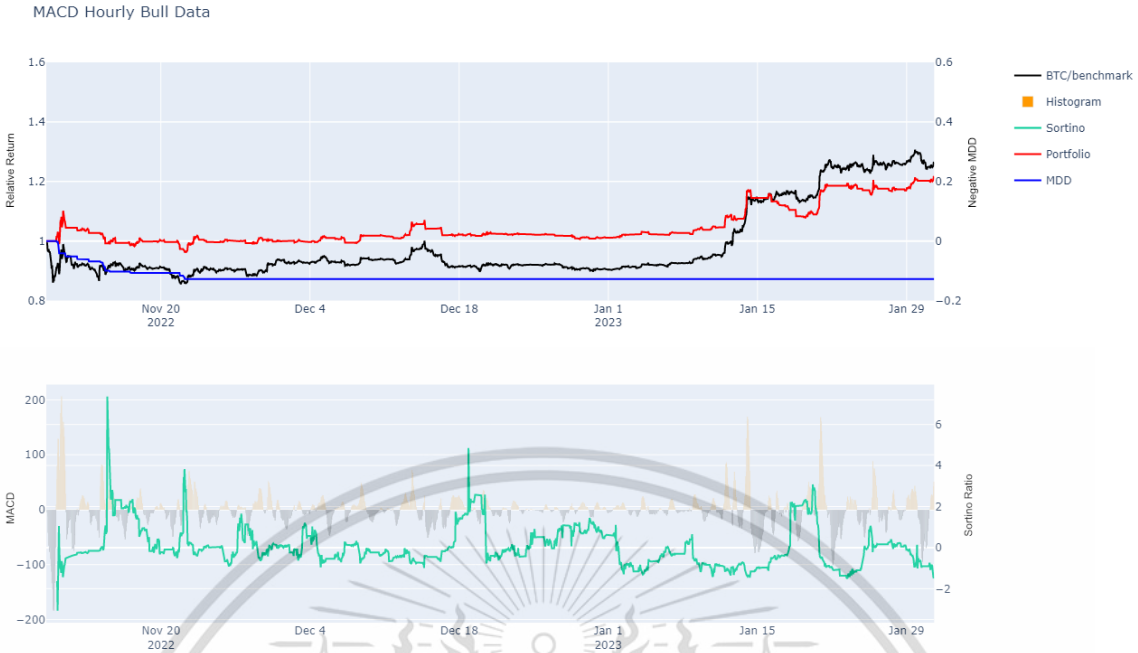

Figure 5.5: MACD strategy metrics for hourly bull data

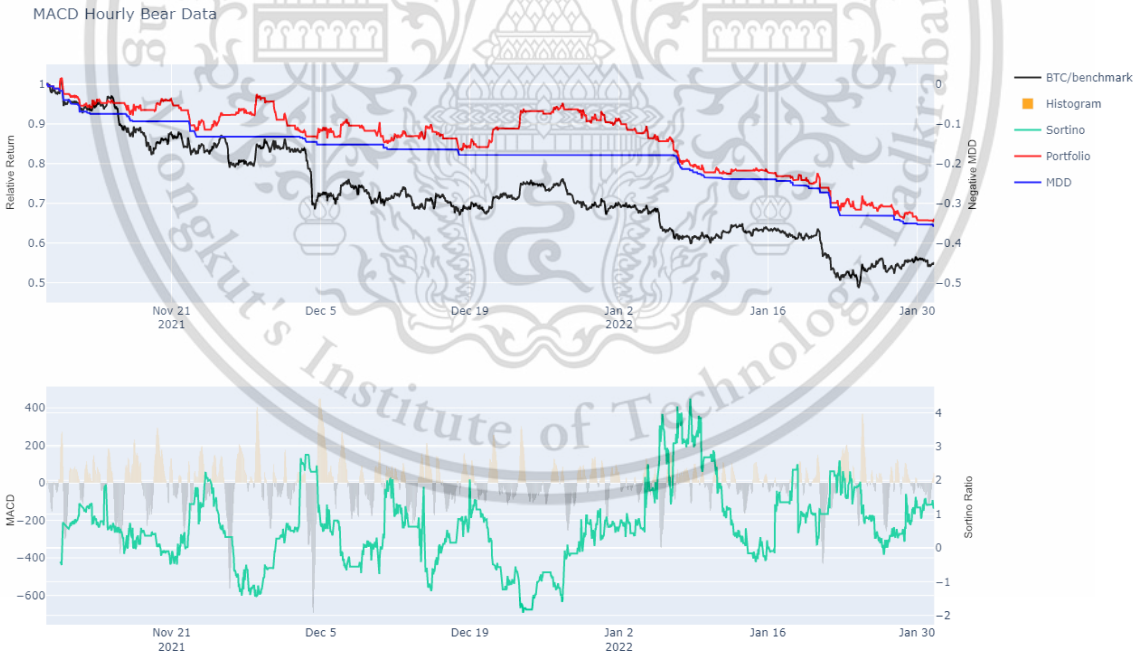

Figure 5.6: MACD strategy metrics for hourly bear data

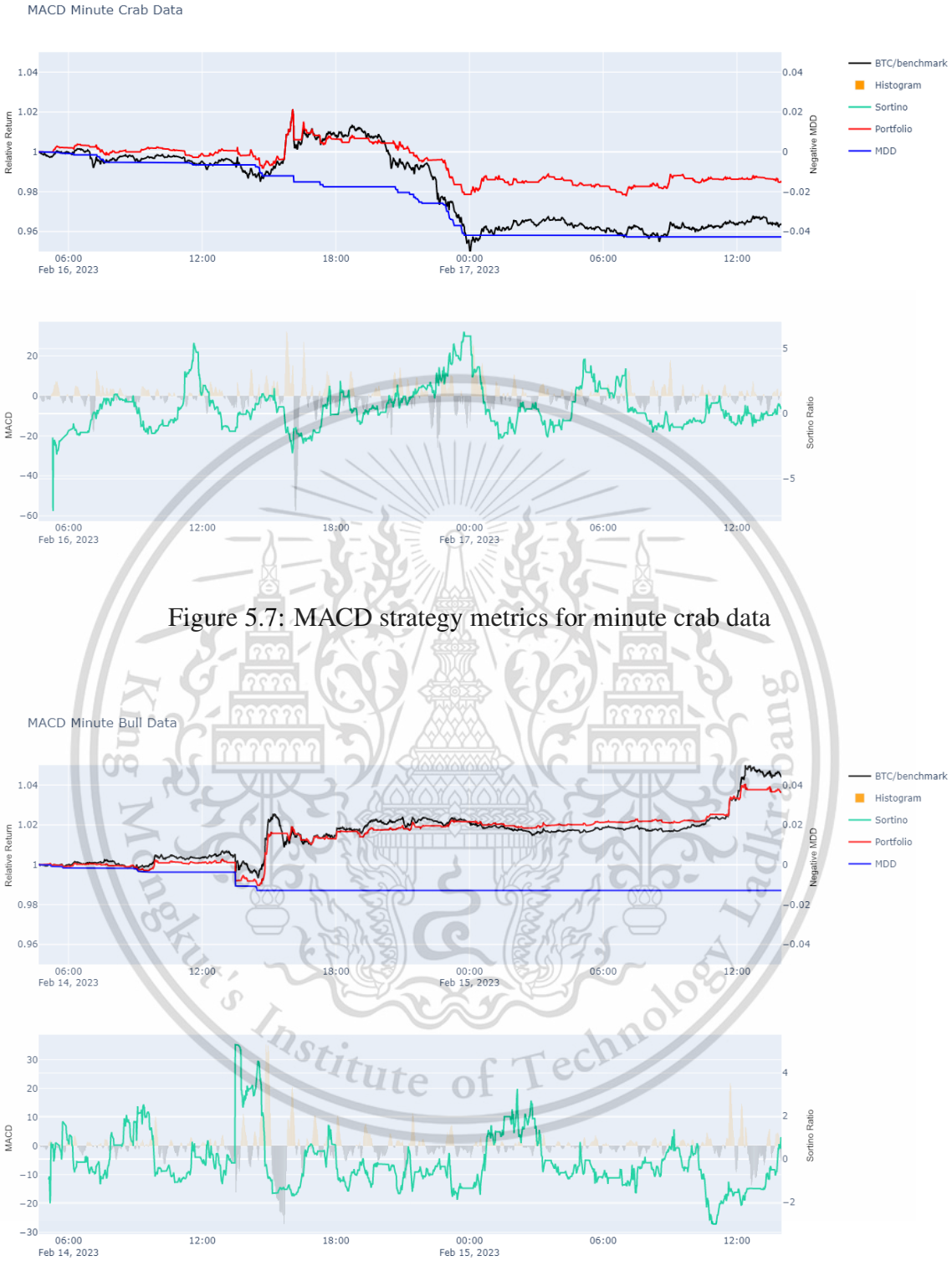

Figure 5.7: MACD strategy metrics for minute crab data

Figure 5.8: MACD strategy metrics for minute bull data

Table 5.4 shows that for each run, the MACD agent does exactly the same amount of BUYs and

SELLs, since it is set as a discrete agent. The MACD agent performs only a few trades, which is quite ideal for cases where trading fee is not free.

Table 5.4: Trading actions statistics for MACD agent. Value and average trade values in BTC.

	BUY			SELL			HOLD
	trades	**value**	**average trade**	**trades**	**value**	**average trade**	
HBu	69	40.06502231	0.58065249	69	40.06502231	0.58065249	1868
HBr	80	14.08360883	0.17604511	80	14.08360883	0.17604511	1846
MCr	75	30.74781048	0.40997080	75	30.74781048	0.40997080	1856
MBu	74	34.00521997	0.45952999	74	34.00521997	0.45952999	1858

5.2.3 DQN Strategy Agent (Discrete)

The DQN agent is the first RL agent to be presented in this chapter, operating in the discrete action space. Using training data (see Fig. 5.9), the agent performs best only at early timestamps (between 100,000 to 500,000 timesteps $\approx$ 9 to 45 episodes). After around $s = 770{,}000$ timestamps or 69 episodes ($\sim$2,500 seconds of training), the rewards and return value flatten.

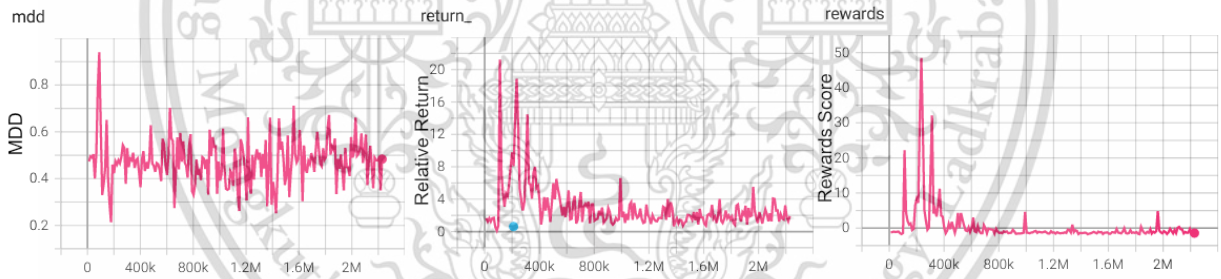

Figure 5.9: DQN Training Results

DQN Agent's test results are given in Table 5.5. By the average portfolio value over benchmark, the DQN agent outperforms the benchmark in all cases, while it fails to give the highest final value for HBu and MBu test datasets. The DQN agent, however, produces lower MDD which signifies that this agent is more risk-averse than the buy-and-hold agent.

Table 5.5: DQN Test Results

$\mathscr{D}$	Agent			Benchmark			δ	Fig.
	Return	**MDD**	***Sortino***	**Return**	**MDD**	***Sortino***		
HBu	0.1540	0.1812	*-0.2212*	**0.2643**	**0.1174**	*0.7060*	0.0204	5.10
HBr	**-0.1184**	**0.1343**	*0.7485*	-0.4484	0.5199	*-0.3982*	0.3015	5.11
MCr	**0.0296**	**0.0313**	*1.1557*	-0.0361	0.0725	*-0.2276*	0.0383	5.12
MBu	0.0357	**0.0061**	*-0.3521*	**0.0441**	0.0164	*1.0024*	0.0102	5.13

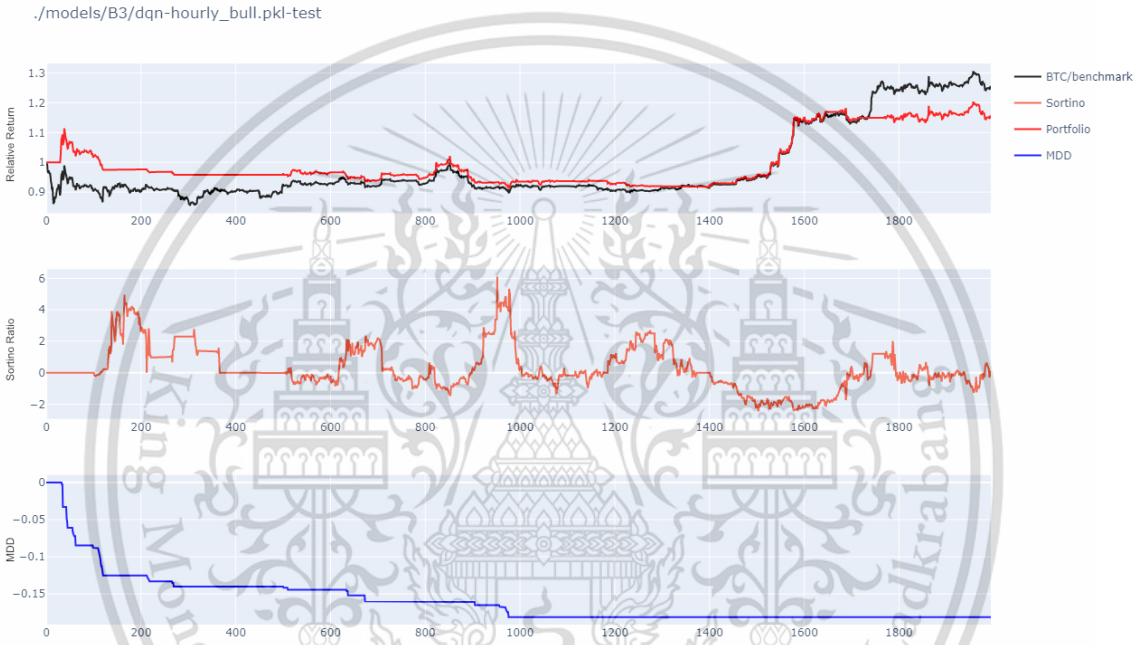

Figure 5.10: DQN agent metrics for hourly bull data

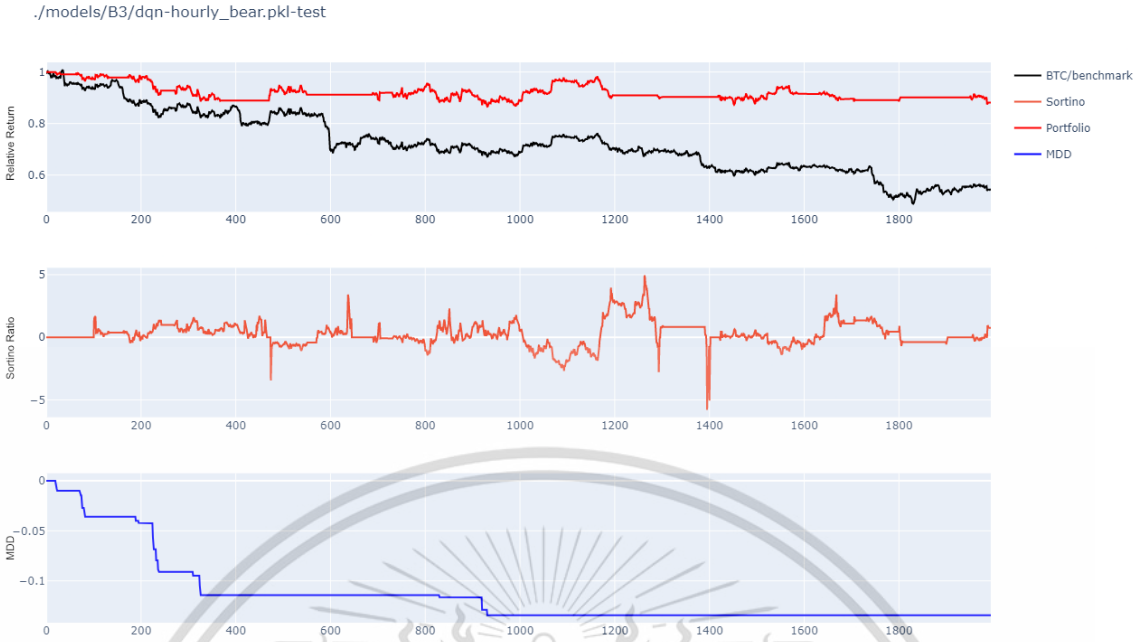

Figure 5.11: DQN agent metrics for hourly bear data

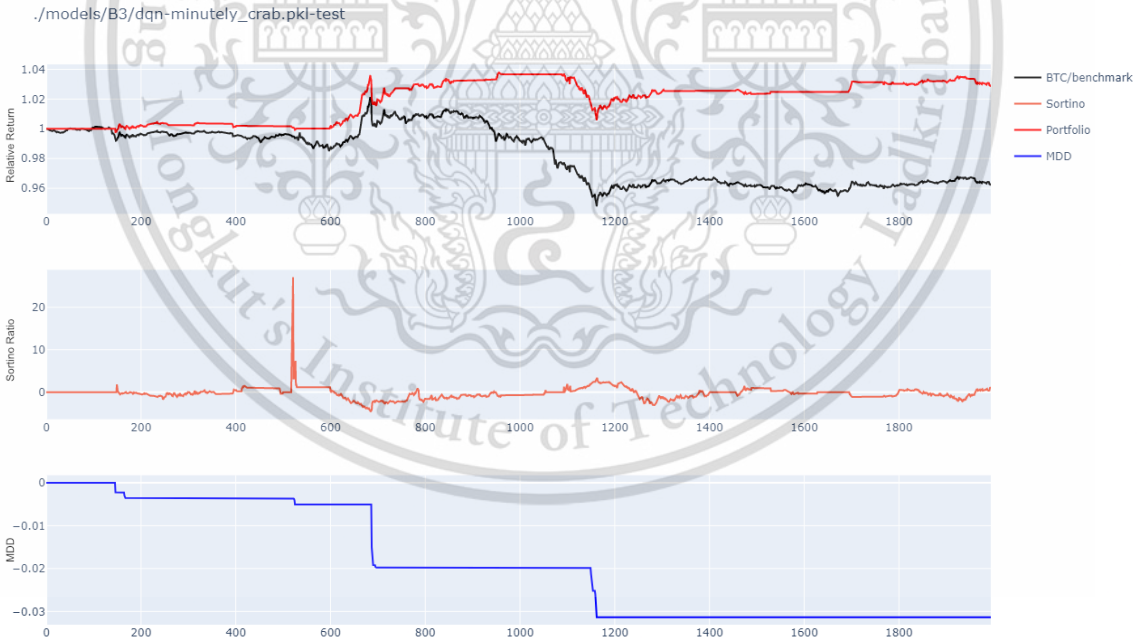

Figure 5.12: DQN agent metrics for minute crab data

Figure 5.13: DQN agent metrics for minute bull data

According to Table 5.6, the DQN agent performs only a few trades, yet the returns are powerful. Minimizing the number of trades and trading volume is ideal for trading cases where trading fees are present, to reduce fund loss from fees.

Table 5.6: Trading actions statistics for DQN agent. Value and average trade values in BTC.

	BUY			SELL			HOLD
	trades	**value**	**average trade**	**trades**	**value**	**average trade**	
HBu	15	8.35176500	0.16778655	15	8.35176500	0.16337112	1970
HBr	13	2.47752899	0.03583234	13	2.47752899	0.03368239	1974
MCr	22	9.28250000	0.11073448	22	9.28250000	0.10682621	1956
MBu	24	11.16878800	0.14659106	24	11.16878800	0.14659086	1952

5.2.4 A2C Strategy Agent (Discrete)

This A2C agent is used with a discrete action space. Using training data (see Fig. 5.14), the agent, by all factors, converges quickly at around ∼45,000 timesteps (4 episodes, ∼90 seconds of training). After that, there is virtually no improvement nor deterioration of the model.

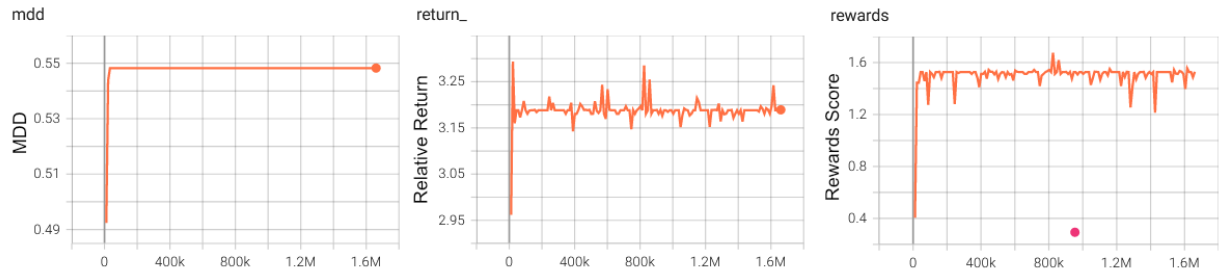

Figure 5.14: Discrete A2C Training Results

Despite the quick stable model turnout, it is suspected that the model only performs a minimal number of trades and just holds the one-time bought until the end, rendering it to be exactly comparable with the buy-and-hold agent. Refer to Table 5.7's δ column which shows that the average difference between the portfolio values of this agent and the benchmark agent is less than 0.5 percent in all cases.

Table 5.7: Discrete A2C Test Results

$\mathscr{D}$	Agent			Benchmark			δ	Fig.
	Return	**MDD**	***Sortino***	**Return**	**MDD**	***Sortino***		
HBu	0.2568	0.1425	*-0.2843*	**0.2643**	**0.1174**	*0.7060*	0.0035	5.15
HBr	-0.4543	**0.5174**	*-0.7186*	**-0.4484**	0.5199	*-0.3982*	-0.0009	5.16
MCr	-0.0368	**0.0719**	*0.8994*	**-0.0361**	0.0725	*-0.2276*	0.0000	5.17
MBu	**0.0462**	0.0165	*-0.3752*	0.0441	**0.0164**	*1.0024*	0.0000	5.18

Figure 5.15: Discrete A2C agent metrics for hourly bull data

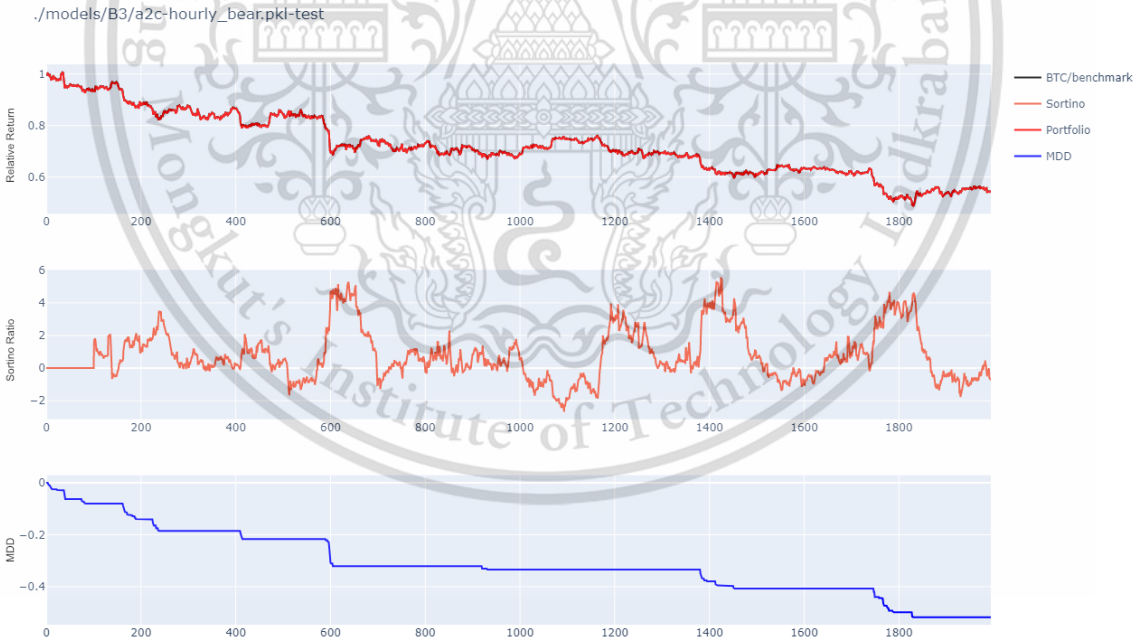

Figure 5.16: Discrete A2C agent metrics for hourly bear data

Figure 5.17: Discrete A2C agent metrics for minute crab data

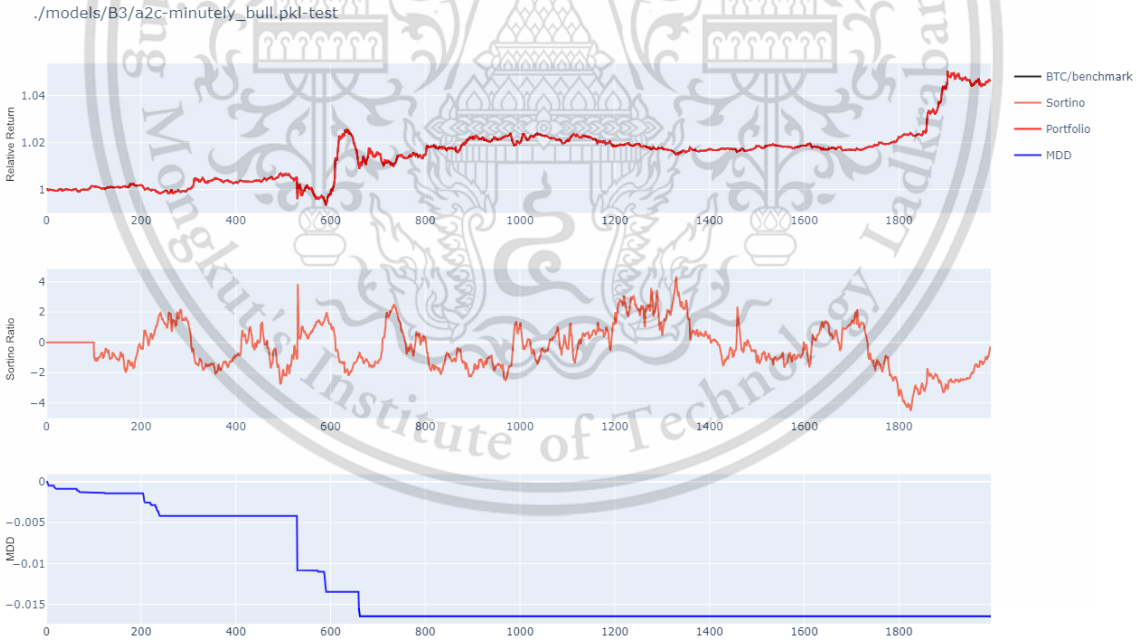

Figure 5.18: Discrete A2C agent metrics for minute bull data

The notion that Discrete A2C mimics the buy-and-hold algorithm's strategy is shown by Table

5.8: it generated only one BUY and one BUY, and the trading volumes are almost identical to that of buy-and-hold agent's.

Table 5.8: Trading actions statistics for Discrete A2C agent. Value and average trade values in BTC.

	BUY			SELL			HOLD
	trades	**value**	**average trade**	**trades**	**value**	**average trade**	
HBu	1	0.54856700	0.54856700	1	0.54856700	0.54856700	1998
HBr	1	0.14680700	0.14680700	1	0.14680700	0.14680700	1998
MCr	1	0.40502700	0.40502700	1	0.40502700	0.40502700	1998
MBu	1	0.45996300	0.45996300	1	0.45996300	0.45996300	1998

5.2.5 A2C Strategy Agent (Continuous)

This A2C agent is used with a continuous action space. Using training data (see Fig. 5.14), the agent never finds a stable ground even after more than 1,671,600 timestamps (150 episodes, 5,500 seconds): the total reward and return still fluctuates randomly, unlike its discrete counterpart. The models with zero reward did not generate any trade, including the final model, hence the last non-zero-reward model at timestamp 1,348,424 (121st episode) is taken as its representative model.

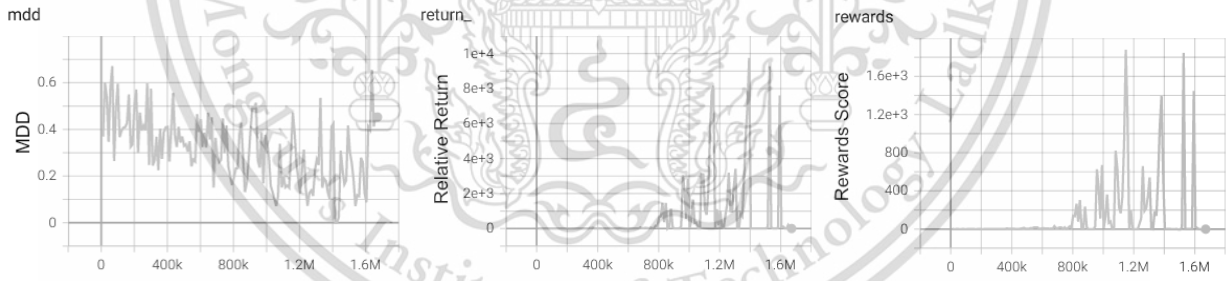

Figure 5.19: Continuous A2C Training Results

Compared to the benchmark, the representative Continuous A2C model results in a slightly more satisfiable performance in minute test datasets, at 1.68% (crab) and 0.23% (bull) average portfolio differences, 40% less drawdown, and higher final return. Conversely, this model fails to produce better performance than benchmark in all aspects. See Table 5.9.

Table 5.9: Continuous A2C Test Results

$\mathscr{D}$	Agent			Benchmark			δ	Fig.
	Return	**MDD**	***Sortino***	**Return**	**MDD**	***Sortino***		
HBu	0.0060	0.1622	*-0.5541*	**0.2643**	**0.1174**	*0.7060*	-0.0602	5.20
HBr	-0.4824	0.5618	*-1.6495*	**-0.4484**	**0.5199**	*-0.3982*	-0.0320	5.21
MCr	**-0.0137**	**0.0462**	*0.6625*	-0.0361	0.0725	*-0.2276*	0.0168	5.22
MBu	**0.0480**	**0.0101**	*-1.2651*	0.0441	0.0164	*1.0024*	0.0023	5.23

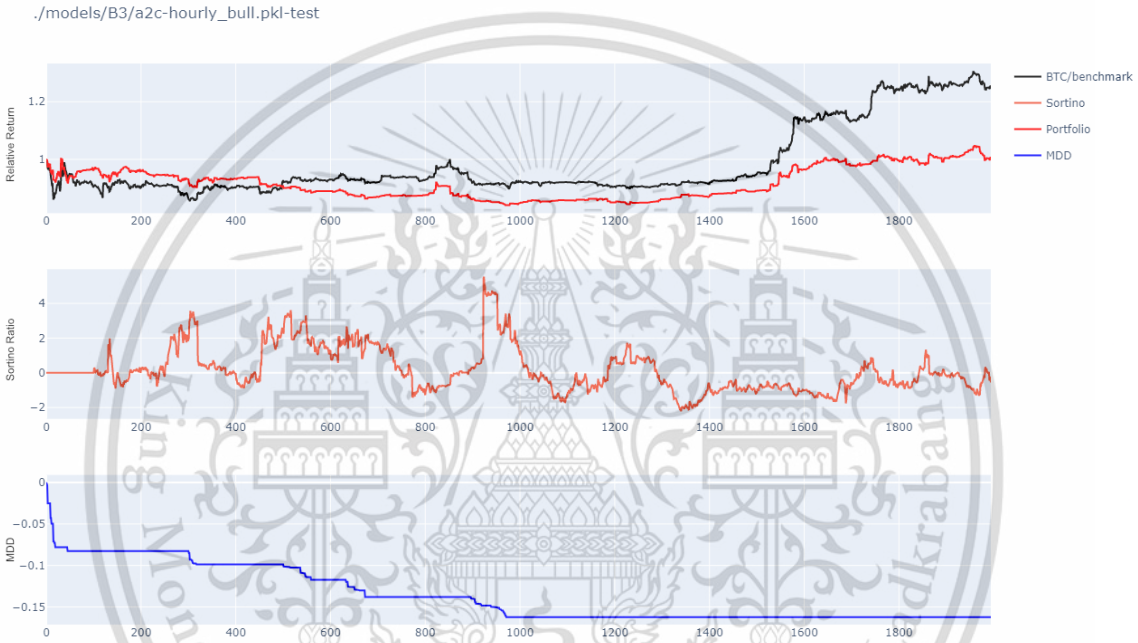

Figure 5.20: Continuous A2C agent metrics for hourly bull data

Figure 5.21: Continuous A2C agent metrics for hourly bear data

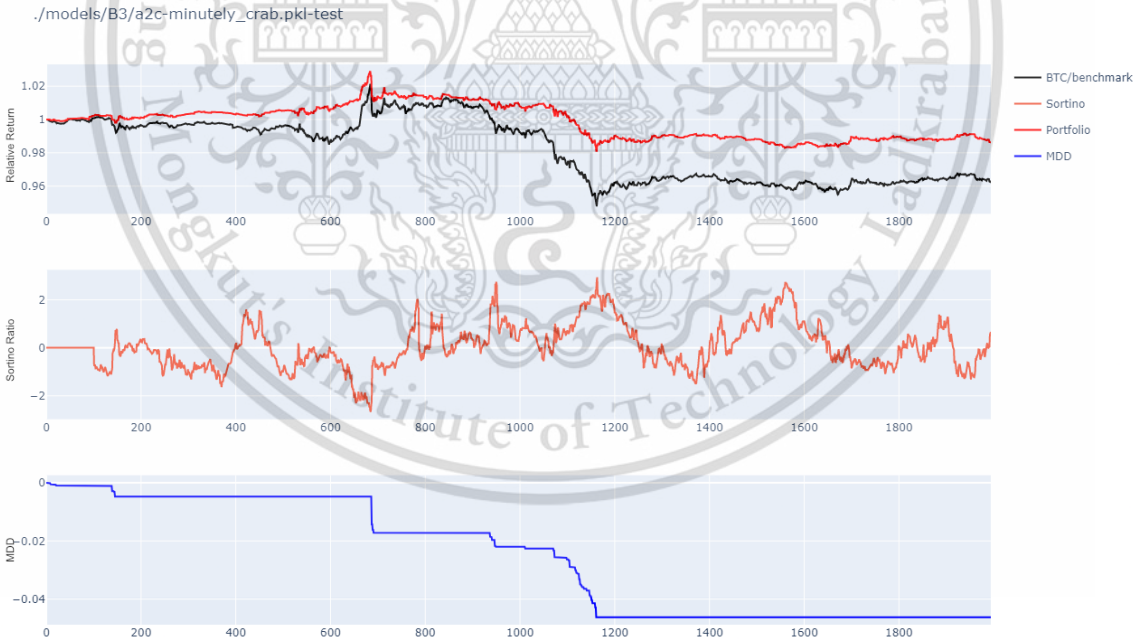

Figure 5.22: Continuous A2C agent metrics for minute crab data

Figure 5.23: Continuous A2C agent metrics for minute bull data

Table 5.10 shows the trading statistics of the Continuous A2c agent. This agent tries to balance the number of BUYs and SELLs to be exactly the same, leaving zero HOLDs. This agent generates a huge number of trade volume, despite being a continuous agent, which is not expected.

Table 5.10: Trading actions statistics for Continuous A2C agent. Value and average trade values in BTC.

	BUY			SELL			HOLD
	trades	**value**	**average trade**	**trades**	**value**	**average trade**	
HBu	1000	512.25255299	0.51225255	1000	512.25255299	0.51225255	0
HBr	1000	141.79918599	0.14179918	1000	141.79918599	0.14179918	0
MCr	1000	410.58552300	0.41058552	1000	410.58552300	0.41058552	0
MBu	1000	459.64078799	0.45964078	1000	459.64078799	0.45964078	0

5.2.6 PPO Strategy Agent (Discrete)

This PPO agent is used with a discrete action space. The Discrete PPO agent performed weakly at the first 2 million timestamps, then surged after. Among 10 training runs, similar peaks appear

and vanish in a similar way at random episodes. The representative model is taken from the final result of the training session shown in Fig. 5.24, at timestamp 2,228,800 (200th episode, ∼5,000 seconds).

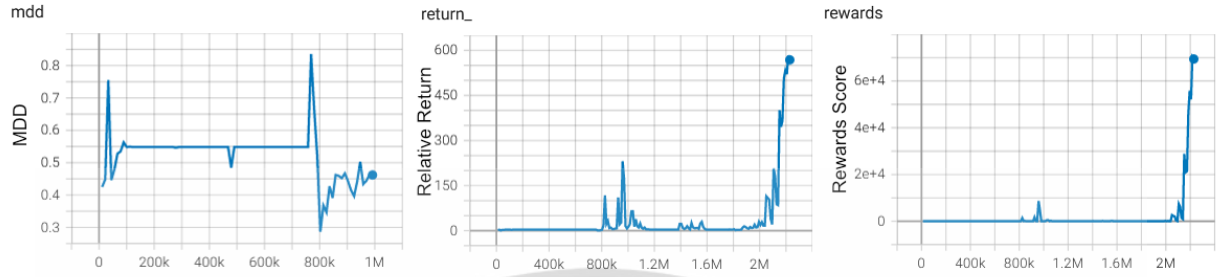

Figure 5.24: Discrete PPO Training Results

The model, scoring 600 at training return value, i.e., 59,900% profit over the training data, results in a significant profit in `HBu` and `HBr` test datasets, performing in average 59.88% and 153.96% better than benchmark, respectively. However, the Discrete PPO agent is not much better than benchmark in minute test datasets. Overall, it performs better in all aspects (return, MDD) compared to the benchmark. See Table 5.11.

Table 5.11: Discrete PPO Test Results

$\mathscr{D}$	**Agent**			**Benchmark**			δ	**Fig.**
	Return	**MDD**	***Sortino***	**Return**	**MDD**	***Sortino***		
`HBu`	**1.6982**	**0.0725**	*-1.1362*	0.2643	0.1174	*0.7060*	0.5988	5.25
`HBr`	**2.2585**	**0.1014**	*-3.2534*	-0.4484	0.5199	*-0.3982*	1.5396	5.26
`MCr`	**0.0090**	**0.0363**	*1.9876*	-0.0361	0.0725	*-0.2276*	0.0287	5.27
`MBu`	**0.0475**	**0.0146**	*-1.0178*	0.0441	0.0164	*1.0024*	0.0006	5.28

Figure 5.25: Discrete PPO agent metrics for hourly bull data

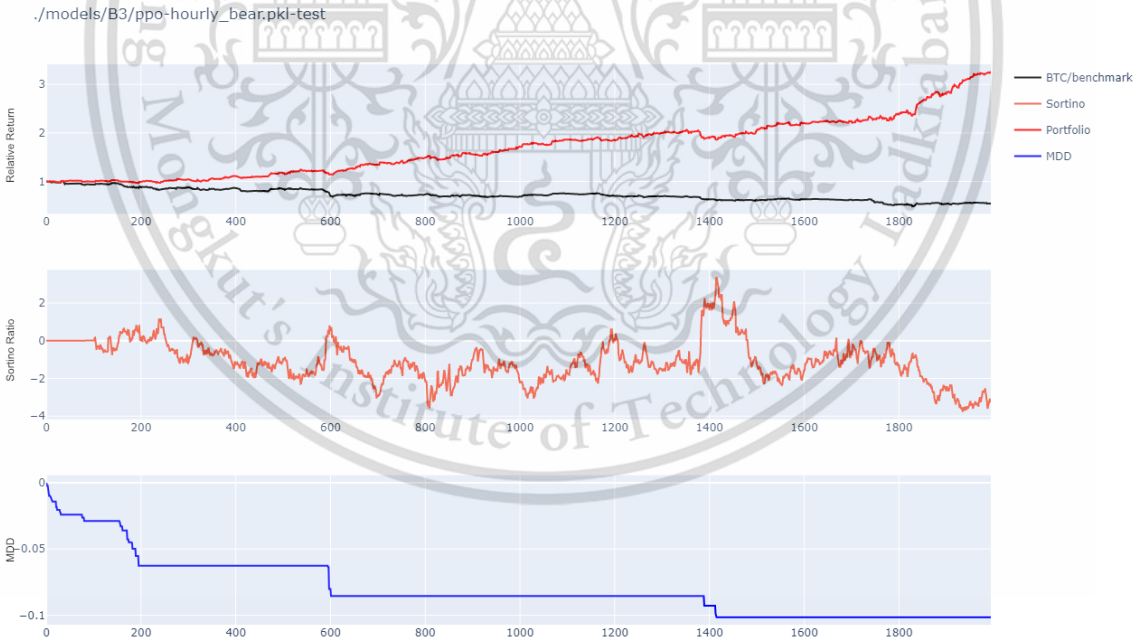

Figure 5.26: Discrete PPO agent metrics for hourly bear data

Figure 5.27: Discrete PPO agent metrics for minute crab data

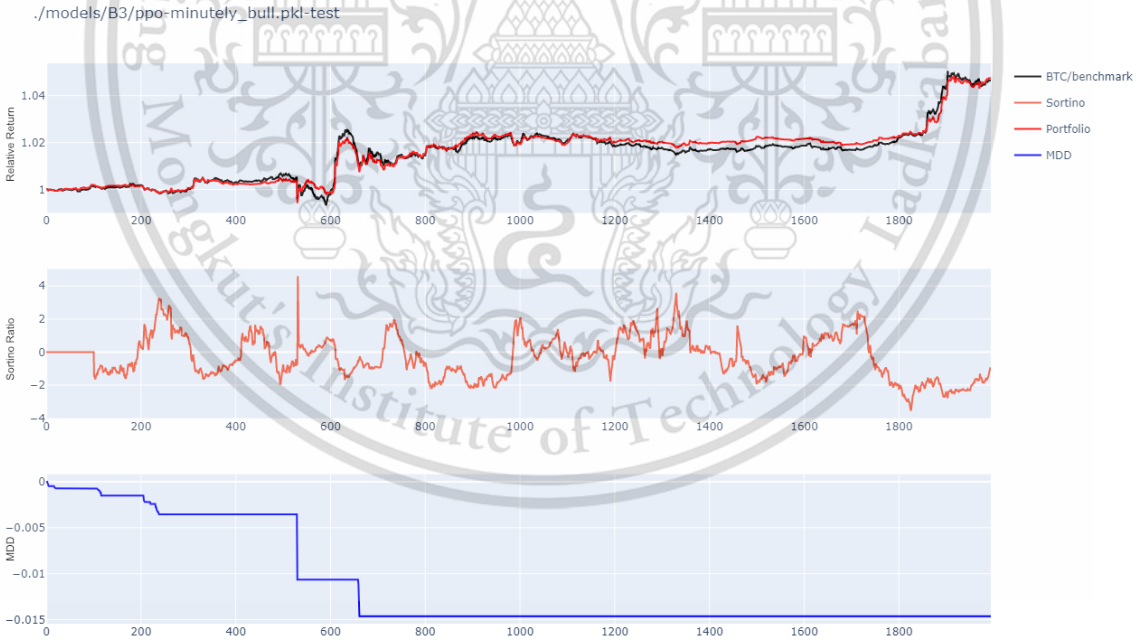

Figure 5.28: Discrete PPO agent metrics for minute bull data

Table 5.12 shows the trading statistics for the Discrete PPO agent. It generated 287–387 BUYs

and SELLs with high total volumes (more than 100 BTC trading volume) and the average volume of 0.4–0.86 BTC per trade. Like all discrete agents, the number of BUYs and SELLs as well as their total volume are identical.

Table 5.12: Trading actions statistics for Discrete PPO agent. Value and average trade values in BTC.

	BUY			SELL			HOLD
	trades	**value**	**average trade**	**trades**	**value**	**average trade**	
HBu	378	328.38829100	0.86875209	378	328.38829100	0.86875209	1244
HBr	287	117.00823000	0.40769418	287	117.00823000	0.40769418	1426
MCr	375	156.17253900	0.41646010	375	156.17253900	0.41646010	1250
MBu	387	178.13890799	0.46030725	387	178.13890799	0.46030725	1226

5.2.7 PPO Strategy Agent (Continuous)

This PPO agent is used with a continuous action space. The Continuous PPO agent results in more stable metrics over Discrete PPO, converging at around 1,800,000 timesteps (∼167 episodes, ∼5,100 seconds) after having a peaking reward value at around 1,400,000 timesteps. Refer to Fig. 5.29.

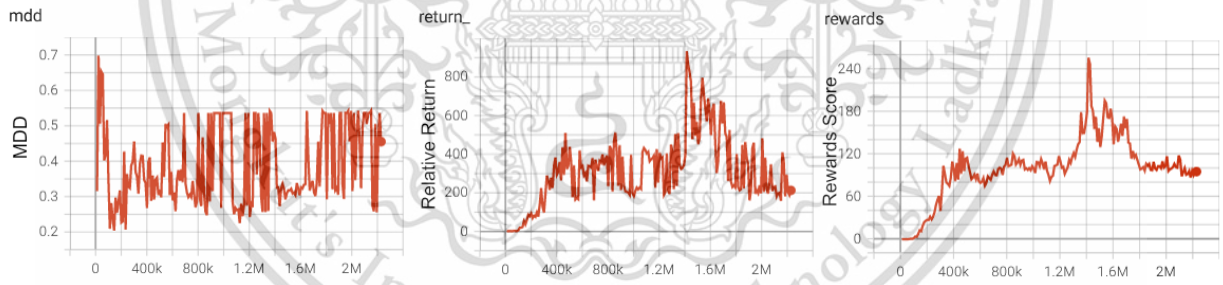

Figure 5.29: Continuous PPO Training Results

Like Discrete PPO agent but less powerful, this Continuous PPO agent performs better than benchmark in all aspects, particularly the hourly datasets. An exception is the final return value for MBu dataset, where it performs 5.6% lower than benchmark. See Table 5.13.

Table 5.13: Continuous PPO Test Results

$\mathscr{D}$	Agent			Benchmark			δ	Fig.
	Return	**MDD**	***Sortino***	**Return**	**MDD**	***Sortino***		
HBu	**0.6357**	**0.0718**	*-1.2784*	0.2643	0.1174	*0.7060*	0.1897	5.30
HBr	**0.2589**	**0.1766**	*-1.6469*	-0.4484	0.5199	*-0.3982*	0.5571	5.31
MCr	**-0.0265**	**0.0551**	*2.0327*	-0.0361	0.0725	*-0.2276*	0.0057	5.32
MBu	0.0416	**0.0118**	*-0.0144*	**0.0441**	0.0164	*1.0024*	0.0017	5.33

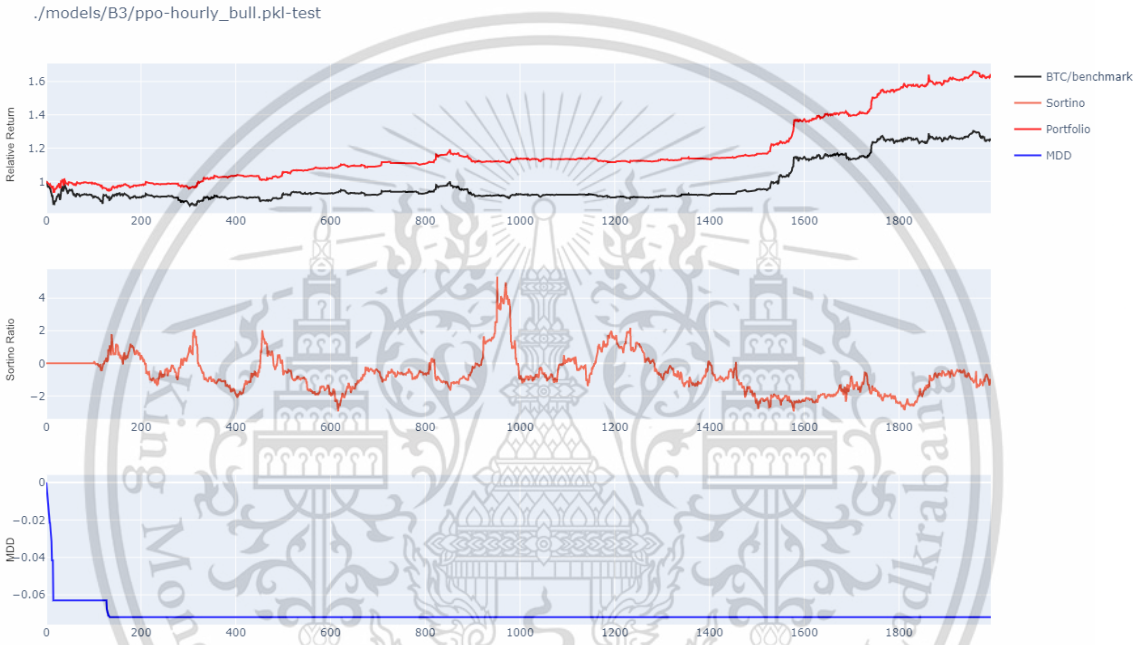

Figure 5.30: Continuous PPO agent metrics for hourly bull data

Figure 5.31: Continuous PPO agent metrics for hourly bear data

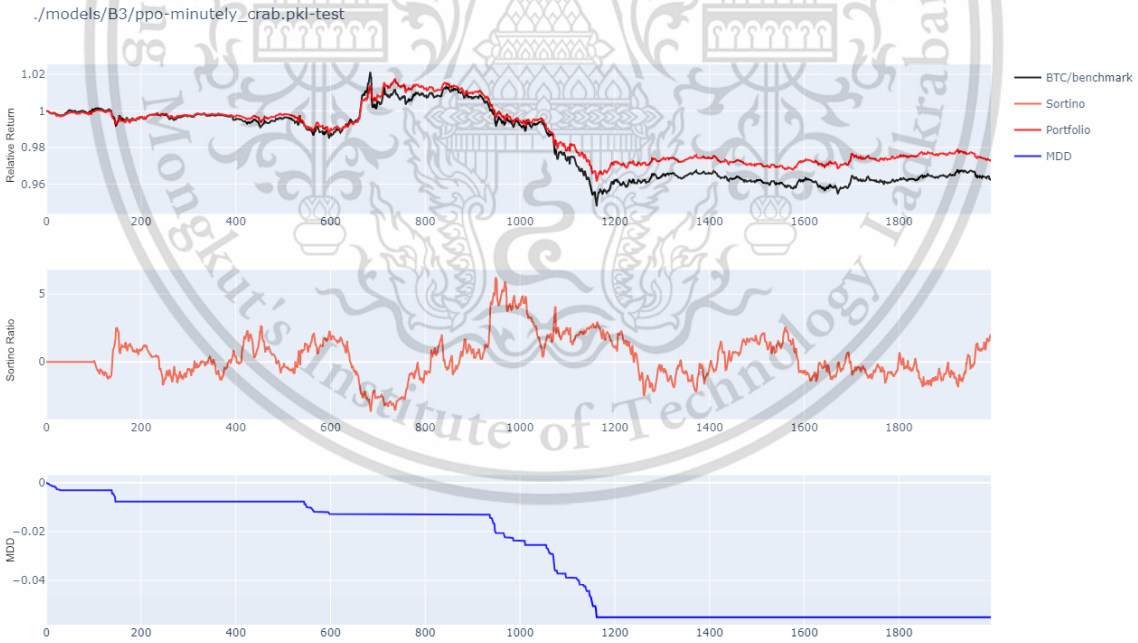

Figure 5.32: Continuous PPO agent metrics for minute crab data

Figure 5.33: Continuous PPO agent metrics for minute bull data

Table 5.14 shows that the agent tries hard to always trade and never hold: it buys little by little and sell an accumulated amount that it has bought before. The case of minute-level test sets illustrate the no-hold policy generated by Continuous PPO. It is also expected that this continuous version of PPO generates lower trading volume than its discrete counterpart.

Table 5.14: Trading actions statistics for Continuous PPO agent. Value and average trade values in BTC.

	BUY			SELL			HOLD
	trades	**value**	**average trade**	**trades**	**value**	**average trade**	
HBu	1505	219.31224699	0.14572242	469	219.31224600	0.46761672	26
HBr	1504	63.46043300	0.04219443	457	63.46040600	0.13886303	39
MCr	1593	120.81664300	0.07584221	407	120.81664500	0.29684679	0
MBu	1594	136.86520200	0.08586273	406	136.86520700	0.33710642	0

5.2.8 SAC Strategy Agent (Continuous)

The SAC agent operates in the continuous action space. Unlike the other agents, the SAC agent takes approximately 10 times more training duration to complete one episode. Using early stopping, the representative model is taken from timestamp 1,080,968 (97th episode, ∼25,800 seconds). Refer to Fig. 5.34.

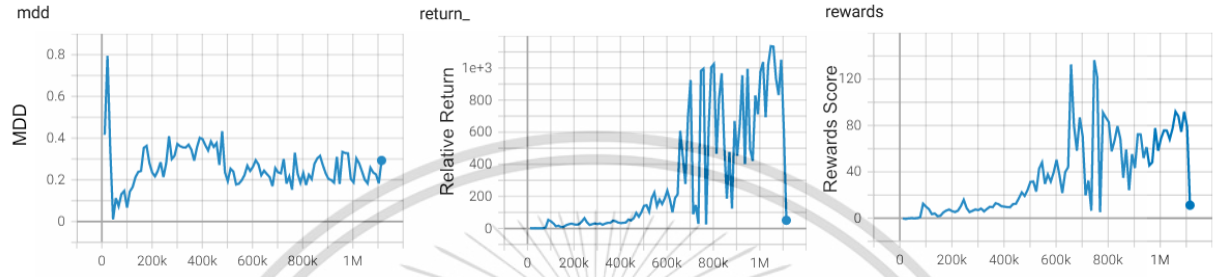

Figure 5.34: SAC Training Results

Despite the high reward and return values, on top of its longer training duration, the SAC agent performs similarly to the Continuous PPO agent, yet less satisfying. The SAC agent averages around 10–13% higher than benchmark in hourly data, but rather equal to benchmark in minute data. However, this agent successfully dampens the maximum drawdown in all cases. See Table 5.15.

Table 5.15: SAC Test Results

$\mathscr{D}$	Agent			Benchmark			δ	Fig.
	Return	**MDD**	***Sortino***	**Return**	**MDD**	***Sortino***		
HBu	**0.3782**	**0.0879**	*-1.1721*	0.2643	0.1174	*0.7060*	0.1097	5.35
HBr	**-0.1993**	**0.2915**	*-0.6846*	-0.4484	0.5199	*-0.3982*	0.1361	5.36
MCr	**-0.0295**	**0.0439**	*1.5296*	-0.0361	0.0725	*-0.2276*	0.0030	5.37
MBu	0.0222	**0.0105**	*-0.6082*	**0.0441**	0.0164	*1.0024*	-0.0065	5.38

Figure 5.35: SAC agent metrics for hourly bull data

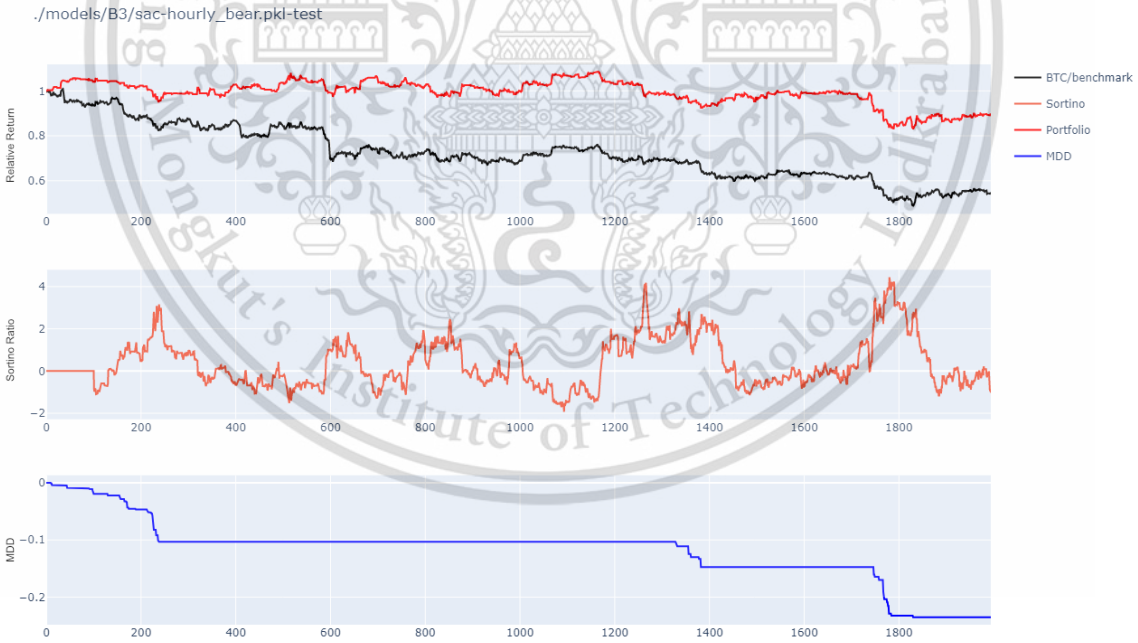

Figure 5.36: SAC agent metrics for hourly bear data

Figure 5.37: SAC agent metrics for minute crab data

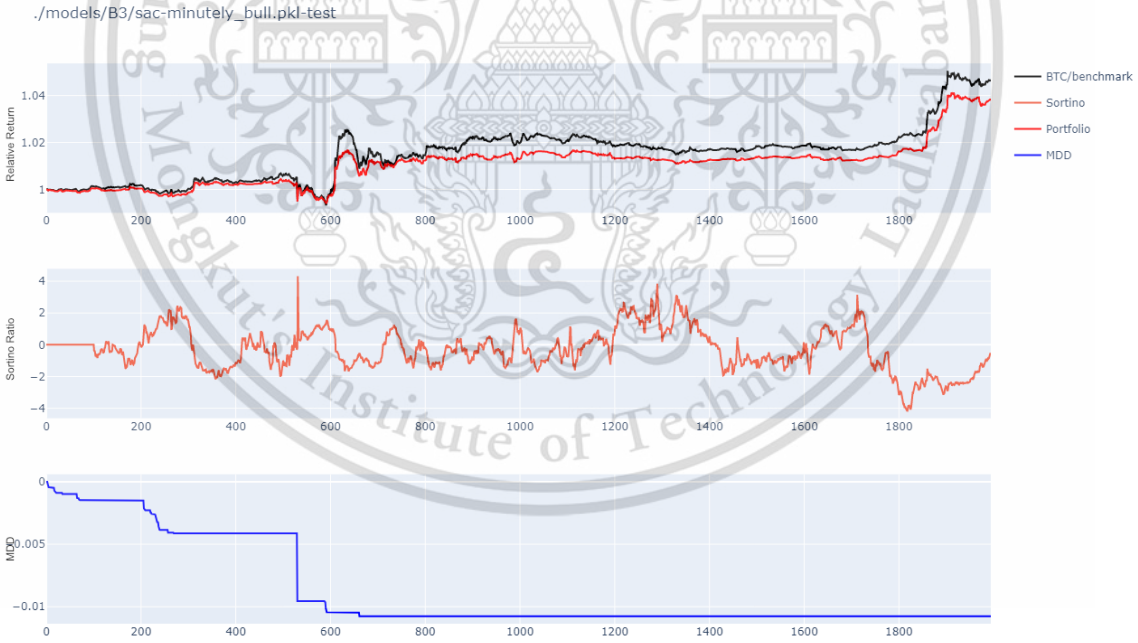

Figure 5.38: SAC agent metrics for minute bull data

The SAC agent performs more BUYs than SELLs, like Continuous PPO. However, it does not

avoid to HOLD. Consult Table 5.16. Interestingly, the total trading value in every case is very small, meaning that it performs many microtransactions instead of many big trades.

Table 5.16: Trading actions statistics for SAC agent. Value and average trade values in BTC.

	BUY			SELL			HOLD
	trades	**value**	**average trade**	**trades**	**value**	**average trade**	
HBu	774	18.62430900	0.02406241	111	18.62430900	0.16778655	1115
HBr	387	1.68412200	0.00435173	47	1.68412200	0.03583234	1566
MCr	661	9.08022700	0.01373710	82	9.08022700	0.11073448	1257
MBu	700	12.75342299	0.01821917	84	12.75342299	0.15182625	1216

5.2.9 A3C Agent (Discrete)

The discrete A3C agent consists of one global (shared) actor critic network and two sub-agents working simultaneously in two separate CPU threads. Asynchronously, the two agents run an A2C training process over the global model and merge their increment after having done an episode of training. All agents are free to submit their increment at any time, hence the name "asynchronous." Training results show completely random reward, return, and MDD trajectories, no matter how many episodes the training runs, as shown in Fig. 5.39. An anomaly can be seen from the results: the last few episodes, typically 4 to 6 episodes, see a drop in MDD. This phenomenon happens when the episode limit is around 300 and over, e.g., 500, 600, 700, but almost never below 300. For this reason, for this agent, the episode limit is set to 300. This agent is trained over 300 episodes (3,343,200 timestamps) with the duration of 7,250 seconds, choosing the model at the last episode, after MDD drops.

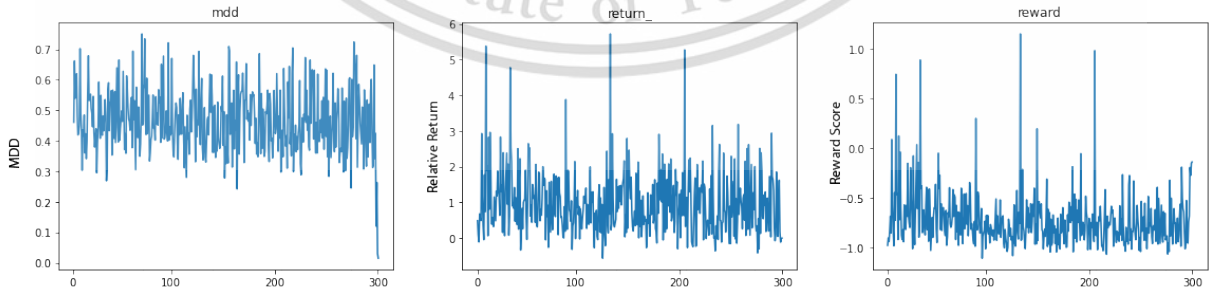

Figure 5.39: Discrete A3C Training Results

The randomness outputs a rather unpredictable result, but over several test runs, the common pattern shows that it dampens MDD and minimizes loss in bear and crab markets, but fails to catch upward motions. See Table 5.17.

Table 5.17: Discrete A3C Test Results

$\mathscr{D}$	Agent			Benchmark			δ	Fig.
	Return	**MDD**	***Sortino***	**Return**	**MDD**	***Sortino***		
HBu	0.1737	0.1201	*-1.6328*	**0.2643**	**0.1174**	*0.7060*	0.0181	5.40
HBr	**-0.1182**	**0.2632**	*-1.1074*	-0.4484	0.5199	*-0.3982*	0.2026	5.41
MCr	**-0.0227**	**0.0298**	*0.4944*	-0.0361	0.0725	*-0.2276*	0.0058	5.42
MBu	-0.0044	**0.0151**	*1.4773*	**0.0441**	0.0164	*1.0024*	-0.0209	5.43

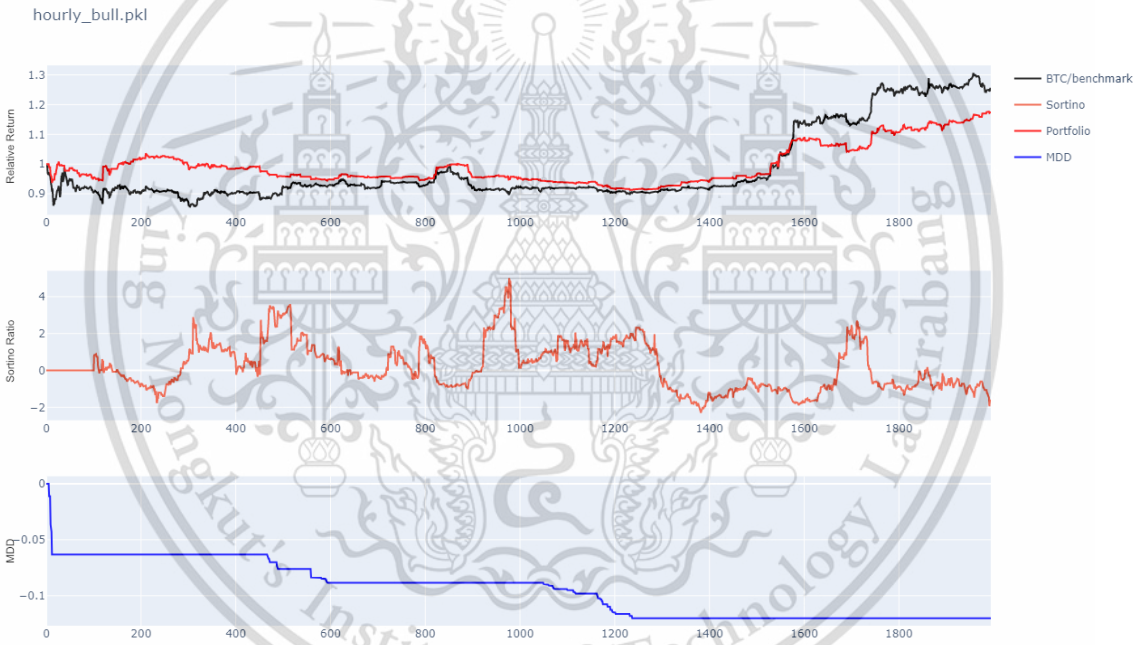

Figure 5.40: Discrete A3C agent metrics for hourly bull data

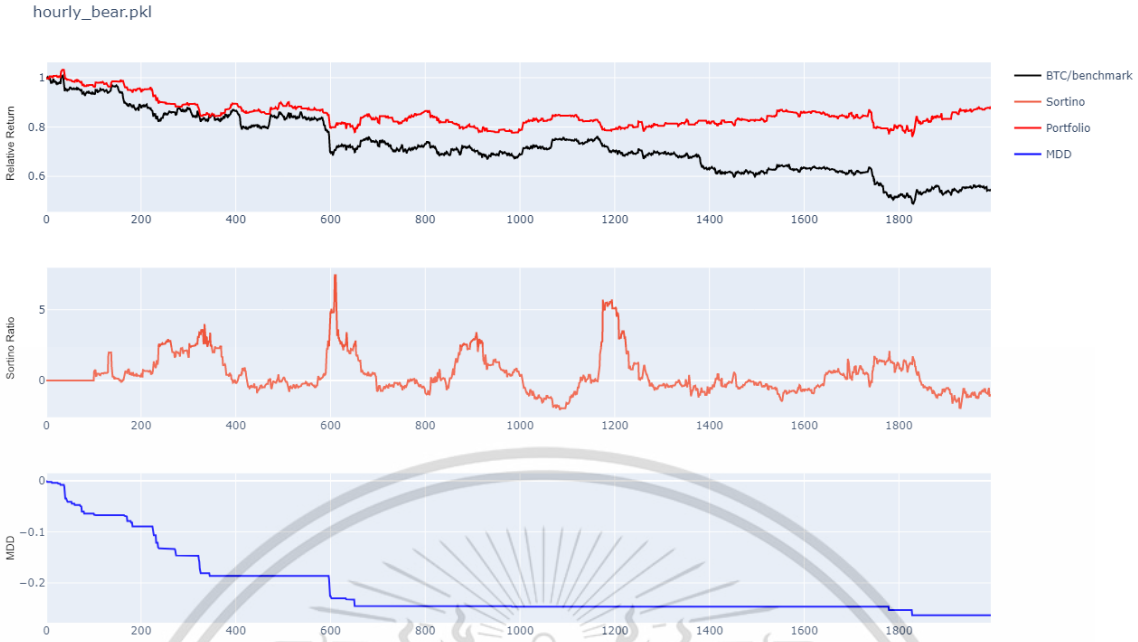

Figure 5.41: Discrete A3C agent metrics for hourly bear data

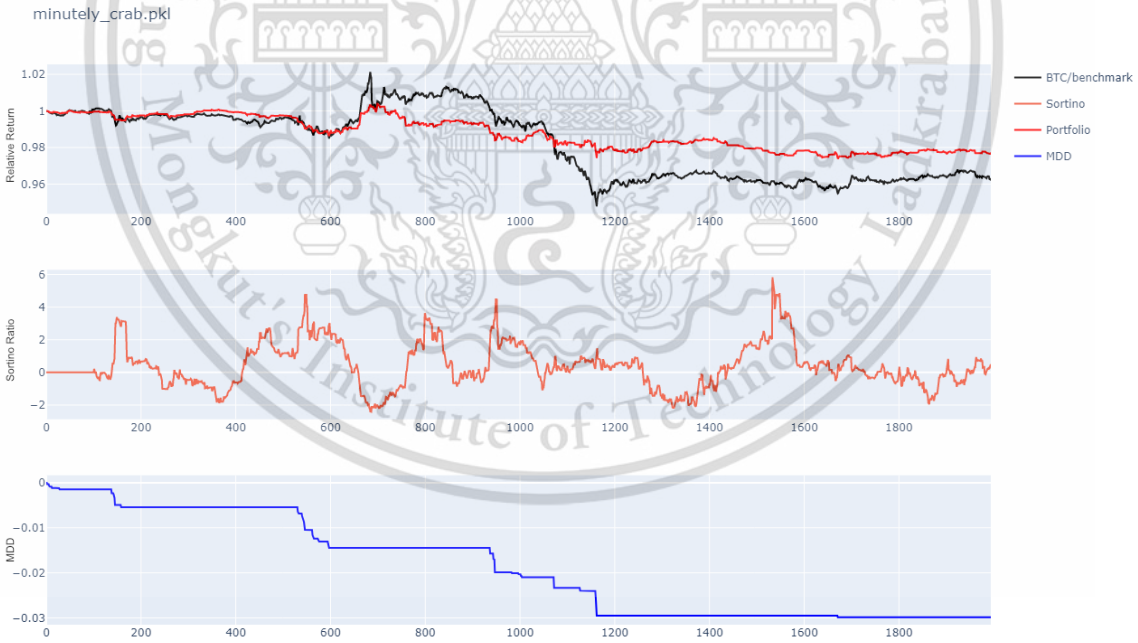

Figure 5.42: Discrete A3C agent metrics for minute crab data

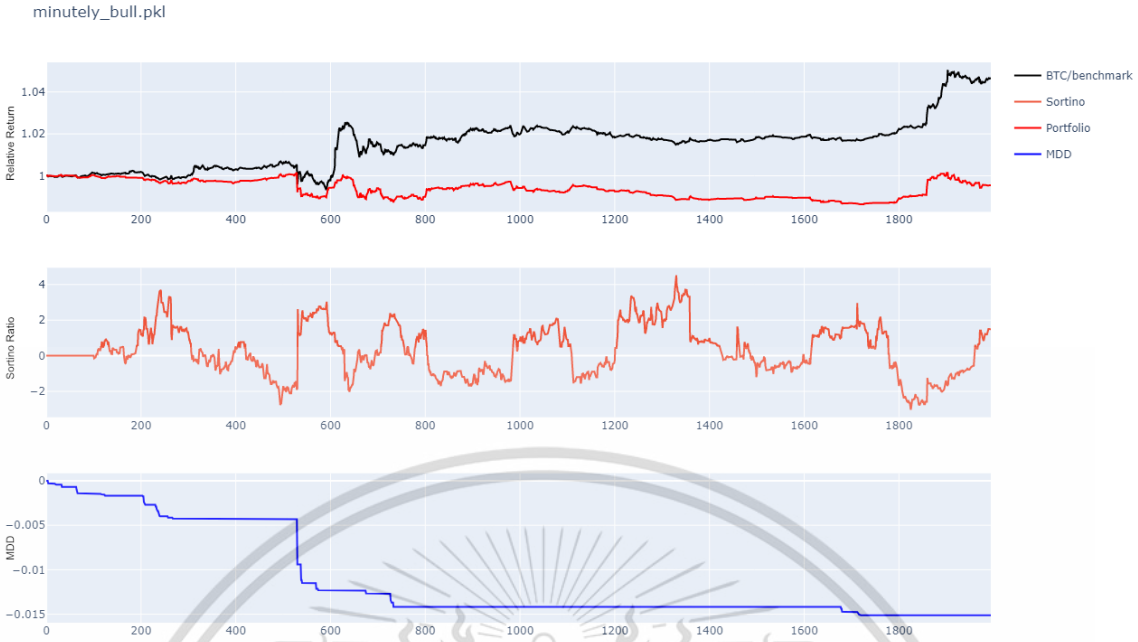

Figure 5.43: Discrete A3C agent metrics for minute bull data

Discrete A3C generates around 300 BUYs and the same amount of SELLs, and the remaining ∼1,300 HOLDs. This result is very different from Discrete A2C which tries to mimic the buy-and-hold agent. See Table 5.18 for Discrete A3C and compare it with Table 5.8.

Table 5.18: Trading actions statistics for Discrete A3C agent. Value and average trade values in BTC.

	BUY			SELL			HOLD
	trades	**value**	**average trade**	**trades**	**value**	**average trade**	
HBu	339	187.91978900	0.55433566	339	187.91978900	0.55433566	1322
HBr	313	55.36334399	0.17687969	313	55.36334399	0.17687969	1374
MCr	330	134.39802299	0.40726673	330	134.39802299	0.40726673	1340
MBu	352	158.51804599	0.45033535	352	158.51804599	0.45033535	1296

5.2.10 A3C Agent (Continuous)

The continuous A3C agent has the same architecture and procedure as its discrete counterpart, yet using a continuous environment. The happening where MDD drops at later episodes is also present

in this agent. Like its discrete counterpart, this agent is trained over 300 episodes (3,343,200 timestamps) with the duration of 7,343 seconds, choosing the model at the last episode, after MDD drops. Consult Fig. 5.44.

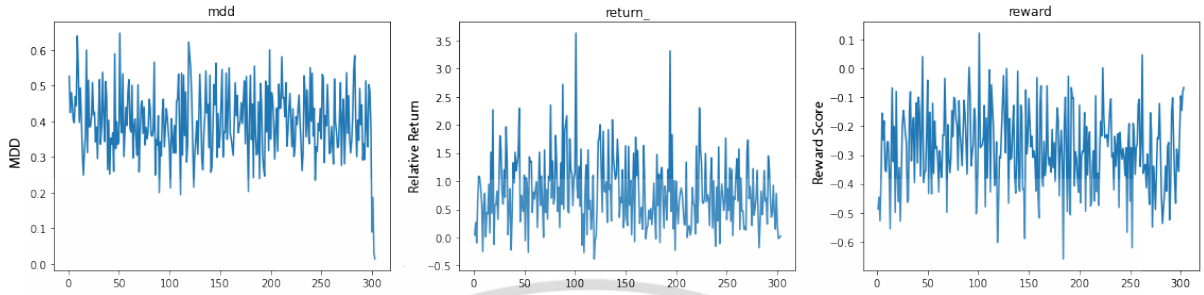

Figure 5.44: Continuous A3C Training Results

The results of Continuous A3C is as unpredictable as Discrete A3C's, yet overall, it performs much better: larger returns and lower MDD. Its strength in bear and crab markets is also more amplified than its discrete counterpart. The main difference is that Continuous A3C seems to catch upward motions much better than Discrete A3C does. See Table 5.19.

Table 5.19: Continuous A3C Test Results

$\mathscr{D}$	Agent			Benchmark			δ	Fig.
	Return	**MDD**	***Sortino***	**Return**	**MDD**	***Sortino***		
HBu	0.2147	**0.0895**	*0.7200*	**0.2643**	0.1174	*0.7060*	0.0265	5.45
HBr	**-0.0195**	**0.1872**	*-0.6413*	-0.4484	0.5199	*-0.3982*	0.4143	5.46
MCr	**-0.0068**	**0.0296**	*-0.8357*	-0.0361	0.0725	*-0.2276*	0.0083	5.47
MBu	0.0253	**0.0142**	*-0.3411*	**0.0441**	0.0164	*1.0024*	-0.0059	5.48

Figure 5.45: Continuous A3C agent metrics for hourly bull data

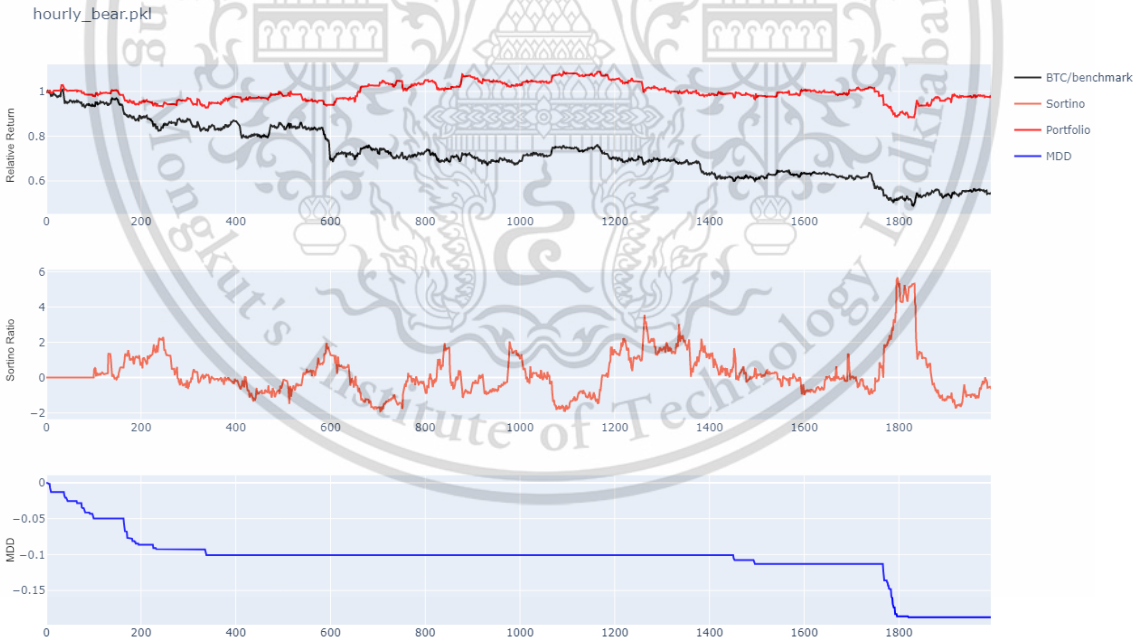

Figure 5.46: Continuous A3C agent metrics for hourly bear data

Figure 5.47: Continuous A3C agent metrics for minute crab data

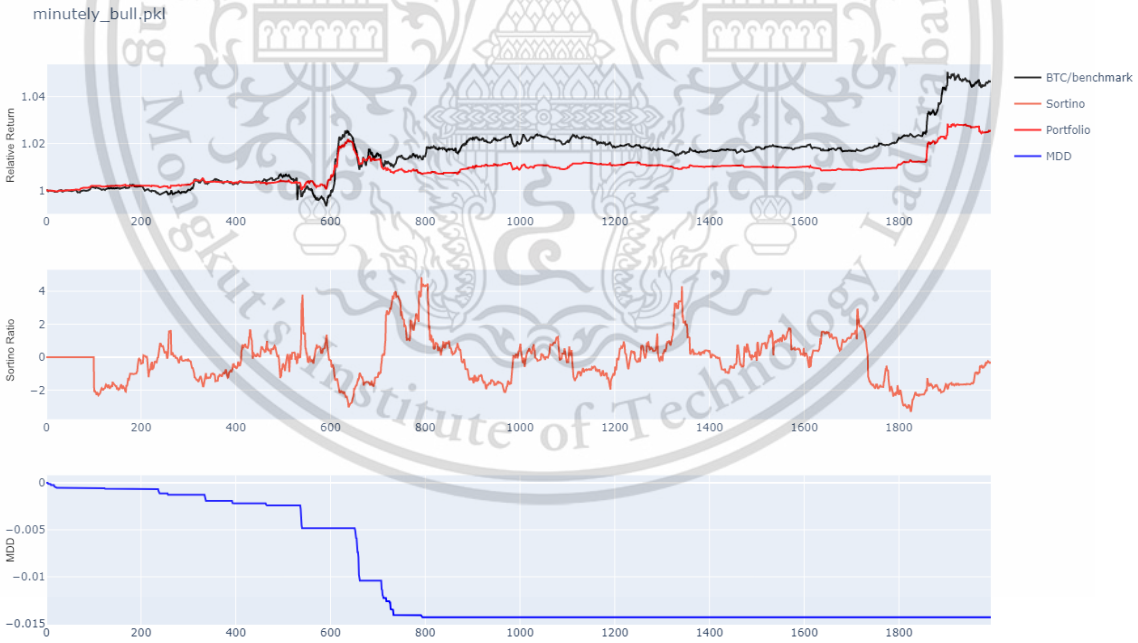

Figure 5.48: Continuous A3C agent metrics for minute bull data

Table 5.20: Trading actions statistics for Continuous A3C agent. Value and average trade values in BTC.

	BUY			SELL			HOLD
	trades	**value**	**average trade**	**trades**	**value**	**average trade**	
HBu	602	123.39140500	0.20496911	597	123.17274499	0.20631950	802
HBr	602	43.83883900	0.07282199	628	43.58935399	0.06940979	771
MCr	621	81.93492100	0.13194029	579	81.71054400	0.14112356	801
MBu	620	96.17292300	0.15511761	606	96.34372900	0.15898305	775

Chapter 6

Discussion

6.1 Results Interpretation

Excluding Sortino, six factors can be used to comparing the ten agents: final return for each test dataset, MDD for each test dataset, average over benchmark for each test dataset, number of trade, trading volume, and training time in timesteps, episode, and seconds. Agents with continuous action space is marked by "-C" and discrete "-D."

6.1.1 Agent Return and Return Over Benchmark

Table 6.1 shows the testing results based on the final results from different test datasets. Discrete PPO is by far leading in hourly data, while DQN leads when tested on `MCr`, and for `MBu`, the most profitable agent is Continuous A2C. Discrete PPO and DQN's lead is confirmed by looking at the more general fact: average portfolio return over benchmark (Table 6.2). In fact, DQN replaces Continuous A2C as the best agent for `MBu`, and the worst-performing agent is Continuous A2C, especially in hour-level data.

Table 6.1: Agent Return Comparison

$\mathscr{D}$	**Return**									
	Benchmark	**MACD**	**DQN**	**A2C-D**	**A2C-C**	**PPO-D**	**PPO-C**	**SAC**	**A3C-D**	**A3C-C**
`HBu`	0.2643	0.2161	0.1540	0.2568	0.0060	**1.6982**	0.6357	0.3782	0.1737	0.2147
`HBr`	-0.4484	-0.3389	-0.1184	-0.4543	-0.4824	**2.2585**	0.2589	-0.1993	-0.1182	-0.0195

Table 6.1 continued from previous page

MCr	-0.0361	-0.0148	**0.0296**	-0.0368	-0.0137	0.0090	-0.0265	-0.0295	-0.0227	-0.0068
MBu	0.0441	0.0364	0.0357	0.0462	**0.0480**	0.0475	0.0416	0.0222	-0.0044	0.0253

Table 6.2: Return Over Benchmark Comparison

$\mathscr{D}$	δ								
	MACD	**DQN**	**A2C-D**	**A2C-C**	**PPO-D**	**PPO-C**	**SAC**	**A3C-D**	**A3C-C**
HBu	0.0620	0.0204	0.0035	-0.0602	**0.5988**	0.1897	0.1097	0.0181	0.0265
HBr	0.1363	0.3015	-0.0009	-0.0320	**1.5396**	0.5571	0.1361	0.2026	0.4143
MCr	0.0118	**0.0383**	0.0000	0.0168	0.0287	0.0057	0.0030	0.0058	0.0083
MBu	-0.0008	**0.0102**	0.0000	0.0023	0.0006	0.0017	-0.0065	-0.0209	-0.0059

6.1.2 MDD

The data about MDD almost aligns with all other findings except for HBu and MCr: in HBu, Continuous PPO produces 1% less drawdown than Discrete PPO while in MCr, Continuous A3C dampens the loss by 0.17 percentage points. Overall, in most cases, RL agents tend to perform better in reducing drawdown than the plain buy-and-hold algorithm in minute level.

Table 6.3: Agent MDD Comparison

$\mathscr{D}$	**MDD**									
	Benchmark	**MACD**	**DQN**	**A2C-D**	**A2C-C**	**PPO-D**	**PPO-C**	**SAC**	**A3C-D**	**A3C-C**
HBu	0.1174	0.1159	0.1812	0.1425	0.1622	0.0725	**0.0718**	0.0879	0.1201	0.0895
HBr	0.5199	0.3570	0.1343	0.5174	0.5618	**0.1014**	0.1766	0.2915	0.2632	0.1872
MCr	0.0725	0.0427	0.0313	0.0719	0.0462	0.0363	0.0551	0.0439	0.0298	**0.0296**
MBu	0.0164	0.0124	**0.0061**	0.0165	0.0101	0.0146	0.0118	0.0105	0.0151	0.0142

6.1.3 Number of Trades and Trading Volume

Table 6.4 shows the number of trades generated from each test case by each agent. Numbers preceded by an asterisk (*) indicate a large imbalance between the number of BUYs and SELLs; those with a caret (^) indicates a small imbalance; otherwise, the two numbers are identical. We

can see that A2C-D indeed copies Benchmark's strategy by buying only once and selling only once. A2C-C and minute-level PPO-C, on the other hand, avoids holding at all costs.

Next, Table 6.5 shows the trading volume generated from each test case by each agent. Normally, `BUY` and `SELL` volumes should be identical, but that is not the case for A3C-C. Apart from single-trading Benchmark and A2C-D, SAC and DQN generate the least trading volume. The others would trade a total of hundreds of BTC – this is not very ideal for usage in a live trading environment.

Table 6.4: Number of trades generated by each agent type.

$\mathscr{D}$	**Number of trades** (`BUY`s + `SELL`s)									
	Benchmark	**MACD**	**DQN**	**A2C-D**	**A2C-C**	**PPO-D**	**PPO-C**	**SAC**	**A3C-D**	**A3C-C**
`HBu`	2	138	30	2	2000	756	*1978	*885	678	^1199
`HBr`	2	160	26	2	2000	574	*1961	*434	626	^1230
`MCr`	2	150	44	2	2000	750	*2000	*743	660	^1200
`MBu`	2	148	48	2	2000	774	*2000	*784	704	^1226

Table 6.5: Trading volume generated by each agent type, in the nearest 0.01 BTC.

$\mathscr{D}$	**Trade Volume** (`BUY` + `SELL`)									
	Benchmark	**MACD**	**DQN**	**A2C-D**	**A2C-C**	**PPO-D**	**PPO-C**	**SAC**	**A3C-D**	**A3C-C**
`HBu`	0.55	80.12	16.70	0.55	1024.50	656.76	438.62	37.24	375.82	^246.56
`HBr`	0.15	28.16	4.95	0.15	283.60	234.01	126.92	3.36	110.72	^87.41
`MCr`	0.41	61.48	18.57	0.41	831.07	312.34	241.62	18.16	268.78	^163.64
`MBu`	0.46	68.01	22.33	0.46	919.28	356.27	273.72	25.50	317.02	^192.51

Theoretically, in a real-life automated trading with fees incurred, those which generate large amount of trade would waste more fees, potentially wiping large profits. However, this thesis does not cover such cases, since in early tests, an introduction of trading fees would lead to agents hesitant to take any trade. Further more thorough research is needed to find a solution that avoids such problem. Moreover, it should also be able to tell if training fee-enabled agents with different fee structures will result in more unpredictable, less certain outcomes or not.

Combined with the knowledge about agents' return, the most impressive one among all here is DQN. Despite its low number of trading and despite being a discrete agent, it still makes the

highest average return over benchmark in minute-level test sets. This agent, as-is, will definitely perform better in fee-enabled environments compared to other agents.

6.1.4 Time Performance

It is undoubtably true that the buy-and-hold and MACD agents win the training time comparison, since they do not require any training to operate. Among RL algorithms, Discrete A2C only takes around 4,500 timesteps (4 episodes, 90 seconds) to find its ideal model, yet at the end it just mimics the buy-and-hold agent's performance. The second fastest model is DQN, with clear advantage when tested with minute-level test datasets. Continuous PPO converges before the episode limit (200 episodes), and results in a decent lead over benchmark. Discrete PPO and SAC failed to converge before the training episode limit (150 and 100, respecively), while it is difficult to tell if Continuous A2C and both A3Cs would ever converge, due to its unstable nature.

Table 6.6: Time Performance Comparison, timestamps in thousands

	s							
	DQN	**A2C-D**	**A2C-C**	**PPO-D**	**PPO-C**	**SAC**	**A3C-D**	**A3C-C**
Timesteps (k)	∼770	**∼45**	> 1,671	> 2,228	∼1,800	> 1,114	> 3,343	> 3,343
Episodes	69	**4**	> 150	> 200	∼167	> 100	> 300	> 300
Time (sec)	∼2,500	**∼90**	> 5,500	> 5,000	∼5,100	> 25,800	> 7,250	> 7,343

Among all agents tested in this environment, according to results, MDD, and average portfolio value above benchmark, DQN performs the best in minute-level (in this case, less volatile) data while PPO-D fits hour-level data better (more volatile). Within expectation, discrete-space agents gives higher return compared to continuous-space agents, since discrete-space agents have less actions to consider and higher exposure to the market.

Overall, RL algorithms can outperform conventional trading strategies in profit maximization and risk aversion, yet they take time to train the model. Despite that, it is apparent that since RL models can be re-trained with novel data, then it can be improved by randomly searching and training the model with various kinds of data.

6.2 Comparison with Literature Review

Contrary to the findings in Sagiraju and Mogalla's work [1] suggesting that PPO performs the worse in cumulative return but best in MDD, PPO in this research performs the best in terms of return, yet it is confirmed that it dampens MDD best only when tested in hourly data. Another discrepancy can be seen by looking at the results of A2C and DQN. The aforementioned work finds that the two performs similarly, while in this thesis, A2C performs rather ineffectively in all cases and DQN is the most preferable in minute-data returns.

This part will discuss about the comparison between A2C and A3C as is performed by Espeholt et al. [7] and Sewak [8]. Comparing the algorithms' speed, Espeholt et al. [7] show that batched A2C with 48 CPUs runs rather at par compared to A3C with 32 workers in 64 CPUs, yielding different findings depending on which problem it solves. See Section 2.4. Experiments in this thesis indicate that using a 2-thread single CPU, A2C-D runs at $\sim$500 FPS, A2C-C at $\sim$300 FPS, and both A3Cs at $\sim$450 FPS. Again, the results moderately differ, and there is no solid conclusion that can be drawn from this comparison.

Sewak [8] points out that A3C's dyssynchrony of knowledge can cause unstable training and slow convergence. This explains the stagnation found in both cases of A3C training in this research. Moreover, this might also explain the drop in MDD at the end of each training session: agents finishing last would try to push their knowledge increment without influence of others' update. However, Sewak's remarks that A2C improves A3C does not apply to non-time performance results of this thesis: in most cases, A3C performs better in return, MDD, and δ than A2C.

Overall, Ouellette's claim [5] that results given by RL can be highly varied and unpredictable between experiments becomes more convincing. Comparing algorithms is a tricky task: hyperparameter tuning might be essential in finding the actual potential of an algorithm.

6.3 RL Design Evaluation

State Model

The current RL agent sees the past five closing prices movement to make a trading decision. Five is a rather ideal window size to capture just enough pattern variation: not too specific (when

using a number larger than 5) and not too general (smaller than 5). Taking MACD and EMA into calculation does not improve the outcome (Fig. 4.6), since the two lagging indicator helps more when used in a long-term window [47].

Action Model

Since this research focuses on trading at the spot market with orders at the market price, agents' actions (`BUY`, `SELL`, and `HOLD`) already cover all possible actions. Discrete-spaced agents buy using the whole balance and sell the whole asset, while continuous agents can determine a good percentage of balance or asset to spend.

Reward Model

The reward model is affected by three factors: (1) profit and additional MDD it bears when selling, (2) penalty when holding too long, and (3) reward when holding is in profit. However, due to the volatility of results fed by different training processes, it is difficult to determine the ideal reward model, for example, the proportion of reward coming from profit and MDD, what other factors should influence reward, etc. Sortino Ratio has also been excluded from reward calculation because of the unpredictability of the meaning of Sortino Ratio relative to price action. Another limitation of this reward model is not being fee-agnostic, meaning that it does not take trading fees into trading loss.

6.4 Recommendations

6.4.1 Live Trading

A robust trading model should be compatible with live trading, however, there are many more variables to consider when designing a live trading agent on top of the commonly exhibited backtest-based trading agents.

Trading Fees

Incorporating trading fees in an RL model needs to be addressed in future researches, especially how to tune an agent such that it will perform well with any level of trading fees (fee-agnostic), while still encouraging agents to take a trade. One idea would be implementing a more condition-based reward function, that is, if an agent will be punished when it wastes too much money for fees or when it misses profitable trades due to hesitance. It might imply that the state space might have to be modified to include these interdependent factors.

Market Impact

The most important difference between backtest agents and live agents is the market impact. A real trade causes the market to adjust its demand and supply, which ultimately reflects in a price movement. The higher the trading size relative to the overall trade volume, the higher the market impact is, i.e., the more volatile the market will react. A good trading model should adjust its trading size to minimize market impact. Some [48, 49] have attempted to construct a live trading simulator, yet such implementation is still not seen in common paper trading simulators [50]. A recommendation would be providing an open-source live trading simulator to test trading agents in a more realistic environment.

Order Mechanism

Related to market impact, a live trading environment relies on order books and fulfillment mechanism to process trades, meaning that a market order might have to be decomposed into several smaller orders, not necessarily at different prices, to match the current demand and supply. This factor may alter the price and volume of an order instructed by the model. This requires a more flexible RL agent with more complex, continuous action spaces.

6.4.2 RL Design in Live Trading

State Space

The current agent design only incorporates past closing prices in its state space, but as previously suggested, more factors might be needed to be taken into account, for example:

- Current order book prices and order volumes at each price demand or supply,
- Past order volumes,
- High, low, open, close prices if necessary.

A more far-fetched approach could include sentiment data [51], yet such idea is excluded from this research's interest.

More Actions

More actions are essential to support live trading, especially the ability to place an order, as opposed to performing a market order. Some useful yet complex actions also cover taking profit (TP) and stop loss (SL), but this might be confused with a regular `SELL` order.

6.4.3 Model Tuning and Maintenance

Common issues faced by RL developers are tuning hyperparameters and automatically selecting a model from the best episode. Hyperparameters' effectiveness might change when different data is provided [52] and the final product of a training might not result in the best model (as presented in Chapter 5). A meta-learning system [52, 53] might be helpful to tune hyperparameters, while one can select the best model by iteratively saving the best episode's model so far based on specific metrics, like training rewards or training loss. Another idea is periodic training: a trading agent model might have to update its knowledge periodically to keep up with new patterns and price actions. Yet, this idea requires a proper, separate research to determine its usability.

Chapter 7

Conclusion

In short, this research concerns about comparing different trading agent algorithms in terms of return, MDD, and the time to get an optimal model. The agents, operating on discrete or continuous action space, are trained against a hourly crab market data and tested on four different data with different trends and different timeframes.

Comparing eight agent types (buy-and-hold, MACD, DQN, Discrete PPO, Continuous PPO, Discrete A2C, Continuous A2C, SAC, A3C, and Continuous A3C), the best model generated by the Discrete PPO agent returns 59.88% higher than benchmark on `HBu` dataset and 153.96% higher on `HBr` dataset, which is the highest among all agents. The lead is taken by the DQN agent with 3.83% and 1.02% higher return than benchmark in `MCr` and `MBu` datasets, respectively. As for MDD, the Continuous PPO model gives the lowest MDD at only 7.18% on the `HBu` model, compared to 11.74% when following the benchmark. For `HBr` model, Discrete PPO reduces drawdown by 80.5% (at 10.14%) compared to benchmark (51.99%). The best agent for both `MCr` and `MBu` is also DQN, which respectively results in 3.13% and 0.61% drawdown, compared to 7.25% and 1.64% for benchmark.

As for the statistics regarding trading count and volume, when cross-checked with return, DQN would come at the top, since it generates only a few trades and low volume and being profitable at the same time. Continuous A2C would perform the worst from generating too much trades and at the same time returning much lower than benchmark, especially in hour-level data.

Buy-and-hold and MACD are the best agents with respect to the time it takes to train an optimal model, since no training is required for both agents. As for RL agents, Discrete A2C completes

the convergence faster at around 45,000 steps (4 episodes, around 90 seconds), yet it performs exactly as good as the Buy-and-hold agent. The best agent for minute-level data, DQN, requires in average 770,000 timesteps (69 episodes, around 2,500 seconds) to converge. Continuous PPO, the best agent in terms of MDD for `HBu`, requires more than 1.8 million timesteps (167 episodes, 5,100 seconds) of training to produce the best model. Meanwhile, the convergence time for Continuous A2C, Discrete A3C, Discrete PPO, and SAC are unknown due to the high volatility of training results.

The RL model used in this research is suitable for a backtrade-based spot market trading: it only accepts past five-timeframe closing data as the observable space, the actions are limited to `BUY`, `SELL`, and `HOLD` with a variable amount (continuous) or the whole portfolio value (discrete). It is not sufficient to be deployed in a live trading environment, which requires a larger observation horizon to consider, such as market impact, order book, and possibly market sentiment. This research's reward model also needs an improvement in terms of balance between rewarding profit versus MDD minimization and holding penalty. Moreover, another aspect to consider is a trading cost-agnostic model, to accommodate environments where a trade costs a percentage or fixed fee. Finally, a meta-training might be useful to support the quality of the training, by tuning hyperparameters crafted to each training dataset or environment.

Appendix A

Symbols and Variables

Table A.1: List of Symbols

Symbol	Scope	Meaning
$\mathscr{D}$	general	dataset
d_t	general	datapoint at timeframe t
p_t	general	closing price at timeframe t
δ	general	average return over benchmark
$EMA(n)$	indicator	exponential moving average with window size n
$MACD(f,s,l)$	indicator	movement average convergence/divergence of fast linie period f, slow line period s, and signal line period l
-C	agent	RL algorithm/agent with a continuous action space
-D	agent	RL algorithm/agent with a discrete action space
s	agent	approximate time for an agent's training reward value to converge
α	agent	RL agent
$\odot$	agent	buy-and-hold agent (benchmark)
$\phi_\alpha(t)$	portfolio	portfolio size of agent α at timeframe t
$\phi_\odot$	portfolio	benchmark (buy-and-hold) portfolio return
MDD_t	portfolio	maximum drawdown at timeframe t

Appendix B

Terminology

Table B.1: List of Terms, sorted by importance

Term	Meaning
ROI	Return on investment: the asset value after performing trading.
Benchmark	the investment or trading strategy that is used as a reference or baseline.
Agent	A code or program that performs trading using a specific algorithm.
Portfolio	The total investment valuation of a trading agent.
ATH	All-time high: the highest value that a portfolio has ever attained.
MDD	Maximum drawdown: the maximum portfolio loss in percent or ratio from portfolio's ATH.
Sortino ratio	The ratio of portfolio return subtracted by benchmark return, divided by the standard deviation of portfolio return. Penalized return below benchmark.
Bull market	An asset price trajectory whose trend is upwards.
Bear market	An asset price trajectory whose trend is downwards.
Crab market	An asset price trajectory whose trend is sideways.
Buy	An act of buying an asset using a reference asset. In the case of `BTCUSD`, a "buy" action trades `USD` for `BTC`.
Sell	An act of selling an asset into a reference asset. In the case of `BTCUSD`, a "sell" action trades `BTC` for `USD`.

Table B.1: List of Terms (continued)

HODL or hold	A commonly used typographical error of “hold," means to perform no trading at a given timeframe.
Continuous agent	a trading agent that can buy and sell assets using a part of its portfolio
Discrete agent	a trading agent that must buy and sell assets using its whole portfolio
`HBu`	Hour-level bull test dataset. See Section 4.1 for details.
`HBr`	Hour-level bear test dataset. See Section 4.1 for details.
`MCr`	Minute-level crab test dataset. See Section 4.1 for details.
`MBu`	Minute-level bull test dataset. See Section 4.1 for details.

Appendix C

Conference Paper

Bitcoin Trading Using Deep Reinforcement and Supervised Learning

Daniel Ekwuazi
Faculty of Engineering
King Mongkut's Institute of Technology Ladkrabang
Bangkok, Thailand
danielekwuazi@gmail.com

Chaiwat Nuthong
Faculty of Engineering
King Mongkut's Institute of Technology Ladkrabang
Bangkok, Thailand
chaiwat.nu@kmitl.ac.th

***Abstract*—The cryptocurrency market is known for its volatility; consequently, issues arise regarding when to buy and what quantity to buy, to maximize return in the long term. This paper aims to implement a deep reinforcement learning and supervised learning; Long short-term memory (LSTM) approach to cryptocurrency trading, specifically bitcoin. Experimental results showed that deep reinforcement learning outperformed the supervised learning approach.**

Keywords— bitcoin; time series; deep reinforcement learning; Supervised Learning, Predictive model

I. INTRODUCTION

The creation of Bitcoin [1] in 2008 and the subsequent bloom in 2017 brought about massive interest in the cryptocurrency market and the (initial coin offering) ICO era, this interest has peaked and ebbed in the following years. Bitcoin was the largest and most popular cryptocurrency in cryptocurrency market as shown by its market capitalization. Over the years the price of bitcoin has had huge fluctuation. As a currency, Bitcoin offers a novel opportunity for price prediction due its relatively young age and resulting volatility, which is far greater than that of fiat currencies [2].

The prediction of mature financial markets such as the stock market has been well researched [3],[4] and these approaches are now being applied to a much younger cryptocurrency market [5],[6]. Some of the works applying deep machine-learning to financial market trading, try to predict the price movements or trends [7], [8] using historic market data. However, to make use of the predicted price or direction of a financial product, a different model would have to be designed that converts the predictions to trading actions which include what and how much to buy/sell in the market. The market can be observed as an environment which has states and actions such as buy, sell, hold, and such action when taken, yield a reward. The problem then can be solved using reinforcement learning. Reinforcement learning [9] approach has been applied in the financial market. In this work both a supervised learning approach and deep reinforcement learning approach are implemented, the former predicts the direction of bitcoin price using historic price, along with selected features, the output is then converted to trading actions.

The contents of this paper are as follows: Section II defines the problem statement. Section III introduces the data accessing and processing steps. The deep Q learning approach will be described in Section IV and the supervised learning approach is described in Section V. Section VI will evaluate both approaches and compare the performances against other trading strategies on the test dataset. Section VII discusses the results. Finally, the conclusion of the can be found in Section VIII.

II. PROBLEM DEFINATION

First, Given a time series of cryptocurrency prices, the return can be maximized by purchasing coins at the right intervals. Making large investments when the price is low can yield significant returns due to the volatile nature of cryptocurrency prices. If P represents the prices of bitcoin between two points in time, P = {p1, p2, p3, …, pn}. Let pk be the lowest price in P. Investing the entire capital at pk will have the largest yield as compared with distributing it over any other combination of prices. The problem this research tries to address is that of deciding whether an investment should be made at a given time interval or not based on the trend in the price.

III. DATA

A. Data Source

Bitcoin Open, High, Low, Close, Volume (OHLCV) data was obtained from Binance. The data is one year and ten months in time span and the trading period is one min. Additional 24 features are computed from the OHLCV values based on Returns, Lagged Returns, Price Level, Change In Price, Acceleration In Price, Volume Level, Change In Volume, Volatility Level, Change In Volatility.

In Table 1, the new features have been selected based on a previous work [10] which showed outstanding results. Return is denoted by r, p is price, and v is volume.

TABLE I. ADDITIONAL COMPUTATED FEATURES

Name	Description	Type
r	Bitcoin return	
r_1	Return from 1 period prior	Lagged Return
r_2	Return from 2 periods prior	Lagged Return
rZ12	Zscore(r,12)	Price Level
rZ96	Zscore(r,96)	Price Level
pma12	Zscore(p/avg(p,12)-1,96)	Change in Price
pma96	Zscore(p/avg(p,96)-1,96)	Change in Price
pma672	Zscore(p/avg(p,672)-1,96)	Change in Price
ma4/36	Zscore(avg(p,4)/avg(p,36)-1,96)	Change in Price
ma12/96	Zscore(avg(p,12)/avg(p,96)-1,96)	Change in Price
ac12/12	Zscore([p/avg(p,12)]/avg[p/avg(p,12),12],96)	Acceleration in price
vZ12	Zscore(v,12)	Volume

978-1-7281-6167-9/20/$31.00 ©2020 IEEE

		level
vZ96	Zscore(v,96)	Volume level
vZ672	Zscore(v,672)	Volume level
vma12	Zscore(v/avg(v,12)-1,96)	Change in volume
vma96	Zscore(v/avg(v,96)-1,96)	Change in volume
vma672	Zscore(v/avg(v,672)-1,96)	Change in volume
vol12	Zscore(std(r,12),96)	Volatility level
vol96	Zscore(std(r,96),96)	Volatility level
vol672	Zscore(std(r,672),96)	Volatility level
dv12/96	Zscore(std(r,12)/avg(std(r,12),96),96)	Change in Volatility
dv96/672	Zscore(std(r,96)/avg(std(r,96),672),96)	Change in Volatility
Ac96/96	Zscore([p/avg(p,96)]/avg[p/avg(p,96), 12],96)	Acceleration in price

B. *Data Preprocessing*

For the deep Q learning approach the one-minute bitcoin time series data is resampled by a frequency of 15 minutes and then the features are calculated. The NaN values are dropped. The dataset is split into training and test sets, with ratio 70:30 respectively. In the supervised learning approach, the one-minute bitcoin time series data is resampled by a frequency of 3 minutes. This produced the best accuracy over 1 minute and 15 minutes frequencies. The additional features are then computed. The target is set to 1 if the bitcoin in the next 5-time step (15 minutes) is greater than the current time step else it is set to 0. All features are normalized using a python library Scikit-learn [11]. The dataset is then split into training and test sets, with ratio 95:5 respectively.

IV. DEEP Q LEARNING

To test the performance of deep Q learning in bitcoin trading, an agent was trained using deep Q learning to predict how much bitcoin to buy under a given condition. The agent can choose to buy anywhere between 0 and 4 bitcoins.

A. *Q Value*

The Q value for a state and action is calculated based on the reward for the action and the q values for future actions. The q value for the last action is the same as the reward for the action. The q values for all actions preceding the last action can be calculated as shown in equation 1. Where r refers to the reward for an action, q_n refers to the q value for the next action and d is the discount factor.

$$q = r+(d*q\_n) \quad (1)$$

B. *Loss Function*

The neural network [12] is trained to directly provide the probabilities of possible actions for a state. A custom loss function was made to train the neural network. If the highest probability of an action predicted by the neural network is a, and q is the q value corresponding to the state given to the neural network, equation 2 can be used to calculate the loss value.

$$loss = -q*\log(a) \quad (2)$$

C. *Reward Function*

The reward of an action is determined based on the rise and fall in prices. If the agent decides to buy and the price goes up, a positive reward is given. Otherwise, a negative reward is given. The reward of an action is based on the following equation 3. In equation 3, a refers to the action performed by the agent. The action specifies the quantity of bitcoin to buy, while cls and opn refer to the closing and opening prices.

$$R = a*(\log(cls/opn)) \quad (3)$$

Table 2 provides the specifications of the neural network used in the Deep Q Learning approach. The neural network has 3 layers, all fully connected. This is implemented using pytorch [13]

TABLE II. DQN MLP MODEL PARAMETER

Parameter	**Value**
Optimizer	Adam
Layers	3
Dropout rate	0.2
Neurons by layer	28,32, 5
Learning Rate	1e-3
Discount Factor	0.9

V. SUPERRVISED LEARNING (LSTM)

Long short-term memory (LSTM) [14] networks excel at learning dependencies in time series data; hence they are well suited to predicting bitcoin prices based on the trend in price. In the LSTM approach, an LSTM neural network is used to predict whether the price of bitcoin is going to go up or down in the next 15 minutes. A bitcoin is purchased if the price is going up.

The LSTM network is trained on price data from the past hour sampled every 3 minutes. The input consists of the open, high, close and low values along with the features defined in the data section for each of the past 20 time steps (1 hour). This is implement using Keras [15] on TensorFlow [16] backend.

Table 3 provides the specifications of the neural network used in the LSTM approach. The neural network has 3 LSTM layers and 2 fully connected layers.

TABLE III. SUPERVISED LEARNING LSTM MODEL PARAMETERS

Parameter	**Value**
Optimizer	Adam
Loss	Sparse categorical entropy
Layers	5
Dropout rate	0.2
Neurons by layer	128, 128, 128, 32, 2

Learning Rate	1e-3
Weight Decay	1e-6

VI. Performance Evaluation

A. Results

The training dataset for the supervised learning LSTM network is of 2017-12-09 to 2019-07-28 and the test dataset from 2019-08-29 to 2019-10-01 while training dataset for the deep reinforcement learning approach is of time span 2017-12-09 to 2019-03-17 and the test dataset 2019-03-17 to 2019-10-01

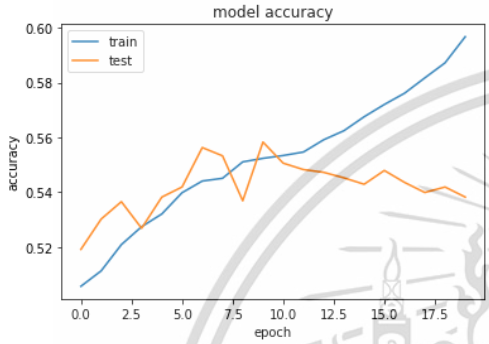

Fig. 1. Model accuracy for 20 epochs (LSTM).

Fig 1 shows the model accuracy for the LSTM model after 20 epochs of training, the test accuracy peaks at epoch 9, though the training accuracy keeps rising, this points towards overfitting. Early stopping is later implemented to prevent such.

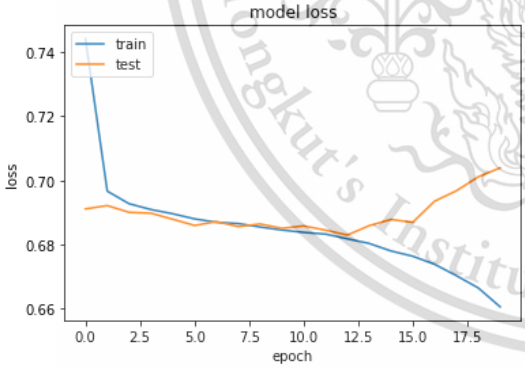

Fig. 2. Model accuracy for 20 epochs (LSTM).

Fig 2 much like fig 1 shows that the LSTM model begins to overfit after epoch 10 as observed by the declining training loss and increasing test loss.

TABLE IV. Confusion Matrix for Supervised Learning(LSTM)

	Predicted True	**Predicted False**
Actual True	4779	2919
Actual False	4076	4125

Table 5 Shows metrics and their values.

TABLE V. SUPERVISED LEARNING LSTM MODEL METRICS

Metric	**Value**
Accuracy	56.0%
Precision	53.95%
Recall	62.07%

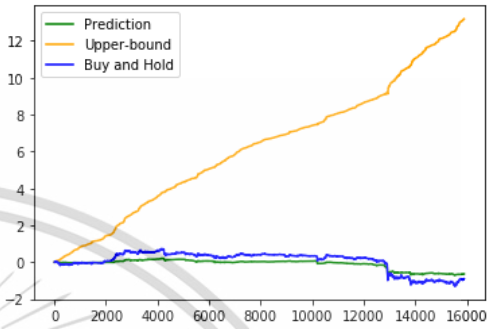

Fig. 3. LSTM (green), buy and hold (blue) and best possible (yellow) returns from 29 augst 2019 to 30 september 2019

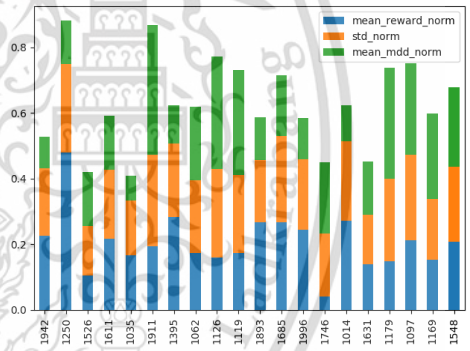

Fig. 4. Graph of 20 agents' test performances between 17 March 2019 and 1 October 2019

The mean reward, standard deviation and mean max draw down of the agents are normalized to fix the same scale.

Buy and hold strategy (BnH): Buys 2 bitcoins at the start of the test period and holds it for the duration.

Momentum strategy (MMT): momentum strategy that will hold 4 bitcoins if the price is above the average price over the previous 30 periods and 0 bitcoins otherwise

Deep Reinforcement Learning (RL): The algorithm choices what amount of bitcoin to buy, this ranges from 0 to 4.
Max Drawdown (MDD): The drawdown is the measure of the decline from a historical peak in returns.

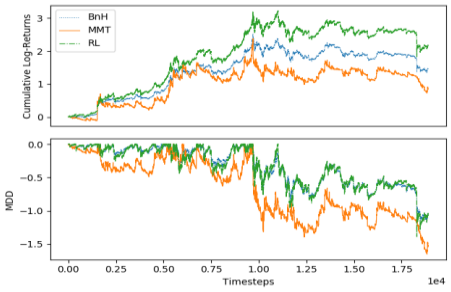

Fig. 5. Confusion Matrix for Supervised Learning(LSTM Aggregated result of the 20 agents

VII. Discussion

The 20 models trained using deep reinforcement learning showed varying degrees of success due to the stochastic nature of the training. Multiple models got stuck in local optima, but those that did not, performed significantly better. The LSTM model had a good prediction accuracy of 56%, and did outperform buy and hold strategy, however the return was poor and was not profitable as bitcoin is on a downward trend. The "upper bound" curve represents the returns possible when the future is known, and coins are bought at the right time step.

VIII. Conclusion and Furture Work

This paper implements supervised learning (LSTM) and deep reinforcement learning addressing the issues of when to buy and what Bitcoin quantity to buy, to maximize return in the long term. The deep learning reinforcement (RL) approach showed the most outstanding performance, though commission is not taken into account in both approaches. That said this could be implemented in the future along with hyperparameter tuning for both approaches, this might help to improve the accuracy for the supervised learning approach and lead to better returns.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] M. Briere, K. Oosterlinck, and A. Szafarz, "Virtual currency, tangible return:Portfolio diversification with bitcoins," Tangible Return: Portfolio Diversification with Bitcoins (September 12, 2013), 2013.I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[3] I. Kaastra and M. Boyd, "Designing a neural network for forecasting financial and economic time series," Neurocomputing, vol. 10, no. 3, pp. 215–236, 1996.

[4] H. White, "Economic prediction using neural networks: The case of ibm daily stock returns," in Neural Networks, 1988., IEEE International Conference on. IEEE, 1988, pp. 451–458.

[5] S. McNally, J. Roche and S. Caton, "Predicting the Price of Bitcoin Using Machine Learning," 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), Cambridge, 2018, pp. 339-343.

[6] L. Alessandretti, A. ElBahrawy, L. M. Aiello and A. Baronchelli: Anticipating cryptocurrency prices using machine learning. ArXiv, abs/1805.08550. (2018)

[7] J. B. Heaton, N. G. Polson, and J. H. Witte, "Deep learning for finance: deep portfolios," Applied Stochastic Models in Business and Industry, 2016.

[8] J. Moody and M. Saffell, "Learning to trade via direct reinforcement," IEEE Transactions on Neural Networks, vol. 12, no. 4, pp. 875–889, Jul 2001

[9] J. E. Moody and M. Saffell. "Reinforcement Learning for Trading." NIPS (1998).

[10] V Poon https://launchpad.ai/blog/trading-bitcoin

[11] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. Vanderplas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," arXiv preprint arXiv:1309.0238, 2013.

[12] Rojas, R. (1996). Neural Networks: A Systematic Introduction, Springer.

[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A.D. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library." NeurIPS 2019.

[14] S. Hochreiter, J. Schmidhuber. "Long short-term memory." Neural computation vol.9 no.8, 1997, pp. 1735-1780.

[15] C. François, "Keras: Deep learning library for theano and tensorflow," URL: https://keras. io/k, 2015.

[16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G S. Corrado, A. Davis, J. Dean, M. Devin, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu and X. Zheng "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," arXiv preprint arXiv:1603.04467, 2016.

Bibliography

[1] K. Sagiraju and S. Mogalla, "Deployment of deep reinforcement learning and market sentiment aware strategies in automated stock market prediction," *International Journal of Engineering Trends and Technology*, vol. 70, no. 1, p. 43–53, 2022.

[2] R. Lippmann, "An introduction to computing with neural nets," *IEEE ASSP Magazine*, vol. 4, no. 2, p. 4–22, 1987.

[3] TradingView, "Bitstamp:btcusd1w." [Online]. Available: https://www.tradingview.com/chart/?symbol=BITSTAMP%3ABTCUSD

[4] T. Starke, "Reinforcement learning for trading – practical examples and lessons learned," 2018.

[5] S. Ouellette, "Reinforcement learning in the presence of nonstationary variables," 2017.

[6] Z. Zhang, S. Zohren, and S. Roberts, "Deep reinforcement learning for trading," 2019. [Online]. Available: https://arxiv.org/abs/1911.10107

[7] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu, "IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures," *CoRR*, vol. abs/1802.01561, 2018. [Online]. Available: http://arxiv.org/abs/1802.01561

[8] M. Sewak, *Actor-Critic Models and the A3C: The Asynchronous Advantage Actor-Critic Model*, 06 2019, pp. 141–152.

[9] M. Mitchell, *An introduction to genetic algorithms*. MIT, 1998.

[10] H.-G. Beyer and H.-P. Schwefel, "Evolution strategies - a comprehensive introduction," *Natural Computing*, vol. 1, pp. 3–52, 03 2002.

[11] M. Srinivas and L. Patnaik, "Genetic algorithms: A survey," *Computer*, vol. 27, no. 6, p. 17–26, 1994.

[12] J. H. Holland, "Genetic algorithms," *Scientific American*, vol. 267, no. 1, pp. 66–73, 1992. [Online]. Available: http://www.jstor.org/stable/24939139

[13] D. Beasley, D. R. Bull, and R. R. Martin, "An overview of genetic algorithms : Part 1, fundamentals," *University Computing*, vol. 15, no. 2, 1993.

[14] M. Kerkhof, L. Wu, G. Perin, and S. Picek, "No (good) loss no gain: Systematic evaluation of loss functions in deep learning-based side-channel analysis," Cryptology ePrint Archive, Paper 2021/1091, 2021, https://eprint.iacr.org/2021/1091. [Online]. Available: https://eprint.iacr.org/2021/1091

[15] A. Mathew, A. Arul, and S. Sivakumari, *Deep Learning Techniques: An Overview*, 01 2021, pp. 599–608.

[16] R. Bertschi, B. Grunder, M. Macdonald, M. Pocinci, and R. Wilhelm, "Technical analysis - explained," 2010.

[17] ANZ, "Dollar cost averaging - moneyworks.co.nz," Sep 2019. [Online]. Available: https://moneyworks.co.nz/site_files/21861/upload_files/anz20997_DollarCostAveragingA5SalesAid_101-20.09.19.pdf?dl=1

[18] H. He, J. Chen, H. Jin, and S.-h. Chen, "Stock trend analysis and trading strategy," vol. 2s006, 01 2006.

[19] K. Bergerson and D. Wunsch, "A commodity trading model based on a neural network-expert system hybrid," in *IJCNN-91-Seattle International Joint Conference on Neural Networks*, vol. i, 1991, pp. 289–293 vol.1.

[20] N. A. Kyriazis, "A survey on efficiency and profitable trading opportunities in cryptocurrency markets," *Journal of Risk and Financial Management*, vol. 12, no. 2, 2019. [Online]. Available: https://www.mdpi.com/1911-8074/12/2/67

[21] F. Fang, C. Ventre, M. Basios, L. Kanthan, D. Martinez-Rego, F. Wu, and L. Li, "Cryptocurrency trading: A comprehensive survey," *Financial Innovation*, vol. 8, no. 1, 2022.

[22] A. Kolková, "Testing ema indicator for the currency pair eur / usd," *International Journal of Entrepreneurial Knowledge*, vol. 5, 06 2017.

[23] B. Huang and Y. S. Kim, "A test of macd trading strategy," 2006.

[24] A. Moghar and M. Hamiche, "Stock market prediction using lstm recurrent neural network," *Procedia Computer Science*, vol. 170, pp. 1168–1173, 2020, the 11th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 3rd International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920304865

[25] S. Varsamopoulos, K. Bertels, and C. Almudever, "Designing neural network based decoders for surface codes," 11 2018.

[26] X.-H. Le, H. V. Ho, G. Lee, and S. Jung, "Application of long short-term memory (lstm) neural network for flood forecasting," *Water*, vol. 11, no. 7, 2019. [Online]. Available: https://www.mdpi.com/2073-4441/11/7/1387

[27] J. Chakole and M. Kurhekar, "Convolutional neural network-based a novel deep trend following strategy for stock market trading," *CEUR Workshop Proceedings*, 2020.

[28] C. Sammut and G. I. Webb, *Encyclopedia of Machine Learning and Data Mining*. Springer US, 2017.

[29] E. Feinberg and A. Shwartz, *Handbook of Markov Decision Processes: Methods and Applications*, 01 2002, vol. 40.

[30] D. B. Rosenfield, R. D. Shapiro, and D. A. Butler, "Optimal strategies for selling an asset," *Management Science*, vol. 29, no. 9, pp. 1051–1061, 1983. [Online]. Available: http://www.jstor.org/stable/2630932

[31] V. Saario, "Limiting properties of the discounted house-selling problem," *European Journal of Operational Research*, vol. 20, no. 2, pp. 206–210, 1985. [Online]. Available: https://www.sciencedirect.com/science/article/pii/037722178590061X

[32] H. Sompolinsky, "Lectures on reinforcement learning," 2017.

[33] R. Sutton and A. Barto, "Reinforcement learning: An introduction," *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 1054–1054, 1998.

[34] H.-D. Lim, D. W. Kim, and D. Lee, "Regularized q-learning," 2022. [Online]. Available: https://arxiv.org/abs/2202.05404

[35] I. Grondman, L. Busoniu, G. A. D. Lopes, and R. Babuska, "A survey of actor-critic reinforcement learning: Standard and natural policy gradients," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 6, pp. 1291–1307, 2012.

[36] T. M. Moerland, J. Broekens, and C. M. Jonker, "Model-based reinforcement learning: A survey," *CoRR*, vol. abs/2006.16712, 2020. [Online]. Available: https://arxiv.org/abs/2006.16712

[37] V. Pong, S. Gu, M. Dalal, and S. Levine, "Temporal difference models: Model-free deep RL for model-based control," *CoRR*, vol. abs/1802.09081, 2018. [Online]. Available: http://arxiv.org/abs/1802.09081

[38] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," *Society for Industrial and Applied Mathematics*, vol. 42, 04 2001.

[39] R. B. Diddigi, P. Jain, P. K. J., and S. Bhatnagar, "Neural network compatible off-policy natural actor-critic algorithm," *CoRR*, vol. abs/2110.10017, 2021. [Online]. Available: https://arxiv.org/abs/2110.10017

[40] R. S. Sutton, S. Singh, and D. Mcallester, "Comparing policy-gradient algorithms," *IEEE Trans. on Systems, Man, and Cybernetics*, 1983.

[41] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[42] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[43] C. Yoon, "Understanding actor critic methods," Jul 2019. [Online]. Available: https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f

[44] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[45] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *CoRR*, vol. abs/1801.01290, 2018. [Online]. Available: http://arxiv.org/abs/1801.01290

[46] CryptoDataDownload, "Btc/usd historical hourly data." [Online]. Available: https://www.cryptodatadownload.com/cdd/Bitstamp_BTCUSD_1h.csv

[47] L. L. X. Yeo, Q. Cao, and C. Quek, "Dynamic portfolio rebalancing with lag-optimised trading indicators using serofam and genetic algorithms," *Expert Systems with Applications*, vol. 216, p. 119440, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417422024599

[48] I. Jericevich, P. Chang, and T. Gebbie, "Simulation and estimation of an agent-based market-model with a matching engine," 2021.

[49] R. Hu and S. Watt, "An agent-based financial market simulator for evaluation of algorithmic trading strategies," 10 2014.

[50] M. Mayer-Krebs, "Paper trading vs live trading: All differences explained – quantitative modelling & research," 2022. [Online]. Available: https://www.qmr.ai/paper-trading-vs-live-trading-all-differences-explained/

[51] K. Suhail, S. Sankar, A. Kumar, T. Nestor, N. Soliman, A. Algarni, W. El-Shafai, and F. Abd El-Samie, "Stock market trading based on market sentiments and reinforcement learning," *Computers, Materials and Continua*, vol. 70, p. 935–950, 09 2021.

[52] R. G. Mantovani, A. L. Rossi, E. Alcobaça, J. Vanschoren, and A. C. de Carvalho, "A meta-learning recommender system for hyperparameter tuning: Predicting when tuning improves svm classifiers," *Information Sciences*, vol. 501, pp. 193–221, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002002551930533X

[53] R. Mantovani, "Use of meta-learning for hyperparameter tuning of classification problems," Ph.D. dissertation, 07 2018.