Comparison of Deep Reinforcement Learning Models for Automated Trading on Heterogeneous HPC System

Nabil Shadman

16 September, 2024

MSc in High Performance Computing with Data Science

The University of Edinburgh

2024

Abstract

This study investigates the application of Deep Reinforcement Learning (DRL) for automated trading on a heterogeneous High-Performance Computing (HPC) system. We implement and optimise two prominent DRL algorithms, Deep Q-Network (DQN) and Proximal Policy Optimisation (PPO), using PyTorch on the Cirrus HPC system. We conduct a comprehensive comparative analysis of these algorithms, considering both trading performance and computational efficiency across CPU and GPU architectures. Through hyperparameter tuning and scalability analysis, we gain valuable insights into the factors influencing their effectiveness in real-world trading scenarios. Our findings reveal that both algorithms can achieve profitable trading outcomes, but they often fall short of the buy-and-hold strategy, particularly in unseen market conditions or when applied to different asset classes. PPO, with its on-policy learning and clipped objective, generally exhibited better generalisation and adaptability compared to DQN, especially when faced with new or similar datasets. Our scalability analysis highlighted the advantages of GPU acceleration, particularly for larger portfolios, with PPO demonstrating superior scalability on GPU hardware. However, both algorithms showed room for improvement in GPU utilisation, suggesting potential for further optimisation. Overall, our research underscores the potential of DRL for automated trading, but also emphasises the need for continued advancements in algorithm design, feature engineering, and optimisation techniques to fully realise its capabilities in real-world trading scenarios.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my deepest gratitude to my mother, father, and sister for their unwavering support and wisdom. Their presence has been instrumental in my journey, and I am forever grateful for their love and encouragement.

I would like to extend my sincere thanks to my advisors, Dr Joseph Lee and Dr Michael Bareford. Dr Lee offered invaluable insights and guidance throughout the project's preparation, design, implementation, and experimentation phases. His experimental approach greatly influenced the direction of my work on this project. Dr Bareford provided actionable feedback on the report, presentation, and code repository, helping to bring the project to completion.

Finally, I am deeply grateful for the camaraderie of my friends. Their shared passions and zest for life, particularly in sports and travel, provided invaluable moments of joy and rejuvenation amidst the rigours of academic and professional life. Their presence served as a constant source of inspiration and motivation, reminding me of the importance of balance and perspective.

Chapter 1

Introduction

In this study, we examine the application of Deep Reinforcement Learning (DRL) in automated trading. We focus on the trading effectiveness and computational efficiency of DRL algorithms when implemented on heterogeneous High-Performance Computing (HPC) systems. By comparing two widely used DRL models, Deep Q-Network (DQN) and Proximal Policy Optimisation (PPO), on the Cirrus HPC system, we aim to highlight the potential of HPC in enhancing the efficiency and robustness of DRL-based automated trading applications [1][2]. We investigate key factors such as resource optimisation, hyperparameter tuning, scalability, and model transferability to evaluate the effectiveness of these methods in real-world investment management applications.

1.1 Background and Motivation

The rise of automated trading systems has revolutionised the financial sector, enabling more efficient trading applications [10]. DRL provides an opportunity to develop advanced models to execute real-time trading decisions. However, these models often require vast computational resources, particularly during training [19]. The computational resources required include significant CPU and GPU power, large memory capacity, and substantial storage to manage complex simulations and datasets [1][23]. HPC systems, such as Cirrus, provide ample computational resources to DRL models for automated trading [2]. This project is motivated by the need to apply HPC resources to improve the performance and scalability of DRL-based trading applications, aiding in more effective investment management.

1.2 Research Question and Objectives

We explore the application of DRL in automated trading, focusing on optimising performance using HPC. The central research question is: *How can DRL algorithms, such as DQN and PPO, when applied to automated trading, be optimised on HPC systems to improve investment performance and computational efficiency?*

The objectives of this study are:

- To implement DQN and PPO algorithms for automated trading using the PyTorch framework.
- To optimise the implementations for both CPU and GPU architectures on Cirrus.
- To compare the performance of these algorithms in terms of trading outcomes and computational efficiency.
- To optimise hyperparameters and analyse the effect of different configurations on trading outcomes and computational efficiency.
- To assess the scalability of both algorithms with increasing number of assets in the investment portfolio.
- To evaluate the transferability of trained models across different datasets.

1.3 Scope and Limitations

We focus on implementing and analysing DQN and PPO algorithms for automated trading using PyTorch on the Cirrus HPC system. The scope includes:

1. Daily equity and exchange-traded fund (ETF) data selected from the United States (US) financial markets.
2. Experiments on both CPU and GPU nodes of Cirrus.
3. Performance analysis considering portfolio value, execution time, and resource utilisation.
4. Scalability tests with varying numbers of assets in the portfolio.
5. Transferability experiments using both similar and different assets.

Limitations include:

1. Use of daily data rather than high-frequency (minute, second, or millisecond) data.
2. Focus on PyTorch implementations, excluding other frameworks.
3. Testing on a specific HPC system (Cirrus), which may not represent all HPC environments.

1.4 Significance and Contributions

We offer practical contributions to the fields of automated trading and HPC. By comprehensively comparing DQN and PPO algorithms for automated trading, we consider both trading performance and computational efficiency on a heterogeneous HPC system (Cirrus), utilising both CPU and GPU architectures [2]. We find actionable insights into the baseline performance of these algorithms and the impact of hyperparameter tuning on trading performance and computational efficiency. We assess the scalability of DRL models with increasing assets in the portfolio. The investigation of model transferability to different datasets, including similar assets and different assets, adds to the practical applicability of the research. Moreover, by demonstrating the implementation and optimisation of DRL algorithms on Cirrus, we provide a framework for future research in applying DRL to financial trading using HPC resources. These contributions can inform academic researchers and industry practitioners in developing, optimising, and deploying DRL-based trading systems that balance trading effectiveness with computational performance on modern HPC platforms.

1.5 Dissertation Structure

We organise this report into six main chapters. Chapter 1 introduces the study, outlining its motivation and objectives. Chapter 2 explores the background of reinforcement learning, deep reinforcement learning, automated trading systems, and HPC systems, alongside a review of related work. Chapter 3 discusses the methods used to implement and optimise DQN and PPO for automated trading, covering the architecture, tools, and evaluation metrics. Chapter 4 presents the experimental design and implementation, detailing the hardware and software setup, hyperparameter tuning, scalability analysis, and transferability analysis. Chapter 5 reports the experimental results, providing a comparative analysis of DQN and PPO models regarding trading performance, execution time, and other vital considerations. Finally, Chapter 6 concludes by summarising the core findings and contributions and recommends potential directions for future research. Additionally, we define the essential concepts of trading, reinforcement learning, and high-performance computing in Appendix A, aimed at an audience unfamiliar with these areas. The code, data, experimental logs, and project notes are available in our associated code repository [30].

Chapter 2

Background

This chapter provides the foundational knowledge for understanding DRL, automated trading, and HPC systems. Additionally, we explore relevant research in these fields to establish the context for our work.

2.1 Deep Reinforcement Learning

Artificial Intelligence (AI) is a branch of computer science focused on creating systems that perform tasks requiring human intelligence, such as problem-solving, learning, perception, and decision-making [5]. Machine Learning (ML), a subset of AI, enables systems to learn patterns from data and make decisions without explicit programming [5]. Reinforcement Learning (RL), a further specialisation within ML, allows agents to make sequential decisions by interacting with an environment to maximise cumulative rewards [5]. Deep Learning (DL), another ML approach, utilises neural networks—models inspired by the human brain's structure, with interconnected layers of nodes that process data and learn patterns [5]. DRL combines RL and DL, allowing agents to tackle complex tasks using neural networks to approximate policies and value functions, enabling practical actions in high-dimensional environments [1].

2.1.1 Fundamentals of Reinforcement Learning

RL is centred on the idea that agents learn by interacting with an environment to achieve goals (refer to Figure 2.1) [6]. The agent receives feedback as a reward signal and learns to maximise cumulative reward through trial and error [6]. The core elements include the state (the current situation the agent observes), the policy (a strategy for choosing actions), the reward signal (feedback from the environment), and the value function (estimating the expected long-term rewards) [6]. These principles form the foundation of RL systems, making them suitable for decision-making tasks under uncertainty.

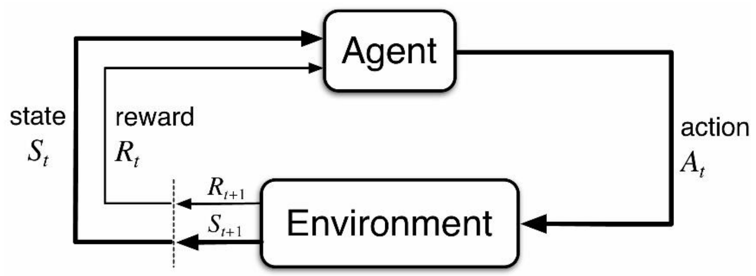

Figure 2.1: The agent-environment interaction in reinforcement learning [6].

Exploration and exploitation trade-off is essential in RL, where agents must balance trying new actions (exploration) with actions they know yield high rewards (exploitation) [6]. This balance allows the agent to learn more about the environment, avoiding locally optimal decisions in favour of potentially better strategies [6].

2.1.2 Deep Learning in Reinforcement Learning

DL becomes integral to RL when dealing with high-dimensional state spaces where the complexity of the environment exceeds simple tabular methods [1]. This is where DRL steps in, using deep neural networks to approximate functions like value functions or policies [1]. For example, DQN approximates Q-values (expected future rewards) for each action, making it possible to apply RL in environments such as image processing, video games, or automated trading, where the state space is vast [1].

Combining RL with deep learning allows for greater flexibility in decision-making, as deep networks can generalise from vast amounts of data, handling complex input spaces that traditional RL methods struggle with [1]. DRL models are particularly effective in environments such as Atari games, robotics, autonomous driving, and automated trading, where input is complex and requires advanced processing [1].

2.1.3 State-of-the-Art Deep Reinforcement Learning Algorithms

Several DRL algorithms represent the cutting-edge research in this field:

- **Deep Q-Network (DQN)**: Introduced by Mnih et al., this algorithm extends Q-learning using deep neural networks [7]. DQN has been vital in mastering Atari games by approximating Q-values for large state spaces [1].

- **Proximal Policy Optimisation (PPO)**: PPO is a policy gradient method that ensures more stable learning by limiting the size of updates, which prevents large, destructive policy changes during training [8].
- **Actor-Critic Methods (A3C, DDPG)**: These algorithms combine value-based and policy-based methods [1]. The actor learns the policy directly, while the critic estimates the value function, leading to faster convergence in continuous action spaces [1].

However, the distinction between value-based, policy-based, and actor-critic algorithms is not mutually exclusive. For example, PPO is both a policy gradient and an actor-critic algorithm, as it directly optimises the policy while using a value function (critic) to evaluate the quality of actions.

2.2 Automated Trading

Automated trading has transformed financial markets, deploying software applications to execute trades with speed and precision far beyond human capabilities [1]. This approach to trading offers various advantages, including reduced human error, faster execution, and the ability to test strategies using historical data (also known as backtesting) [2]. As markets become increasingly complex and data-driven, automated trading systems have progressed to combine advanced techniques such as machine learning [3].

2.2.1 Overview of Automated Trading Systems

Automated trading systems (algorithmic trading systems) are computer programs that create orders and automatically submit them to a market centre or exchange [9]. These systems operate based on a predefined set of rules using parameters such as price, quantity, timing, and other market conditions [9].

The main components of an automated trading system typically include [9]:

- Data handling: Ingesting and processing market data.
- Alpha model: Predicting future price movements.
- Risk model: Assessing and managing potential risks.
- Transaction cost model: Estimating costs associated with trades.
- Execution system: Implementing trading decisions.

2.2.2 Application of Machine Learning in Automated Trading

ML has become increasingly significant in automated trading, extending practical tools for pattern recognition, prediction, and decision-making [10]. Common applications include:

1. Alpha factor creation: ML can help identify new predictive signals or enhance existing ones [11].
2. Portfolio optimisation: Techniques like RL can dynamically adjust portfolio allocations [10].
3. Trade execution: ML models can optimise order routing and execution strategies to minimise market impact [11].
4. Risk management: Advanced ML models can better assess and predict financial risks [10].

However, it is necessary to note that applying ML in finance comes with unique challenges. Financial data is often non-stationary, has a low signal-to-noise ratio, and is susceptible to regime changes [11]. Therefore, robust feature engineering, careful model selection, and rigorous backtesting are essential for successful implementation [10][11].

2.3 High-Performance Computing (HPC) Systems

HPC systems apply parallel processing for complex computations [12]. They supplement deep learning and automated trading by enabling efficient model training and real-time decision-making [11][12].

2.3.1 Introduction to HPC Systems

HPC utilises parallel processing techniques to execute large-scale computational jobs, often requiring the coordination of thousands of cores working simultaneously [13]. Modern HPC systems utilise both multicore processors and specialised accelerators, such as graphics processing units (GPUs), to provide substantial computing power and speed [12]. These systems are essential for solving complex problems in science, engineering, and data analysis, where the sheer volume of calculations necessitates highly optimised parallel algorithms and efficient memory management [12][13]. The architecture and scalability of HPC systems make them suitable for scientific simulations, weather modelling, and financial computations, where processing speed is critical [12][13].

2.3.2 Benefits of HPC in Deep Reinforcement Learning and Automated Trading

HPC provides the computational resources required for efficiently training DRL models and running automated trading algorithms [1][11]. In DRL, large-scale simulations and model training benefit from parallelisation, which accelerates applications by distributing workloads across multiple processors and GPUs [12]. Likewise, in automated trading, HPC enables faster data processing, backtesting, and real-time decision-making, enabling traders to analyse and execute trades with precision [11]. By applying HPC, financial institutions can run more advanced models, optimise algorithms, and stay competitive in dynamic markets.

2.4 Related Work

In this section, we explore related work in the field. We cover the application of DRL in automated trading, the use of parallel computing and HPC techniques for DRL, and identify gaps in existing research.

2.4.1 Deep Reinforcement Learning for Automated Trading

Théate and Ernst present Trading DQN (TDQN), a model adapted from DQN to optimise stock trading positions and maximise risk-adjusted return across various markets [14]. TDQN outperforms benchmark strategies across a test bench of 30 diverse stocks [14].

AbdelKawy et al. propose a synchronous DRL model for multi-stock trading, using advanced approaches for feature extraction and time-series prediction [15]. They applied DQN and Deep Deterministic Policy Gradient (DDPG) algorithms, and the model outperformed traditional approaches in terms of accumulated wealth [15].

Xia et al. discuss a stock trading strategy using PPO on 30 stocks from the US market [16]. The model outperforms a baseline index in terms of annualised return, and incorporating risk-averse behaviour into the reward function further improves performance [16].

Park et al. present a PPO model for stock trading, incorporating both stock data and macroeconomic factors such as interest rates [17]. They demonstrate that including broader market indices and interest rate data alongside stock-specific data improves trading performance [17].

Yang et al. propose an ensemble strategy for stock trading using three DRL models: PPO, Advantage Actor-Critic (A2C), and DDPG [18]. In terms of risk-adjusted returns, the ensemble strategy outperforms individual algorithms and baselines [18].

Ma et al. discuss a Parallel Multi-Module RL (PMMRL) algorithm for stock trading, applying advanced approaches to analyse market and fundamental data and to capture long-term market trends [3]. The PMMRL algorithm significantly outperforms multiple baseline models, achieving higher profits and lower drawdowns on the Chinese stock market [3].

The DRL-assisted techniques mentioned in the studies significantly outperform traditional approaches. For instance, Xia et al., utilising PPO, achieved a 13.8% annualised excess return compared to the Dow Jones Industrial Average (DJIA) [16]. The DJIA is a stock market index that tracks 30 large, publicly traded companies in the US, reflecting overall market performance. Ma et al. achieved an excess cumulative return of 373% on a single stock traded on the Chinese stock market, compared to the buy-and-hold strategy, over a test period of 44 months [3]. Buy-and-hold is a strategy where an investor purchases assets and holds them long-term, regardless of market fluctuations, to benefit from long-term growth. However, DRL models can also face drawbacks, including increased volatility and the potential for significant drawdowns if their risks are not properly managed [16][18].

2.4.2 HPC Systems for Deep Reinforcement Learning

Kopel and Szczurek explore GPU-based environment simulation for parallelising RL algorithms [19]. They compare DQN and PPO algorithms using the CUDA Learning Environment framework. Results show that PPO benefits more from parallelisation, achieving optimal performance with 256 parallel environments, while DQN's performance degrades with more than 128 parallel environments. GPU-only simulation and training outperform CPU-GPU approaches in resource utilisation [19].

Lan et al. present WarpDrive, a framework for running end-to-end multi-agent RL entirely on a GPU [20]. By eliminating CPU-GPU data transfers and maximising parallelisation, WarpDrive achieves significantly faster training compared to CPU-based implementations [20]. Benchmarks show it scales nearly linearly to thousands of environments and agents, enabling high-throughput RL on a single GPU [20].

Baláž et al. introduce a GPU-based implementation of Monte Carlo Tree Search (MCTS), a widely used heuristic search algorithm in model-based RL for improved action selection [21]. Their approach, which fully utilises GPU tensor operations to process multiple trees in parallel, significantly outperforms existing methods, especially when handling numerous observations and simulations [21].

Espeholt et al. present IMPALA, a distributed DRL framework that combines CPUs and GPUs [22]. IMPALA outperforms previous methods on multi-task benchmarks, achieving better performance and data efficiency while scaling to thousands of machines [22].

Nair et al. introduce Gorila, a distributed deep reinforcement learning framework [23]. Applied to DQN, Gorila outperformed single-GPU DQN on most Atari games in half the training time, demonstrating improved performance and efficiency [23].

2.4.3 Gaps in Existing Research

The papers applying DRL models in automated trading utilise advanced techniques in feature engineering, adapt baseline models like DQN and PPO, employ ensemble strategies, or enhance risk management. They considerably contribute to the fields of financial engineering and machine learning. However, these studies typically do not address computational efficiency, parallel or distributed computing, or HPC.

On the other hand, papers exploring parallel computing techniques in DRL focus broadly on CPUs and GPUs, predominantly within the gaming domain. While their insights are valuable, the finance domain presents unique challenges and requirements. Thus, the learnings may not always generalise to the finance domain.

These findings highlight a clear gap in current research: applying parallel and HPC techniques to DRL-based automated trading. This gap presents an opportunity for further exploration, with the potential for advancements in this field.

Chapter 3

Methods

Chapter 3 outlines the methodology for exploring and comparing DRL models for automated trading. We provide a detailed explanation of the algorithms, architectures, and experimental setup used throughout the research. We also cover the implementation framework, the HPC environment configuration, and the performance analysis evaluation metrics. The aim is to present a clear, reproducible methodology that underpins the findings discussed in later chapters.

3.1 Deep Reinforcement Learning Algorithms

We explore two widely used DRL algorithms (DQN and PPO) for their efficacy in automated trading. We selected each algorithm for its suitability of sequential decision-making and its proven success in real-world applications.

3.1.1 DQN: Description and Justification

DQN is a model-free, off-policy algorithm that extends Q-learning by utilising deep neural networks to approximate the Q-value function [1]. Model-free means that DQN learns directly from experiences without building a model of the environment. Off-policy means that DQN can learn from experiences generated by a policy different from the one it is currently improving. DQN's objective is to estimate expected future rewards for each possible action, given a state, and select actions that maximise expected return [1].

DQN was selected for this study because of its robustness in discrete action spaces, making it well-suited to the sequential nature of trading decisions [1]. It also allows for efficient training in complex environments, such as financial markets, where state-action pairs can quickly grow in number [1]. Algorithm 3.1 details DQN's pseudocode [7].

Algorithm 3.1: DQN

Here are the steps of the DQN algorithm:

```
1. Initialise a replay memory to store experience tuples (state, action, reward, next state).
2. Initialise the Q-network with random weights.
3. For each episode:
    1. Reset the environment to the initial state.
    2. Observe the initial state, and preprocess it if necessary.
    3. For each time step in the episode:
        1. Choose an action using an epsilon-greedy policy based on the Q-network output.
        2. Execute the action and observe the reward and the next state.
        3. Store the experience tuple in the replay memory.
        4. Sample a random batch of experiences from the replay memory.
        5. Compute the target Q-values for each experience tuple in the batch using the Bellman equation.
        6. Update the Q-network weights using stochastic gradient descent to minimise the mean squared error between the predicted Q-values and the target Q-values.
        7. Every C steps, copy the Q-network weights to the target Q-network.
    4. End the episode if a terminal state is reached or a maximum number of steps is exceeded.
4. Repeat step 3 for a specified number of episodes.
```

In this algorithm, the Q-network is a neural network that estimates the Q-values of state-action pairs, and the target Q-network is a copy of the Q-network used to compute the target Q-values [7]. The target Q-value is calculated using the reward and the maximum estimated Q-value of the next state. The replay memory stores experience tuples to be randomly sampled for training the Q-network, which improves sample efficiency and reduces the correlation between consecutive experiences [7]. The epsilon-greedy policy balances exploration and exploitation during action selection by choosing a random action with probability epsilon, or the action with the highest estimated value with probability (1 - epsilon) [1].

The Bellman equation is used to compute the target Q-values, which are updated during training to improve the accuracy of the Q-network [7]. For Q-learning, the Bellman equation for the optimal Q-function is a recursive function: **Q*(s,a) = R(s,a) + γ E[max_a' Q*(s',a') | s, a]**, where s is the current state, a is the action, R(s,a) is the immediate reward, γ is the discount factor, and E[max_a' Q*(s',a') | s, a] is the expected maximum Q-value of the next state [7]. This equation states that the optimal Q-value of a state-action pair is the sum of the immediate reward and the discounted expected maximum Q-value of the next state. In DQN, the learning algorithm aims to minimise the difference between the current estimated Q-values and the target Q-values derived from this equation [7]. The C parameter determines how often the Q-network weights are copied to the target Q-network to stabilise the training process [1].

3.1.2 PPO: Description and Justification

PPO is a model-free, on-policy algorithm designed to improve the stability and performance of policy gradient methods [1]. On-policy means PPO learns by interacting with the environment using the same policy it is currently updating, collecting data and improving its decision-making directly from its actions [8]. PPO maintains a balance between exploration and exploitation while ensuring policy updates do not deviate excessively from the previous iteration. This is achieved through a clipped surrogate objective that restricts large updates, making the training process more stable than traditional policy gradient approaches [8].

We chose PPO for this study because of its proven effectiveness in complex, high-dimensional environments [8]. Automated trading requires frequent policy updates in response to volatile market conditions, and PPO's stability and sample efficiency make it particularly suitable for this task [8]. Its capacity for fine-tuning policy updates also provides an edge in ensuring robust performance over multiple trading episodes [17]. Algorithm 3.2 details PPO's pseudocode [8].

Algorithm 3.2: PPO

Here are the steps of the PPO algorithm:

```
1. Initialise a policy network with random weights.
2. Initialise a value network with random weights.
3. For a specified number of iterations or until convergence:
    1. Collect a batch of experiences by running the policy in the environment.
    2. Compute advantages using the value network to estimate the expected return of
       each state.
    3. For a specified number of epochs:
        1. Compute the surrogate objective function, which measures the policy's
           performance improvement.
        2. Calculate the ratio between the new and old policy probabilities for the
           collected experiences.
        3. Apply a clipping function to the ratio to ensure it stays within a specified
           range.
        4. Compute the minimum between the clipped and unclipped ratios, multiplied
           by the advantages.
        5. Calculate the surrogate objective as the average of the minimum values.
        6. Update the policy network using a stochastic gradient descent optimiser to
           maximise the surrogate objective.
        7. Update the value network using a mean squared error loss to minimise the
           difference between the predicted and actual returns.
4. Repeat step 3 for a specified number of iterations or until convergence.
```

In this algorithm, the policy network learns to output a probability distribution over actions for a given state, while the value network estimates the expected return for each state [8]. The probability distribution indicates the likelihood of selecting each possible action in a particular state. The advantages represent the advantage of taking an action compared to the average value of actions in that state [8]. The surrogate objective function measures how the policy performance improves based on the advantages and the ratio between new and old policy probabilities [8]. Clipping the ratio ensures that the policy update does not deviate too much from the old policy, adding stability to the learning process. The policy network is updated to maximise the surrogate objective, while the value network is updated to minimise the difference between predicted and actual returns [8].

3.2 Deep Reinforcement Learning Architectures

We have designed our DQN and PPO architectures to balance complexity, learning capacity, and computational efficiency. While both architectures share some common elements, they also have distinct features that cater to their respective algorithms.

We adopted a single-file implementation for our agents, a practice commonly found in many open-source libraries and research codes [24][25][26]. This approach consolidates the details of a specific algorithm into a single, self-contained file. It serves as an excellent reference for those who prefer not to navigate through the complexities of a fully modular library [24].

The common components in both DQN and PPO architectures include [30]:

1. **Environment:** Both programs use the same multi-asset environment class to simulate a trading environment. The environment manages asset prices and portfolio value and executes trading actions, providing a consistent interface for both algorithms.
2. **Data Handling:** Both codes use identical methods to load asset price data from a CSV file and split it into training and testing sets. This ensures that both algorithms are trained and evaluated on the same data, allowing for fair comparison.
3. **Action Space:** The action space is the same in both programs, allowing three actions (sell, hold, buy) for each asset. The resulting possible actions depend on the number of assets (i.e. 3^N, where N is the number of assets in the portfolio). The discrete action space simplifies the decision-making process while allowing for complex trading strategies.
4. **State Representation:** Both programs use a state representation that includes the number of shares owned for each asset, the current price of each asset, and the available cash. This comprehensive state enables the agents to make informed decisions based on their current portfolio and market conditions.
5. **Reward Function:** In both programs, the reward is the change in portfolio value after taking action. This aligns the agents' objectives with maximising portfolio value, a key goal in trading.
6. **Transaction Costs:** Both programs include a 2% transaction cost rate in the trading process, making the simulation more realistic. This encourages the agents to consider the costs of frequent trading and develop more efficient strategies.

7. **Scaler:** Both programs use a standard scaler to normalise the state space, which helps with training stability by ensuring all input features are on a similar scale.

8. **Configuration:** Both codes use YAML configuration files and command-line arguments for flexible parameter settings. This allows for easy experimentation with different hyperparameters without changing the core code.

9. **Logging and Statistics:** Both programs provide similar logging of episode results and final statistics. The detailed logging enables comprehensive evaluation of the agents' performance.

10. **Common Hyperparameters:** Both implementations share several key hyperparameters:

    - Initial investment: 100,000 US Dollars (USD), providing a considerable starting capital for trading.
    - Discount factor (gamma): 0.99, indicating a strong consideration for future rewards.
    - Learning rate (alpha): 0.0003, determining the step size during optimisation.
    - Batch size: 32, specifying the number of samples used in each training step.

11. **Episode Count:** Both use two episodes for profiling and 100 for performance analysis. This allows for quick testing during development and a more comprehensive evaluation of long-term performance.

3.2.1 DQN Architecture: Design and Rationale

The DQN-specific components include [30]:

1. **Q-Network:** The DQN uses a single Multi-Layer Perceptron (MLP) with two hidden layers (each containing 32 neurons) to approximate the Q-function. This network takes the state as input and outputs Q-values for each possible action [1].

2. **Replay Buffer:** DQN utilises a replay buffer with a size of 500 to store experiences. This buffer helps to decorrelate the training data and improve learning stability by allowing the agent to learn from a diverse set of past experiences rather than just the most recent ones [1].

3. **Epsilon-Greedy Exploration:** The DQN agent uses an epsilon-greedy policy for exploration, with an initial epsilon of 1.0, decreasing to a minimum of 0.01 with a decay rate of 0.995. This strategy balances exploration of new actions with

exploitation of known good actions, gradually shifting towards exploitation as training progresses [1].

4. **Training Process:** DQN performs updates after every step in the environment, sampling from the replay buffer. This allows for frequent updates and efficient use of experiences, potentially leading to faster learning [1].

In the Q-Network, we opted for two hidden layers to provide sufficient depth for capturing complex patterns in financial data without risk of overfitting. The choice of 32 neurons per layer offers a good trade-off between model capacity and training speed. The output dimension corresponds to the number of possible actions. We use ReLU activations for the hidden layers because they effectively mitigate the vanishing gradient problem and promote sparse activations [27]. The relatively compact structure ensures quick inference times, which is vital for real-time trading decisions.

We utilised an open-source DQN-based automated trading implementation by Lazy Programmer (LP) as the foundation for our DQN implementation [25]. LP encourages experimentation with their code, allowing us to develop and customise the implementation for our specific requirements [28]. As the original code already incorporated some components of our intended architecture, we concentrated on enhancing and optimising these existing elements to suit our purpose better. The essential enhancements from the LP implementation include [30]:

- **GPU Support:** Implemented device selection (CPU/GPU) to leverage the performance capabilities of compatible hardware, ensuring data (or tensors) are correctly transferred to the chosen device.
- **Enhanced Environment:** Modified trading environment to include transaction costs in trading decisions, making the simulation more realistic.
- **Model Architecture Improvements:** Increased default hidden layers from 1 to 2 for potentially better model capacity.
- **Customisable Hyperparameters:** Made several parameters configurable, including epsilon decay rate, learning rate, and replay buffer size.
- **Flexible Data Input:** Modified to accept data file path as input, allowing for easier data switching and testing on different datasets.
- **Configuration Management:** Implemented YAML configuration file loading for easier parameter management and experiment tracking.
- **Improved Randomness:** Added seed setting for reproducibility across runs.

- **Portfolio Metrics:** Added calculation and reporting of portfolio statistics (median, min, max). Included median portfolio value of last 30 episodes for trend analysis.
- **Execution Time Tracking:** Added total execution time measurement for performance monitoring.
- **Performance Monitoring and Profiling:** Integrated TensorBoard for real-time training monitoring and PyTorch Profiler for detailed performance analysis, optimising resource usage and training efficiency**.**
- **Hyperparameter Tuning:** Tuned key parameters like learning rate, gamma, and epsilon decay to optimise model performance across datasets.

3.2.2 PPO Architecture: Design and Rationale

The PPO-specific components include [30]:

1. **Actor-Critic Architecture:** PPO uses separate networks for the actor (policy) and critic (value function), each being an MLP with two hidden layers (with 32 neurons in each layer).

2. **Proximal Policy Updates:** PPO uses a clipped surrogate objective with an initial clip parameter of 0.2 to update the policy.

3. **Advantage Estimation:** PPO uses Generalized Advantage Estimation (GAE) with a lambda value of 0.95 to compute advantages. GAE helps balance the trade-off between bias and variance in policy gradient estimates, potentially leading to more stable and efficient learning [8].

4. **On-Policy Learning:** PPO is an on-policy algorithm that learns from recent experiences, updating every predetermined number of steps [8].

5. **Training Process:** PPO performs episodic updates, conducting predetermined number of epochs of training on each batch of collected data. This allows the algorithm to extract more information from each batch of experiences, potentially leading to more sample-efficient learning [8].

Our PPO implementation uses separate actor and critic networks. The actor outputs action probabilities, with an input layer matching the state space, two hidden layers of 32 neurons each with ReLU activations, and a softmax-activated output layer. Softmax is an activation function that converts the actor network's raw output values into a probability distribution over possible action [27]. The critic network, which estimates the value function, follows a similar structure but ends with a single output neuron for state value estimation. This dual-network design enables simultaneous learning of both the policy and value function, promoting stable learning and potentially more refined trading strategies. GAE reduces variance in policy gradient estimates, while the clipping mechanism prevents overly large updates [8].

We built our PPO-based automated trading system on Phil Tabor's open-source PPO agent implementation [26]. Tabor's code is designed for experimentation, giving us the flexibility to tailor it to our needs [29]. We focused on refining and optimising the existing elements to align with our architecture, as the original code already included some of our desired components. Key enhancements from Tabor's implementation include [30]:

- **Environment Integration:** Integrated the multi-asset environment class with transaction cost modelling (to match that of DQN), allowing the PPO agent to interact with a realistic trading environment.
- **Episodic Training/Testing:** Implemented a function to facilitate episodic training and testing.
- **Customisable Hyperparameters:** Made several key parameters configurable, including learning rate, GAE lambda, policy clip, and batch size.
- **Flexible Data Input:** Modified to accept data file path as input, allowing for easier data switching and testing on different datasets.
- **Configuration Management:** Implemented YAML configuration file loading for easier parameter management and experiment tracking.
- **Improved Randomness:** Added seed setting for reproducibility across runs.
- **Portfolio Metrics:** Added calculation and reporting of key portfolio statistics (median, min, max, last 30 episodes).
- **Execution Time Tracking:** Added total execution time measurement for performance monitoring.
- **Performance Monitoring and Profiling:** Integrated TensorBoard for real-time training monitoring and PyTorch Profiler for detailed performance analysis.
- **Hyperparameter Tuning:** Tuned key parameters like learning rate, epohcs, and policy clip to optimise model performance across datasets.

3.2.3. Correctness Testing

Our correctness testing approach for the trading environment and RL agents (DQN and PPO) is designed to verify the fundamental components of the system before engaging in full-scale training and evaluation [30].

For the trading environment, we use a script to initialise the environment with sample data and execute a predefined sequence of actions, including buying, holding, and selling assets [30]. This test allows us to verify that the environment correctly handles state transitions, calculates rewards, and updates portfolio values in response to different actions.

The DQN agent testing, implemented in a separate script, focuses on initialising the agent and environment, then testing the agent's action selection process [30]. We verify the agent's response to a single state-action-reward cycle, ensuring that the core mechanisms of the DQN algorithm are functioning as expected.

Similarly, for the PPO agent, another script sets up the agent and environment, then tests the agent's action selection and value estimation capabilities [30]. We again verify the agent's response to a single state-action-reward cycle, confirming that the essential components of the PPO algorithm are correctly implemented.

This testing approach allows us to quickly verify the correct implementation of core functionalities across the environment and both RL algorithms. By focusing on these essential components, we can identify and address any issues before proceeding to more time-consuming full training episodes and comprehensive evaluations.

We conducted our tests across two distinct environments: a Windows 10 Pro device and the compute nodes (both CPU and GPU) of the Cirrus HPC system, which operates on CentOS 8 Linux [2]. The hardware specifications for the Windows 10 Pro device can be found in Table 3.1, while Table 3.3 provides the corresponding details for the Cirrus system. Our testing methodology guarantees compatibility across these hardware environments and operating systems.

Component	**Specification**
CPUs	1 x Intel Core i7-8550U
Cores per CPU	4
CPU core clock	1.8 GHz
Memory	16 GB (15.7 GB usable)

Table 3.1: Hardware details of Windows 10 Pro device [31].

3.3 Implementation Framework

Our implementation framework was carefully chosen to provide a robust, efficient, and flexible environment for developing and testing our DRL models. We prioritised tools and technologies that offer high performance, good community support, and seamless integration with high-performance computing systems.

3.3.1 Programming Languages and Libraries

Python served as our primary programming language due to its versatility and extensive ecosystem of scientific computing libraries [32]. For our DRL implementations, we leveraged PyTorch 1.13.1, a dynamic computational graph framework that offers both ease of use and high performance [33]. PyTorch's native support for GPU acceleration via CUDA 11.6 was particularly valuable for our HPC endeavours.

Some essential libraries utilised in our project include:

1. NumPy and Pandas: For efficient numerical computations and data manipulation [32].
2. Scikit-learn: Used primarily for data preprocessing, particularly the StandardScaler for normalising financial data [32].
3. Matplotlib: For generating visualisations of simulations [32].
4. YFinance: To fetch historical financial data for trading simulations [34].

In the early phase of our research, we conducted a comprehensive analysis of various RL libraries and ML frameworks. We selected PyTorch for its adaptability and greater control over implementation details. While other libraries offered useful features, they lacked the same level of customisation or were relatively new. PyTorch's familiarity and maturity made it a more suitable choice for our needs, particularly for debugging and scaling our solution. This approach gave us deeper control and insight into our models, allowing us to implement custom optimisations specifically tailored to our trading scenario. Table 3.2 provides a concise summary of the ML frameworks and RL libraries we explored in the study.

Library/Framework	Description	Chosen	Reason
TensorFlow	ML framework with RL support via TensorFlow Agents	No	Focus shifted to PyTorch due to performance differences observed in feasibility study
OpenAI Baselines	Implements popular RL algorithms	No	In maintenance mode, lacks comprehensive documentation
Stable Baselines	Simplified interface built on OpenAI Baselines	No	Supports only older TensorFlow versions
Stable Baselines 3	PyTorch-based version with better maintainability	No	Imports from other library parts, complicating full implementation understanding
CleanRL	Lightweight RL implementations focusing on simplicity	No	Single-file implementations but fewer comments
RLlib (Ray)	Distributed RL library for scalability	No	Highly modular with many imports from other modules
TorchRL	PyTorch-based RL library with parallel environment support	Considered	Chosen for prototyping, offers proof-of-concept for future studies
PyTorch	General-purpose DL library	Yes	Allows custom implementations with full control, enabling fair comparisons

Table 3.2: Summary of frameworks and libraries explored in the project [24][33][35][36][37][38][39][40].

In the initial stage of the project, we utilised TorchRL to prototype its application in automated trading research. This involved adapting our existing PyTorch-based environment to the TorchRL framework and ensuring its accuracy. We developed preliminary versions of both DQN and PPO agents within this library, which need further debugging for correctness. The environment and agents are designed to handle multiple assets concurrently, incorporating transaction costs and variable action spaces. However, completing these implementations, along with comprehensive testing for correctness and performance, falls outside the scope of this project. This prototype serves as a proof of concept, showcasing TorchRL's potential for RL-based automated trading. Further implications and potential future developments are explored in the concluding chapter.

3.3.2 Tools and Technologies

Our development and experimentation process were supported by a range of tools and technologies:

- Bash: Bash scripts were used to automate tasks such as cleaning up old results, generating configuration files, and organising output data, ensuring efficient and consistent execution of experiments across different setups.
- Version Control: We used Git for version control, with our repository hosted on the university's GitLab instance. This facilitated collaborative development and maintained a clear history of our project's evolution.
- Conda: We managed our Python environments using Conda, ensuring consistency across different development and execution environments.
- Cirrus HPC Cluster: Our primary computation platform was the Cirrus HPC cluster. We utilised both its CPU nodes and GPU nodes for our experiments.
- SLURM: We used the SLURM workload manager to schedule and manage our jobs on Cirrus, allowing for efficient utilisation of the cluster's resources.
- Profiling Tools: To optimise our code, we employed PyTorch's built-in profiler for detailed performance analysis.
- TensorBoard: We used TensorBoard (for PyTorch) for visualising training metrics and model graphs.

After evaluating various profiling tools, including PyTorch Profiler, NSight Systems, and DLProf, we selected PyTorch Profiler for its seamless integration with PyTorch, ease of use, and comprehensive profiling features. It provides detailed insights into CPU and GPU performance, and its tracing functionality offers a timeline view, allowing us to track execution over time and identify bottlenecks more effectively.

3.4 HPC System Implementation

The execution of our DRL programs on a HPC system was vital for conducting large-scale experiments and optimising performance.

3.4.1 HPC System Architecture and Configuration

Our primary HPC platform, Cirrus, offers a heterogeneous computing environment that we utilised for different aspects of our project. We used the standard CPU compute nodes for data preprocessing and CPU-based training and inference runs. We used the GPU compute nodes for accelerated training and inference tasks. We utilised the 24 PiB Lustre parallel file system for high-performance data access, which was beneficial for handling our financial datasets [2]. To manage our jobs on compute nodes, we used the SLURM workload manager, crafting job scripts that specified our resource requirements and execution parameters. Table 3.3. details the hardware specifications of Cirrus.

Component	Compute nodes	GPU nodes
Nodes	280	36 (+2 older nodes)
GPUs	-	4 NVIDIA Tesla V100-SXM2-16GB
CPUs	2 x Intel Xeon E5-2695	2 x Intel Xeon Gold 6148 (on 36 nodes), 2 x Intel Xeon Gold 6248 (on 2 nodes)
Cores per CPU	18	20
CPU core clock	2.1 GHz	2.5 GHz (on 36 nodes), 2.4 GHz (on 2 nodes)
Memory	256 GB (2x128) main memory	384 GB (2x192) main memory (on all nodes), 64 GB (4x16) GPU memory (on all nodes)

Table 3.3: Hardware details of Cirrus [2].

3.4.2 Optimisation Techniques for Deep Reinforcement Learning on HPC

To utilise the power of the HPC system, we implemented several optimisation techniques:

1. **GPU Acceleration:** We adapted our PyTorch code to run on GPUs, which significantly sped up both training and inference. This involved ensuring all tensors and models were moved to the GPU.

2. **Efficient Data Structures and Vectorised Operations:** Our implementation utilises NumPy arrays, custom memory buffers, and compact action space representations for efficient data handling. We leverage vectorised operations throughout, including in state normalisation, reward calculations, and neural network computations, significantly enhancing computational performance.

3. **Checkpointing:** We implemented model checkpointing to safeguard against potential job interruptions, a common concern in shared HPC environments.

4. **Profiling and Monitoring:** We used PyTorch's built-in profiler to identify and address performance bottlenecks. We also implemented logging to track GPU utilisation and memory usage throughout our runs.

5. **Hyperparameter Tuning:** We developed scripts to automate hyperparameter searches, taking advantage of Cirrus's several nodes to explore a range of model configurations.

3.5 Dataset and Preprocessing

The financial dataset used in this project consists of equity (stock) and exchange-traded fund (ETF) data sourced from Yahoo Finance via the yfinance API [34]. An ETF is a type of investment fund that holds a collection of assets, such as stocks or bonds, and is traded on stock exchanges like a single security [9]. The dataset encompasses daily prices for multiple equities and ETFs over six years (2018-2023), used to train and evaluate the DRL models. Data preprocessing is essential to ensure the quality and consistency of the dataset before it is fed into the models.

3.5.1 Dataset Description and Sources

Our dataset comprises daily closing prices of equities and ETFs from various sectors in the US financial markets. We initially focused on three equities for our baseline experiments: Apple (AAPL), JPMorgan Chase (JPM), and Walmart (WMT). This selection provides a diverse representation of the technology, financial, and retail sectors.

In our scaling experiments, we expanded our dataset to encompass up to 10 equities, covering a period from January 2, 2018, to December 30, 2023. This six-year timeframe captures diverse market conditions, including the volatility triggered by the COVID-19 pandemic. Additionally, for our transferability experiments, we incorporated data from three ETFs in the US commodities markets—United States Oil Fund, SPDR Gold Trust, and United States Natural Gas Fund—to evaluate how our trained models performed across different asset classes.

3.5.2 Data Preprocessing Techniques

To prepare our dataset for the DRL models, we employed several preprocessing techniques:

1. **Standardisation:** We used scikit-learn's StandardScaler to normalise the asset prices, ensuring that all features were on a similar scale. This step is vital for the effective training of neural networks.
2. **Time series splitting:** We divided our dataset into training and testing sets, using the first 50% of the data for training and the remainder for testing. This chronological split maintains the temporal nature of financial data and ensures fair comparison of training and inference execution times.

3. **State space construction:** For each time step, we constructed a state vector comprising the number of shares owned for each asset, the current asset prices, and the available cash.
4. **Action space definition:** We defined a discrete action space representing different combinations of buy, hold, and sell decisions for the assets in our portfolio.

3.6 Evaluation Metrics

The evaluation metrics provide a comprehensive view of our models' performance, not just in terms of trading effectiveness, but also computational efficiency and resource utilisation. They allow us to make informed comparisons between our DQN and PPO implementations and assess the scalability of our approach in an HPC context.

3.6.1 Portfolio Value: Description and Justification

Portfolio value, measured in USD, is our primary metric for assessing the financial performance of our trading agents. It represents the total value of the agent's holdings (cash and asset values) at each time step. We track this metric throughout training and testing, calculating statistics such as final, mean, median, and maximum values. This metric directly reflects the agent's ability to make profitable trading decisions and is the most relevant measure from an investor's perspective.

3.6.2 Execution Time: Description and Justification

Execution time (in seconds) measures the duration of training and inference for our models. We record this for both entire runs and individual episodes. This metric is vital for assessing the computational efficiency of our algorithms and their suitability for real-time trading scenarios. It also allows us to compare the performance benefits of our HPC implementation.

3.6.3 Memory Consumption: Description and Justification

We monitor both CPU and GPU memory usage (in Megabytes) throughout our experiments. This metric is vital for understanding the scalability of our models and ensuring efficient use of computational resources. It helps us identify potential memory bottlenecks and optimise our implementations for the HPC environment.

3.6.4 Energy Consumption: Description and Justification

Energy consumption (in joules) is measured to assess the environmental impact and operational cost of our models. We use Cirrus's built-in energy monitoring capabilities to track this metric. In an era of increasing focus on sustainable AI, this metric provides valuable insights into the energy efficiency of our DRL approaches.

3.6.5 Power Consumption: Description and Justification

Closely related to energy consumption, power consumption (in watts) gives us a more granular view of our models' resource usage over time. The measurement enables us to identify power usage spikes during different phases of training and inference. This metric is particularly relevant for understanding the demands our models place on the HPC infrastructure.

3.6.6 GPU Utilisation: Description and Justification

GPU utilisation, measured as a percentage, indicates how effectively our models are leveraging the available GPU resources. We use NVIDIA's system management interface (nvidia-smi) to collect this data. High GPU utilisation generally indicates efficient use of resources, while low utilisation might suggest opportunities for optimisation or increased parallelism. This metric is vital for ensuring we are making the most of the powerful V100 GPUs available on Cirrus.

Chapter 4

Experiment Design and Implementation

This chapter presents the design and implementation of experiments conducted to evaluate the performance of DQN and PPO programs. It covers the setup on the Cirrus HPC system, use of CPU and GPU nodes, and key metrics such as portfolio value and execution time. The chapter also details the configurations and profiling tools used to ensure robust and reproducible results.

4.1 Experimental Setup

The experiments were conducted on the Cirrus HPC system, utilising both CPU and GPU nodes to evaluate the performance of the DRL programs. This section outlines the hardware and software configurations used to ensure reproducibility in the experimental results.

4.1.1 Hardware Configuration

The experiments were executed on Cirrus, which is equipped with both CPU and GPU nodes. The CPU nodes consist of two 18-core Intel Xeon E5-2695 v4 (Broadwell) processors with 256 GB of RAM per node [2]. The GPU nodes consist of two 20-core Intel Xeon Gold 6148 (Cascade Lake) processors with 384 GB of RAM per node, along with four NVIDIA Tesla V100-SXM2-16GB GPUs per node [2]. This setup allowed for a comparative analysis of performance across different hardware environments.

4.1.2 Software Configuration

The software stack was built on PyTorch version 1.13, ensuring compatibility with the hardware available on Cirrus. CUDA version 11.6 was employed to utilise GPU acceleration in DRL workloads. Profiling was performed using the PyTorch Profiler for detailed tracing and performance analysis. The experimental setup also involved using SLURM scripts to manage job submissions across the CPU and GPU nodes [30]. Configuration files in YAML format were used to manage hyperparameters and training settings consistently across different experimental runs.

4.2 Experiment 1: Baseline Performance Analysis

This experiment aims to establish a baseline for comparing the performance of the DQN and PPO algorithms in an automated trading environment. The objective is to evaluate each model's efficiency in terms of portfolio performance, execution time, and resource utilisation. This experiment provides insights into the initial performance of the models without extensive optimisation, forming the foundation for further enhancements.

4.2.1 Experimental Design

The design captures performance variations due to differences in hardware, with GPU nodes expected to achieve faster execution times through parallelisation. The evaluation focuses on how well the models scale with available resources and identifies bottlenecks in the initial implementations. The experiment assesses the DQN and PPO models under default configurations. Metrics such as portfolio value, execution time, memory consumption, and energy usage were collected during both training and inference.

The default hyperparameters for DQN and PPO agents were chosen from common values observed in research codes and open-source RL libraries [7][8][40]. We kept common hyperparameters, such as gamma (discount factor), alpha (learning rate), and batch size, equal across both DQN and PPO implementations for a fair comparison. We set gamma at 0.99, alpha at 0.0003, and batch size at 32 state-transitions. Additionally, the neural network architectures of both implementations were standardised with two layers (each with 32 neurons) for a fair comparison. The hyperparameters were kept constant across runs to ensure comparability. Tables 4.1, 4.2, and 4.3 detail the common, DQN-specific, and PPO-specific hyperparameters used in the baseline experiments.

Hyperparameter	Value
Number of assets	3
Initial investment ($)	100000
Transaction cost rate	0.02
Batch size	32
Discount factor (gamma)	0.99
Learning rate (alpha)	0.0003

Table 4.1: Common hyperparameters of DQN and PPO in baseline experiments.

Hyperparameter	Value
Replay buffer size	500
Initial epsilon	1
Minimum epsilon	0.01
Epsilon decay rate	0.995

Table 4.2: DQN-specific hyperparameters in baseline experiments.

Hyperparameter	Value
Steps between learning updates (N)	128
GAE lambda	0.95
Policy clip	0.2
Number of epochs	4

Table 4.3: PPO-specific hyperparameters in baseline experiments.

4.2.2 Implementation

First, we profiled the programs with tracing enabled for two episodes, allowing us to visualise the timeline and identify potential bottlenecks. Next, we profiled the programs over 10 episodes without tracing to tabulate the most time-consuming functions. The chosen episode count balances the overhead of profiling with obtaining reliable estimates. Profiling was conducted once, as the results were already reliable.

We then ran the programs for 100 episodes to measure portfolio values. Following this, we executed 10 episodes with the profiler to measure memory consumption, as profiling is time intensive. Running 10 episodes provided a reliable estimate. Energy consumption data from the 100-episode runs was captured using the Slurm accounting database (sacct) on Cirrus. Additionally, we ran the programs for 100 episodes with nvidia-smi capturing GPU metrics (power, utilisation) every 5 seconds. Each configuration was run three times to ensure reliable estimates.

The models were executed using SLURM scripts, ensuring consistent job scheduling and resource allocation across runs. Each algorithm's script was configured with a YAML file to define the hyperparameters [30]. Separate SLURM scripts were created for CPU and GPU nodes, tailoring the execution for each hardware type. The baseline experiments used a dataset of three equities—Apple (AAPL), JPMorgan Chase (JPM), and Walmart (WMT)—spanning six years (2018–2023). The dataset was evenly split 50-50 between training and testing, providing a fair comparison of execution times. In real-world trading, a 60-40 split is more typical [11].

4.3 Experiment 2: Hyperparameter Optimisation

This experiment focused on tuning key hyperparameters of the DQN and PPO algorithms to enhance their performance in the trading environment. The optimisation aimed to improve the agent's portfolio value and execution time. Through a systematic search, we evaluated multiple configurations to identify the optimal balance between financial and computational performance for both algorithms. Our main goal was to find hyperparameter sets for each algorithm that outperform our baseline experiment's starting parameters and are better optimised for an automated trading environment.

4.3.1 Experimental Design

We used grid search to optimise hyperparameters, testing various combinations. This technique exhaustively evaluates all specified hyperparameter combinations to determine the best-performing model configuration [27]. Tables 4.4 and 4.5 show the search spaces for DQN and PPO respectively, including expected impacts. The parameters tested for each algorithm were selected based on our literature review, focusing on those expected to most significantly impact performance [7][8]. We selected value ranges based on common practices observed in studies and open-source projects [25][26][40].

To gauge parameter effects, we ran multiple training episodes per configuration on both CPU and GPU nodes of Cirrus. This allowed us to compare performance across hardware setups. We tracked key metrics like portfolio value and execution time to guide our selection of the best configuration.

Hyperparameter	**Values**	**Impact**
Learning rate	0.0001, 0.0003, 0.0010	Smaller to larger Q-value updates
Gamma	0.90, 0.95, 0.99	Short-term to long-term reward emphasis
Epsilon decay	0.990, 0.995, 0.999	Quicker to slower transition from exploration to exploitation

Table 4.4: DQN-specific hyperparameter tuning search space.

Hyperparameter	Values	Impact
Learning rate	0.0001, 0.0003, 0.0010	Conservative to aggressive policy/value function updates
Epochs	4, 8, 16	Fewer to more policy update iterations per batch
Policy clip	0.1, 0.2, 0.3	Tighter to looser policy updates

Table 4.5: PPO-specific hyperparameter tuning search space.

4.3.2 Implementation

We implemented separate SLURM scripts for DQN and PPO, running them on both CPU and GPU nodes. Bash scripts automated the creation of tuning folders and job submissions to Cirrus, generating configuration files for each set of hyperparameters [30]. Each SLURM script within a folder ran the agent with specific hyperparameters and recorded performance metrics. This automated approach enabled efficient testing of multiple configurations and ensured reproducibility across hardware setups.

We ran these experiments for 60 episodes, balancing learning time with efficient use of Cirrus's compute resources. For portfolio value, we focused on the median of the last 30 episodes, neglecting initial poor performance and concentrating on results after some learning. Each configuration was run three times for reliability. We then collated the results to determine the most effective parameter set for each algorithm.

4.4 Experiment 3: Scalability Analysis

This experiment aims to evaluate the scalability of the DQN and PPO algorithms when applied to varying numbers of assets in the trading environment. The experiment explores how increasing the number of assets in the input data affects execution time. This experiment relates to weak scaling in HPC, where problem size increases proportionally with resources to assess performance effects [12]. This analysis is vital for determining the robustness of each algorithm when handling larger datasets, thereby simulating a more complex trading environment.

4.4.1 Experimental Design

The experiment was designed to test four different configurations. Assets were increased from 1 to 3, 5, and 10, roughly doubling each time. Beyond 10, at 20, we hit the single GPU memory limit, so we kept it at 10 for the study. This is because the action space in the models is 3^N, requiring significantly more memory with each increment of assets. Table 4.6 provides a breakdown of the portfolio compositions for each of the configurations used in the experiment.

Number of Stocks	Equity Tickers
1	BA
3	BA, CAT, CVX
5	BA, CAT, CVX, DIS, GE
10	BA, CAT, CVX, DIS, GE, HON, IBM, JNJ, KO, MMM

Table 4.6: Portfolio configurations for scalability testing.

For both DQN and PPO, the experiment evaluated execution time. By running the experiments on both CPU and GPU nodes of the Cirrus HPC system, the performance differences between these hardware configurations were also assessed. Each test was executed over 20 episodes for a reliable estimate while conserving Cirrus compute resources.

The best hyperparameters from the tuning experiment were selected and replaced the older hyperparameters used in the baseline experiments to improve performance. Table 5.3 provides details of the optimal hyperparameter sets for the models.

4.4.2 Implementation

For both DQN and PPO, we set up four configurations representing varying equity sizes. The datasets were downloaded from Yahoo Finance API using a Python script and pre-processed for the experiment [30]. We generated four configuration files for each algorithm, corresponding to different datasets. SLURM scripts managed execution on both CPU and GPU nodes of Cirrus. We automated folder generation and result storage using a setup script, ensuring organised data collection for analysis [30].

The experimental setup was identical for both algorithms to allow direct comparison of scalability. Each configuration ran three times for reliability. Detailed logs, including execution time, were produced for each test, providing insights into how the algorithms handled increasing input complexity. We then analysed the results to identify potential bottlenecks and performance differences as the number of equities increased.

4.5 Experiment 4: Robustness and Generalisation

Experiment 4 focused on evaluating the robustness and generalisation capabilities of the DQN and PPO models by training them on one set of assets and then testing their performance on different, unseen sets. The objective was to assess how effectively the models could adapt to varying asset compositions. This experiment specifically aimed to evaluate the transferability of the trained models, both within the same asset class and across different asset classes.

4.5.1 Experimental Design

In this experiment, three datasets were utilised: a baseline equities dataset, a similar equities dataset, and a commodities ETFs dataset. Table 4.7 details the assets used in the experiment. The baseline and similar equities were selected consistently across the technology (Apple, NVIDIA), finance (JPMorgan, Bank of America), and retail (Walmart, Costco) sectors. The commodity ETFs represent three of the most widely traded commodities in the US: oil, gold, and natural gas [41].

Dataset Type	Assets (Tickers)
Baseline Equities	Apple Inc. (AAPL), JPMorgan Chase & Co. (JPM), Walmart Inc. (WMT)
Similar Equities	NVIDIA Corporation (NVDA), Bank of America Corporation (BAC), Costco Wholesale Corporation (COST)
Commodities ETFs	United States Oil Fund, LP (USO), SPDR Gold Shares (GLD), United States Natural Gas Fund, LP (UNG)

Table 4.7: Asset selection for evaluating model robustness and generalisation.

The models were trained and tested on different train-test splits of the baseline equities dataset to evaluate their performance on unseen data. Subsequently, the trained models were tested on the similar equities and commodities ETFs datasets. Testing on similar equities assessed performance within the same sectors, while commodities ETFs evaluated performance across different asset classes. The optimal hyperparameters identified from tuning experiments were employed for each model to enhance performance.

4.5.2 Implementation

We downloaded the data from the Yahoo Finance API using our Python script and prepared it for the experiments. Both DQN and PPO models were executed with their respective scripts, using separate YAML configuration files to specify the different datasets [30]. SLURM scripts managed job submissions on CPU nodes, while a shell script automated the cleanup of intermediate files. Since the focus was on model transferability, only the CPU nodes were used in this experiment. Each configuration was run three times to ensure reliable data collection. The collected portfolio values were then analysed to assess model generalisation and robustness.

Chapter 5

Results

Chapter 5 presents the results of four key experiments evaluating the performance of DQN and PPO models. These experiments include an assessment of baseline performance, hyperparameter optimisation, scalability analysis, and robustness to unseen data. The chapter provides quantitative results, visual interpretations, and a comparative analysis, highlighting insights and implications for model performance across different configurations.

5.1 Experiment 1: Baseline Performance Results

In this section, we discuss the results of our baseline experiments, providing a comprehensive comparison of DQN and PPO across several key dimensions.

5.1.1. Profiling Results

In this subsection, we analyse the results obtained from profiling our DQN and PPO models.

The profiling logs for the **DQN** code on a **CPU** node reveal interesting patterns across training and testing phases. In the training phase, the most CPU-intensive operation was **Optimizer.step#Adam.step** (updating model parameters), consuming about 18% of self-CPU time and 41% of total CPU time [42]. Self-CPU time refers to the time spent executing a specific operation itself, while total-CPU time includes both the self-time, and the time spent in any child operations called by that operation. This operation was followed by **aten::linear** (matrix multiplication for neural network layers) and **aten::addmm** (matrix multiplication with addition), which together accounted for nearly 20% of self-CPU time [42]. Memory-wise, **aten::linear** and **aten::addmm** were the top consumers, each using about 252 MB. **Figure 5.1** shows the timeline view of a section of the execution of a step, highlighting **Optimizer.step#Adam.step** as a dominant operation.

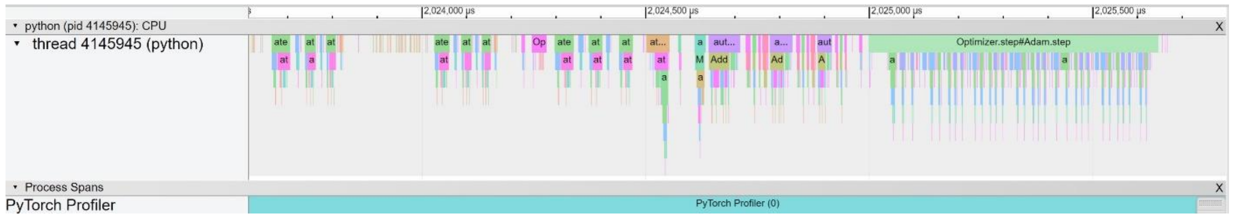

Figure 5.1: Timeline view of a single step during the DQN training phase on a CPU node.

For the testing phase, the picture shifts slightly. **Aten::linear** and **aten::addmm** remain significant, but consume a larger proportion of CPU time (about 11% and 34% of self-CPU time, respectively). This makes sense, as the testing phase does not include the costly parameter updates of training. Memory usage during testing was much lower overall, with **aten::linear** and **aten::addmm** each using only about 2.6 MB. **Figure 5.2** shows the timeline view of a section of the execution of a step, with **aten::linear** and **aten::addmm** as two dominant operations.

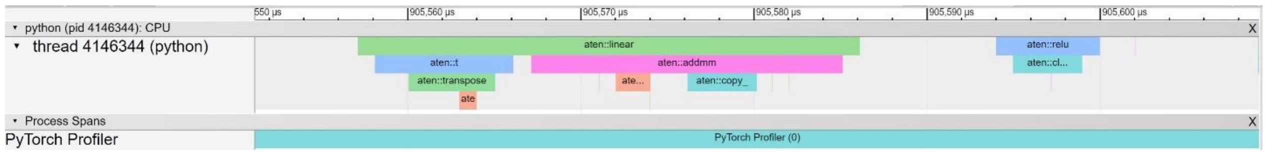

Figure 5.2: Timeline view of a single step during the DQN testing phase on a CPU node.

These results suggest that optimising matrix operations and the Adam optimiser step could yield the most significant performance improvements, especially for the training phase. The memory usage difference between training and testing also highlights the substantial memory overhead of maintaining gradients and optimiser states during training.

The profiling logs for the **DQN** code on a **GPU** node reveal interesting patterns across training and testing phases. During training, **cudaLaunchKernel** (launching CUDA kernels) dominated CPU time at 18%, while **Optimizer.step#Adam.step** took 11%. On the GPU, **aten::addmm** was most time-consuming at 22% [43]. Memory-wise, **aten::to** and **aten::_to_copy** (tensor conversion and copying operations) each used about 50 MB on CPU, while **aten::linear** and **aten::addmm** each consumed around 263 MB on GPU [42].

In the testing phase, the resource utilisation shifted notably. **CudaStreamIsCapturing** (checking CUDA stream state) became the most CPU-intensive operation at 36%, followed by **cudaFree** (freeing CUDA memory) at 16% [43]. On the GPU, **aten::addmm** remained dominant, using 71% of CUDA time. Memory usage during testing was significantly lower, with **aten::to** and **aten::_to_copy** each using about 787 KB on CPU, and **aten::linear** and **aten::addmm** each consuming around 10.93 MB on GPU. The significant difference between CPU Time Total (12.228 seconds) and CUDA Time Total (0.699 seconds) suggests that while the GPU is highly efficient at processing the neural network operations, there is significant CPU overhead in managing the GPU and coordinating data transfers, indicating potential room for optimisation in CPU-GPU interaction. At the baseline problem size (3 equities and 6 years of time series data), the dataset is too small for both training and testing phases to fully leverage GPU capabilities. Consequently, the overhead from launching GPU operations and copying data comprises significant proportions of computation time.

The profiling logs for the **PPO** code running on a **CPU** node reveal interesting insights into resource utilisation during both training and testing phases. In the training phase, **aten::linear** and **aten::addmm** were the most CPU-intensive operations, consuming about 25% and 18% of total CPU time respectively. **Optimizer.step#Adam.step** was also significant, using 13% of total CPU time. Memory-wise, **GatherBackward0** (gradient computation for gather operation) was the top consumer at 96.84 MB, followed closely by **aten::gather_backward** (calculates gradients for element selection) and **aten::new_zeros** (creates zero-filled tensor), both using similar amounts of memory [42].

The testing phase showed a shift in resource allocation. **Aten::linear** and **aten::addmm** remained the most CPU-intensive operations, but with increased proportions of 34% and 21% of total CPU time respectively. Memory usage during testing was significantly lower, with **aten::linear** and **aten::addmm** each consuming about 4.49 MB, followed by **aten::relu** (activation function) at 3.68 MB [42]. This reduction in memory usage during testing is expected, as the model is not storing gradients or updating parameters. These profiles highlight the computational intensity of linear operations in neural networks and the memory demands of backpropagation during training.

The profiling logs for the **PPO** code on a **GPU** node reveal interesting patterns across training and testing phases. During training, **aten::linear** dominated CPU time at 22%, while **aten::linear** and **aten::addmm** were most time-consuming on the GPU, using about 22% and 19% of CUDA time respectively. **Optimizer.step#Adam.step** used 12% of CPU time and 16% of CUDA time. Memory-wise, **aten::to** and **aten::_to_copy** each used about 1.83 MB on CPU, while **GatherBackward0**, **aten::gather_backward**, and **aten::new_zeros** each consumed around 96.84 MB on GPU. These operations highlight the resource-intensive nature of training, particularly in gradient computations and memory usage. **Figure 5.3** shows the timeline view of a section of the execution of a step, highlighting **aten::linear** as the dominant operation.

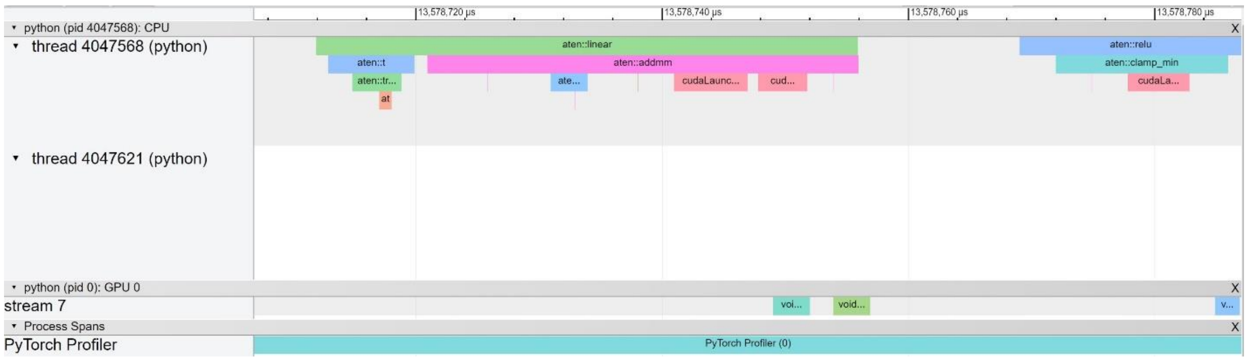

Figure 5.3: Timeline view of a single step during the PPO training phase on a GPU node.

In the testing phase, the resource utilisation shifted notably. **Aten::linear** and **aten::addmm** remained dominant, with **aten::addmm** using 30% of CUDA time. **CudaStreamIsCapturing** and **cudaLaunchKernel** were significant CPU operations, each using about 15% of CPU time [43]. Memory usage during testing was lower, with **aten::to** and **aten::_to_copy** each using about 206 KB on CPU, and **aten::linear** and **aten::addmm** each consuming around 22.09 MB on GPU. Similar to DQN, launching the GPU and copying data between the CPU and GPU consume a significant proportion of execution time due to the small problem size. **Figure 5.4** shows the timeline view of a section of the execution of a step, highlighting **aten::linear** and **aten::addmm** as the dominant operations.

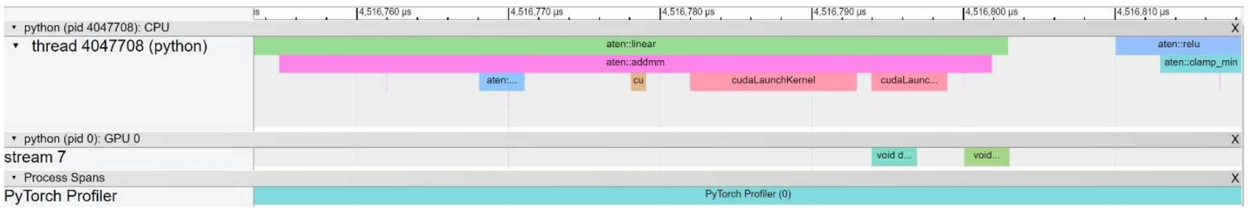

Figure 5.4: Timeline view of a single step during the PPO testing phase on a GPU node.

The profiling logs for DQN and PPO algorithms on both CPU and GPU nodes reveals several key insights. Across all configurations, linear operations (**aten::linear** and **aten::addmm**) consistently dominate computational time, highlighting the importance of optimising these core neural network operations. GPU implementations show significant speedups, particularly for these linear operations. However, they also introduce additional overhead in memory management and data transfer (e.g., **cudaLaunchKernel**, **cudaMemcpyAsync**) [43]. Memory usage patterns differ notably between training and testing phases, with training requiring substantially more memory due to gradient storage and parameter updates. The **Optimizer.step#Adam.step** operation is a significant time consumer during training, emphasising the cost of parameter updates. Tensor conversion operations (**aten::to**, **aten::_to_copy**) also appear as consistent consumers of both time and memory, indicating an area for potential optimisation in data preprocessing or model architecture [42]. All profiling logs are available in our associated code repository [30].

5.1.2. Baseline Results

In this subsection, we analyse the baseline implementations of DQN and PPO across a range of financial and computational performance metrics. For portfolio values, we report the median across multiple runs. This approach is chosen because financial data often exhibits a fat-tailed distribution, where outliers can significantly skew the average [44]. A fat-tailed distribution is a probability distribution where extreme events (outliers) have a disproportionately large impact on the overall properties of the distribution [44]. By focusing on the median, we obtain a more robust representation of the central tendency of the portfolio values [45]. For execution time and other resource utilisation metrics, we assume thin-tailed distributions (where extreme values do not dominate the distribution's properties) [44]. In these cases, we report the mean values across runs to provide reliable estimates.

Table 5.1 reveals intriguing performance differences between DQN and PPO models on CPU nodes. DQN achieves a higher median portfolio value during training (USD 135,324 vs PPO's USD 113,413), but PPO takes the lead in testing (USD 131,217 vs DQN's USD 111,366). This flip in performance might be due to DQN's experience replay mechanism allowing quick learning in stable environments, while PPO's on-policy approach potentially offers better adaptability to new scenarios. PPO demonstrates faster training times, but slower testing times compared to DQN. The faster training of PPO could be attributed to its on-policy learning approach, which, combined with an update frequency of 128 steps and 4 epochs per update, enhances sample efficiency.

The slower testing time for PPO can be explained by its use of two networks (actor and critic) in action selection, compared to DQN's single network. This means PPO performs two forward passes per action during testing, while DQN needs only one. Memory usage patterns align with the algorithms' structures, with DQN consuming more CPU memory during training, likely due to its replay buffer. Energy consumption follows a similar pattern, with DQN using more energy in training and PPO in testing.

CPU Node	**DQN**		**PPO**	
	Train	**Test**	**Train**	**Test**
Median Portfolio Value (USD)	135,324	111,366	113,413	131,217
Mean Execution Time (seconds)	138.64	28.33	113.62	58.28
Mean CPU Memory Usage (total) (MB)	1,014.72	4.43	851.42	10.20
Mean Consumed Energy (joules)	29,810	13,310	27,940	14,460

Table 5.1: Comparative performance metrics of DQN and PPO models during training and testing on CPU node.

Table 5.2 presents a comparison of DQN and PPO models running on GPU nodes, offering insights into their performance and resource utilisation. In terms of portfolio value, both DQN and PPO achieve similar performance (in testing). The difference between these results and their performance seen on CPU nodes is attributable to different seed values. However, PPO maintains faster training times, though it is slower in testing. This aligns with our earlier observation about PPO's dual-network architecture affecting testing speed.

The resource utilisation metrics reveal interesting patterns. DQN uses more CPU memory in both training and testing phases compared to those of PPO, suggesting that DQN's memory-intensive operations are not fully offloaded to the GPU. GPU memory usage appears very high due to **aten::empty**, which allocates large, uninitialized tensors on the GPU [42]. This optimises performance by avoiding unnecessary initialisation overhead, as these tensors are typically filled immediately by subsequent operations. However, these uninitialized tensors still occupy GPU memory.

Energy consumption is higher for DQN in training but lower in testing, aligning with the execution time differences between the algorithms. The mean power consumption remains consistent across both algorithms and phases, suggesting that the GPU maintains a relatively constant power draw regardless of the specific computations being performed. The low GPU utilisation for both algorithms suggest inefficient GPU resource use, likely due to frequent CPU-GPU data transfers or un-optimised GPU operations. Additionally, the problem size (3 equities and 6 years of daily time series data) may be too small to fully leverage GPU's capabilities. This indicates significant room for optimisation to better utilise GPU resources and potentially improve performance for both algorithms in automated trading.

GPU Node	**DQN**		**PPO**	
	Train	Test	Train	Test
Median Portfolio Value (USD)	161,144	126,406	120,193	124,652
Mean Execution Time (seconds)	170.61	31.28	136.22	77.61
Mean CPU Memory Usage (total) (MB)	94.22	1.34	3.94	0.31
Mean GPU Memory Usage (total) (MB)	89,678.63	22,198.89	43,165.49	41,484.13
Mean Consumed Energy (joules)	55,630	12,990	43,070	26,380
Mean Power (watts)	62.37	62.00	61.96	61.93
Mean GPU Utilisation (%)	11.40	6.14	11.48	11.47

Table 5.2: Comparative performance metrics of DQN and PPO models during training and testing on GPU node.

In **Figure 5.5**, we plot the median portfolio value over 100 episodes for DQN on a CPU Node. The training line shows a steady increase in portfolio value over 100 episodes, indicating that the DQN model is learning and improving its strategy as it interacts with the environment. The observed volatility is attributable to the exploration-exploitation trade-off inherent in DQN.

The testing line, however, remains lower than the training line, suggesting that the model, despite performing well during training, struggles to generalise to unseen data in the testing phase. Although lower than in training, the volatility in the testing line can also be attributed to the exploration-exploitation trade-off, with epsilon set at a constant 0.01. The gap between training and testing performance could indicate overfitting or difficulty adapting to new market conditions and volatility during testing. This highlights the potential need for hyperparameter tuning to enhance the model's generalisation capabilities.

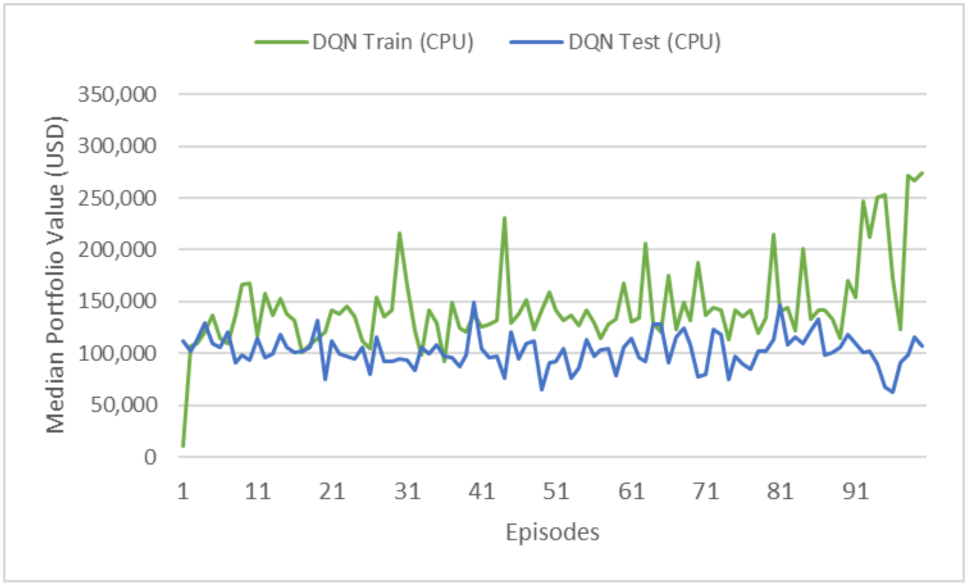

Figure 5.5: Median portfolio value over 100 episodes for **DQN** on **CPU** node.

In **Figure 5.6**, we observe similar behaviour of the DQN model on the GPU node compared to the CPU node. The performance difference can be attributed to the variation in seed values.

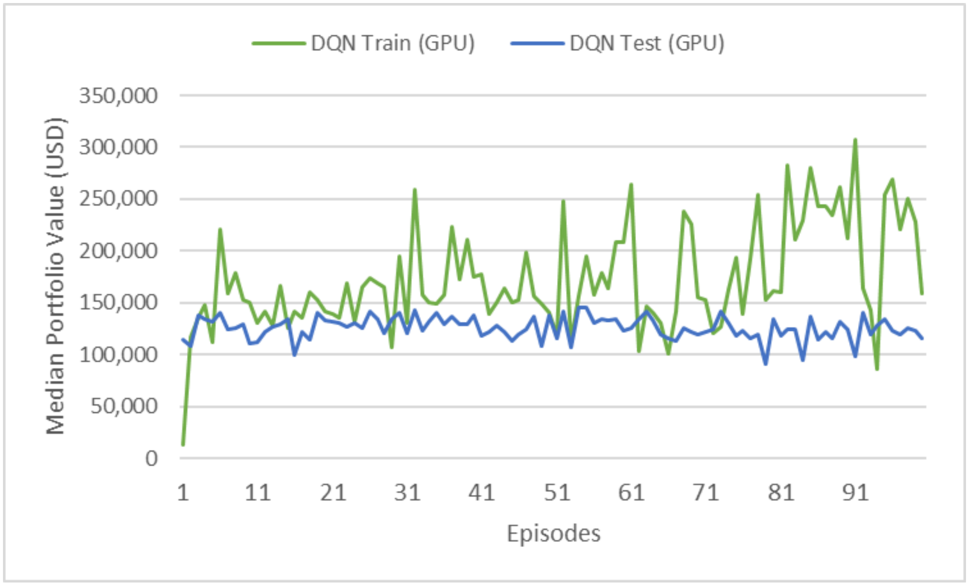

Figure 5.6: Median portfolio value over 100 episodes for **DQN** on **GPU** node.

Figure 5.7 illustrates the median portfolio value of the PPO model over 100 episodes on a CPU node. The training line shows a gradual upward trend, indicating that the PPO model is learning and improving its trading strategy, but at a slower rate compared to DQN. The testing line shows less volatility compared to the training line. During testing, the model applies its learned policy without further updates, leading to more consistent and stable behaviour compared to training, where exploration introduces variability. Interestingly, the testing performance often exceeds the training performance, particularly in later episodes. This could indicate that PPO is generalising well to unseen data, possibly due to its on-policy learning approach and ability to adapt to new market conditions.

Both the training and testing lines exhibit less volatility compared to DQN, reflecting PPO's more stable policy updates (due to the clipping mechanism) and gradual learning process. The downside of PPO is that it can get stuck in local optima [46]. Therefore, PPO can also benefit from hyperparameter tuning to improve performance.

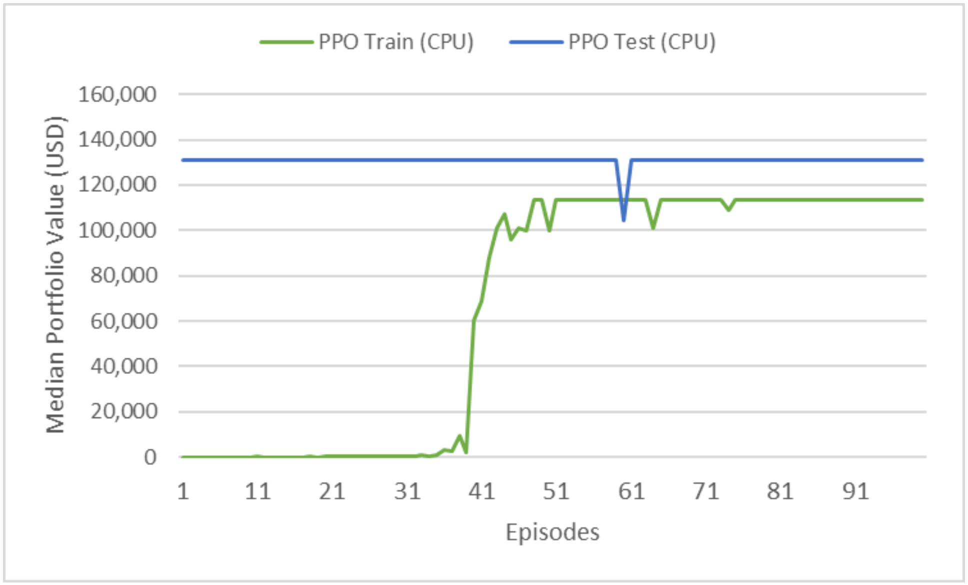

Figure 5.7: Median portfolio value over 100 episodes for **PPO** on **CPU** Node.

In **Figure 5.8**, we observe similar behaviour of PPO on a GPU node compared to that on a CPU node. The difference in values between the CPU and GPU nodes is attributed to the difference in seed values.

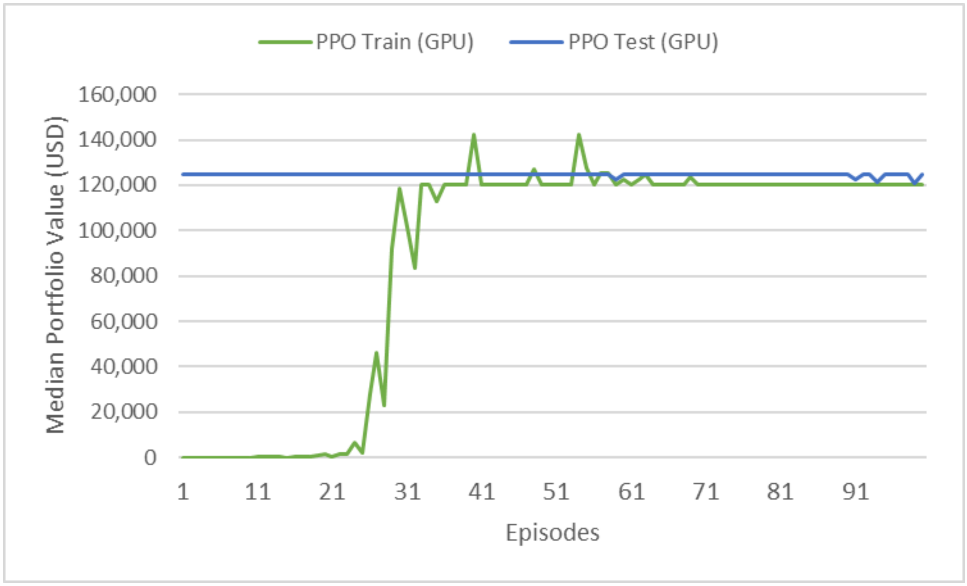

Figure 5.8: Median portfolio value over 100 episodes for **PPO** on **GPU** Node.

5.2 Experiment 2: Hyperparameter Optimisation Results

In this section, we present the results of our hyperparameter tuning experiment, highlighting the optimal parameters identified and analysing the impact of different hyperparameters on the models' performance. We explore how tuning key parameters improved both trading outcomes and computational efficiency, offering insights into their influence on overall model behaviour.

Our hyperparameter set selection focused on optimising both the median testing portfolio value and the mean total training execution time. We sorted the configurations by median portfolio value and then selected the configuration that offered the best balance of financial and computational performance. This approach prevented us from over-optimising for portfolio value at the expense of significantly longer training times.

We specifically chose a configuration that demonstrated strong results across both CPU and GPU nodes. This enables users to run the code with an already optimised set of parameters, eliminating the need to adjust them for specific hardware. In general, we observed that configurations performed similarly on both types of nodes, with any variations in ranking being mostly attributable to differences in seed values. The full dataset from our hyperparameter tuning process is available in our code repository [30].

Table 5.3 presents the optimal hyperparameter values for DQN and PPO, derived from the hyperparameter tuning experiment. For DQN, a learning rate of 0.001, gamma of 0.95, and epsilon decay of 0.99 were found to provide the best balance between exploration and exploitation, while promoting stable learning. These values likely helped DQN make gradual yet effective updates, ensuring that future rewards were well-considered without making the learning process too conservative.

For PPO, the optimal learning rate was 0.0003, with 4 epochs per update and a policy clip value of 0.2. The lower learning rate allows PPO to make more controlled and precise policy updates, while the policy clip parameter ensures that changes in the policy do not deviate too much, maintaining stability. The 4 epochs strike a balance between improving the policy and preventing overfitting during each update cycle. These settings collectively helped optimise PPO's performance in the trading environment.

Algorithm	Hyperparameter	Value
DQN	Learning Rate	0.001
	Gamma	0.95
	Epsilon Decay	0.99
PPO	Learning Rate	0.0003
	Epochs	4
	Policy Clip	0.2

Table 5.3: Best hyperparameters found from tuning DQN and PPO in Experiment 2.

The following charts in this section illustrate the general trends observed when varying each hyperparameter. To avoid overcrowding the charts with too many lines, we focus on a specific portion of the hyperparameter grid to highlight the effects. As similar patterns were observed across both CPU and GPU nodes, we present the results from one node per parameter to prevent cluttering the section with excessive charts.

Figure 5.9 compares the median portfolio values of DQN and PPO during testing across a range of learning rates. We see a performance improvement for DQN as the learning rate increases, reaching its peak median portfolio value at 0.001. This suggests that a moderate learning rate enables DQN to update its Q-values effectively, striking a balance between overfitting and underfitting. Conversely, PPO's performance is optimal at a lower learning rate of 0.0003, with a decline observed at higher rates. This indicates that PPO, due to its clipped policy updates, is less sensitive to changes in the learning rate beyond a certain point. The clipping mechanism prevents abrupt shifts in the decision-making process even with larger learning rates.

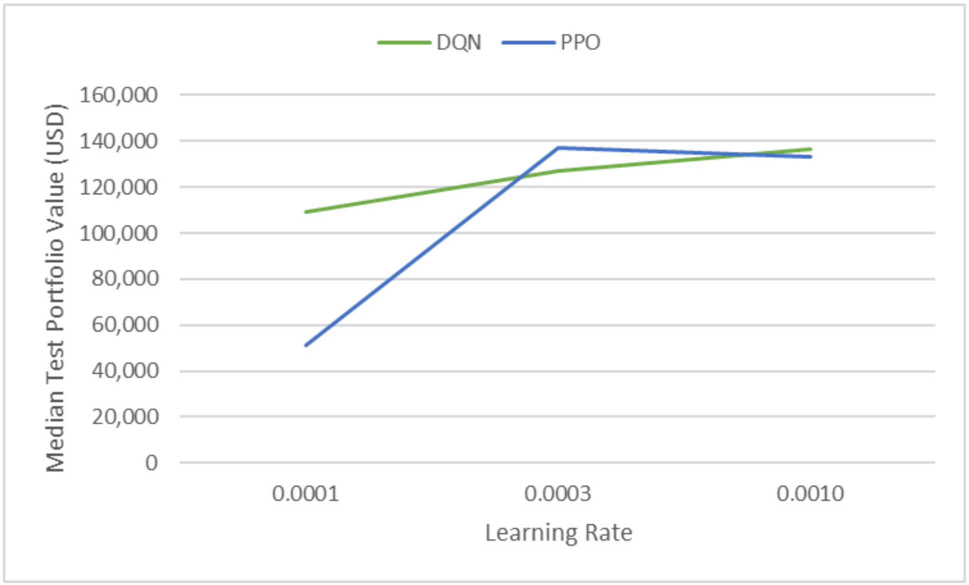

Figure 5.9: Median portfolio value for DQN and PPO during testing (on CPU Node) across different learning rates. DQN uses fixed values of gamma (0.95) and epsilon decay (0.990). PPO uses fixed values for epochs (8) and policy clip (0.1).

Figure 5.10 illustrates the mean execution time for DQN and PPO during training across various learning rates. DQN's execution time remains relatively stable across all learning rates, suggesting that changes in the learning rate do not significantly impact its time efficiency. This stability is because DQN's off-policy nature relies on experience replay, which decouples learning updates from real-time interactions, resulting in consistent computation across learning rates [1].

Conversely, PPO demonstrates a notable decrease in execution time when transitioning from a learning rate of 0.0001 to 0.0003, after which it plateaus. This suggests that PPO's time efficiency is more sensitive to changes in the learning rate, but only up to a certain point. The reduction in execution time is likely attributed to the larger updates per gradient step facilitated by the increased learning rate, potentially leading to faster convergence. However, as the portfolio value results in Figure 5.9 illustrate, faster execution does not always translate to improved trading outcomes.

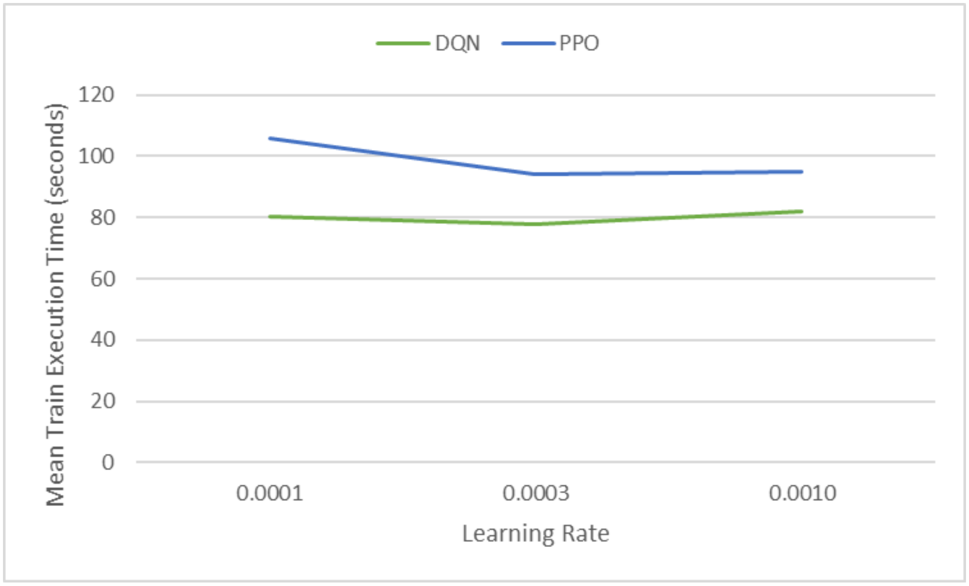

Figure 5.10: Mean execution time for DQN and PPO during training (on CPU Node) across different learning rates. DQN uses fixed values of gamma (0.95) and epsilon decay (0.990). PPO uses fixed values of epochs (8) and policy clip (0.1).

Figure 5.11 illustrates the median portfolio value of DQN across different gamma values. The graph shows that as gamma increases from 0.90 to 0.95, the portfolio value improves, reflecting DQN's ability to better optimise future rewards. However, the performance slightly declines when gamma reaches 0.99, suggesting that while considering future rewards is important, placing too much emphasis on long-term rewards can lead to instability or overly cautious behaviour. This indicates that balancing short-term and long-term rewards is vital for optimal trading performance. It is worth noting that while 0.95 performs best, the differences are not dramatic in the testing phase, suggesting DQN is relatively robust to gamma variations within this range.

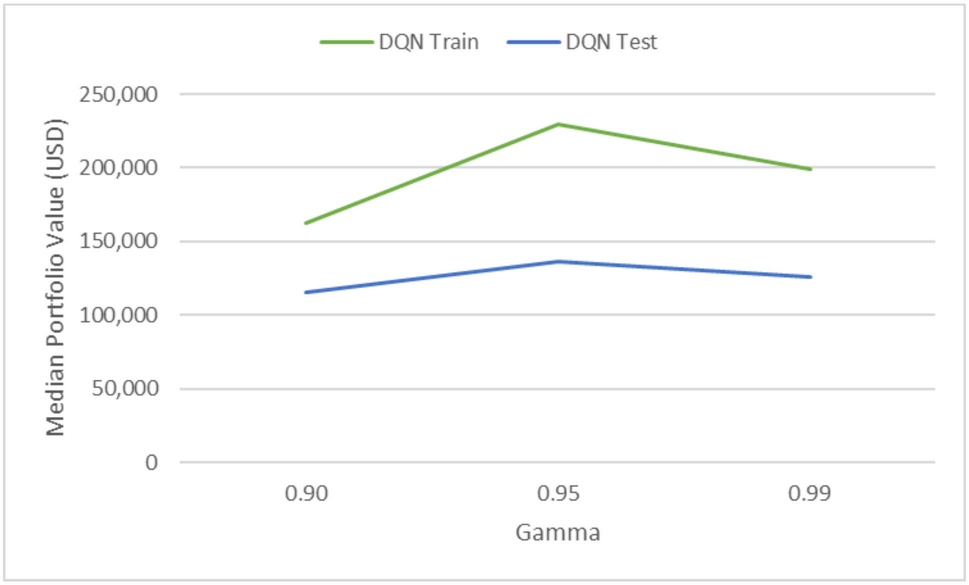

Figure 5.11: Median portfolio value of DQN across different gamma values (on CPU Node). Learning rate (0.0010) and epsilon decay (0.990) are fixed.

Figure 5.12 shows the median portfolio value for DQN during training and testing across different epsilon decay values. The results indicate that as the epsilon decay rate increases from 0.990 to 0.999, both the training and testing performance decline significantly. At an epsilon decay rate of 0.990, the model achieves the highest portfolio values in both training and testing, suggesting that a more rapid transition from exploration to exploitation leads to better overall performance. As the decay rate increases, allowing the model to explore for longer, the portfolio values drop, with the lowest performance observed at 0.999, both in training and testing. This indicates that too much exploration may prevent the model from effectively capitalising on learned strategies, demonstrating the need for a well-balanced exploration-exploitation trade-off.

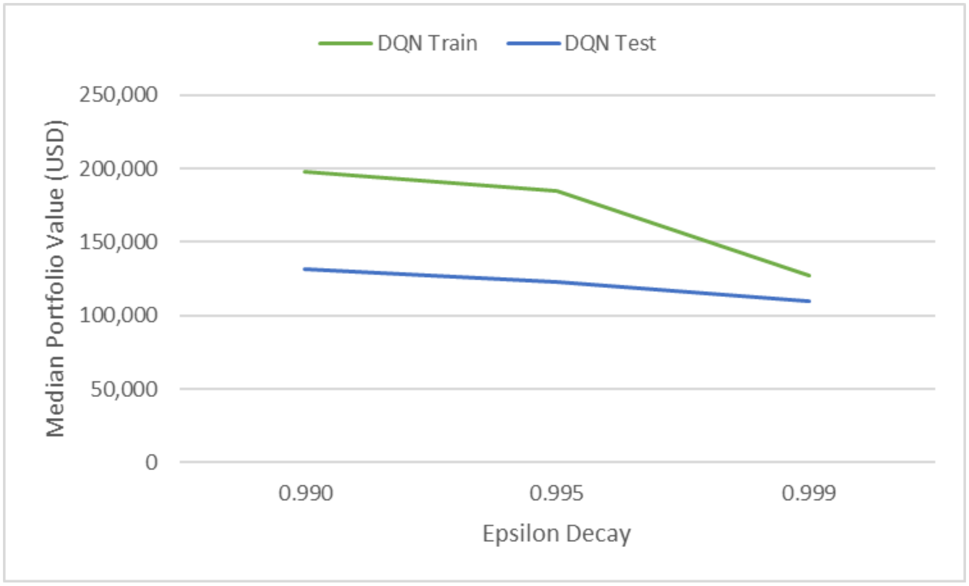

Figure 5.12: Median portfolio value of DQN across different epsilon decay values (on GPU Node). Learning rate (0.0010) and gamma (0.90) are fixed.

Figure 5.13 illustrates the effect of varying the number of epochs on PPO's median portfolio value during training and testing. The data reveals an intriguing pattern across different epoch settings. For 4 and 8 epochs, the performance remains identical in both training and testing, suggesting that doubling the epochs from 4 to 8 does not yield any improvement in this scenario. However, when the epochs are increased to 16, there is a substantial rise in training performance, but a slight decrease in testing performance. This divergence between training and testing results at 16 epochs indicates potential overfitting, where the model performs exceptionally well on the training data but fails to generalise as effectively to unseen data. The results suggest that for this trading environment, 4 epochs might be the most efficient choice, offering a good balance between computational cost and performance without the risk of overfitting.

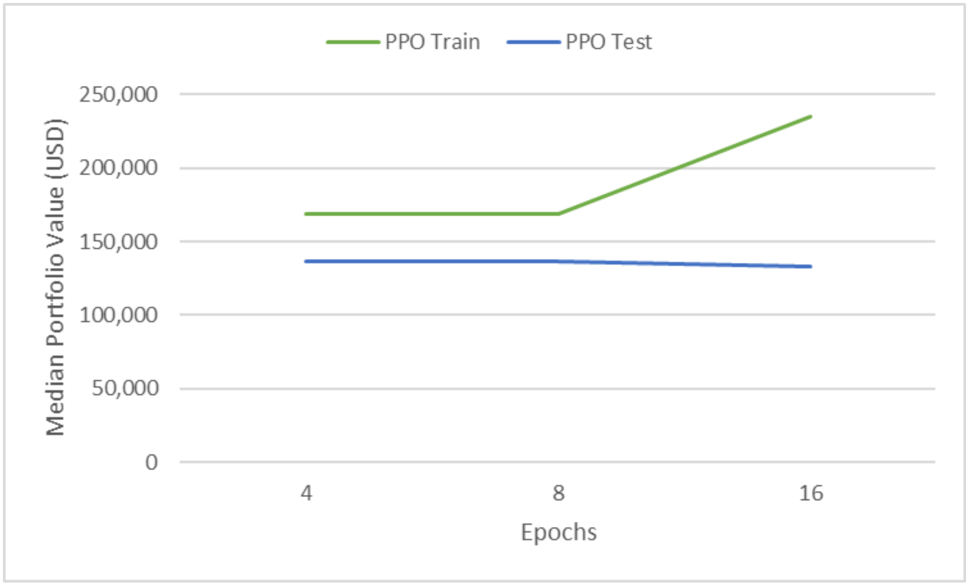

Figure 5.13: Median portfolio value of PPO across different epochs (on CPU Node). Learning rate (0.0003) and policy clip (0.2) are fixed.

Figure 5.14 illustrates the effect of different policy clip values on the PPO model's performance during training and testing. The results show that a moderate clip value of 0.2 leads to the best overall performance, while both lower (0.1) and higher (0.3) clip values result in significantly reduced performance. A smaller clip value of 0.1 overly restricts the policy updates, limiting the model's ability to adapt, which results in poor performance. Conversely, a larger clip value of 0.3 allows for too much variation in the policy updates, leading to instability and suboptimal results. The optimal performance at 0.2 suggests that a balanced policy clip value enables sufficient learning while maintaining stability, allowing the model to effectively generalise during testing. The substantial difference in performance across these clip values underscores the critical role of this hyperparameter in PPO's learning process.

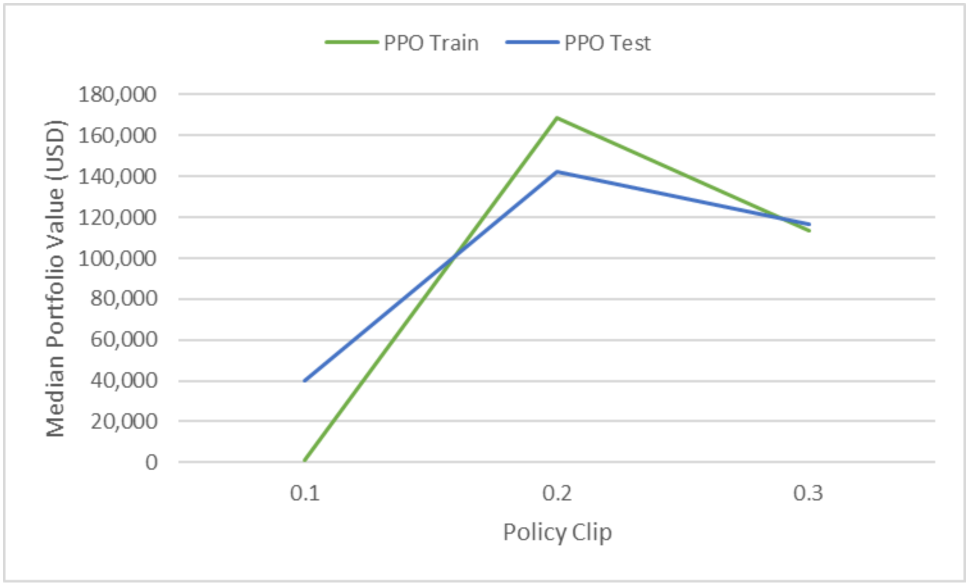

Figure 5.14: Median portfolio value of PPO across different policy clips (on GPU Node). Learning rate (0.0003) and epochs (4) are fixed.

5.3 Experiment 3: Scalability Analysis Results

In section 5.3, we present the results of our scalability analysis for the DQN and PPO algorithms in automated trading. This experiment evaluates how the performance and computational requirements of these algorithms change as we increase the number of assets in the trading portfolio, providing insights into their scalability on both CPU and GPU nodes.

Figure 5.15 shows the mean execution time for DQN and PPO models across different numbers of equities in the portfolio on a CPU node. For DQN, the training time remains relatively stable when moving from 1 to 3 equities, but then increases sharply for 5 and 10 equities. This indicates that DQN's computational complexity grows considerably with larger portfolio sizes, leading to higher processing times. However, in testing, the execution time increases only slightly, suggesting that DQN is more efficient in handling the testing phase despite the larger portfolio size.

PPO demonstrates a different scaling pattern. Its training time increases more gradually than that of DQN across the range of portfolio sizes. PPO's testing time also shows a more notable increase compared to DQN as the number of equities grow from 5 to 10. These results suggest that while both algorithms face increased computational demands with larger portfolios, DQN's training process is more sensitive to the number of equities, particularly beyond 5 assets. PPO, while generally faster for smaller portfolios, sees a more consistent increase in both training and testing times as the portfolio size grows.

The different scaling behaviours are attributed to the algorithmic differences between DQN and PPO. The action space is 3^N for both models, where N is the number of equities. For both algorithms, there is no significant difference between the execution times of one equity and 3 equities. This is attributed to the vectorised operations in the neural network computations. Vectorised operations on CPU allow simultaneous processing of multiple data elements, leveraging CPU's SIMD (Single Instruction, Multiple Data) capabilities to efficiently handle small increases in input size without significantly impacting execution time [12]. At 10 equities, both DQN and PPO's training times increase significantly compared to 5 equities, reflecting the larger neural network computations required for the expanded action space.

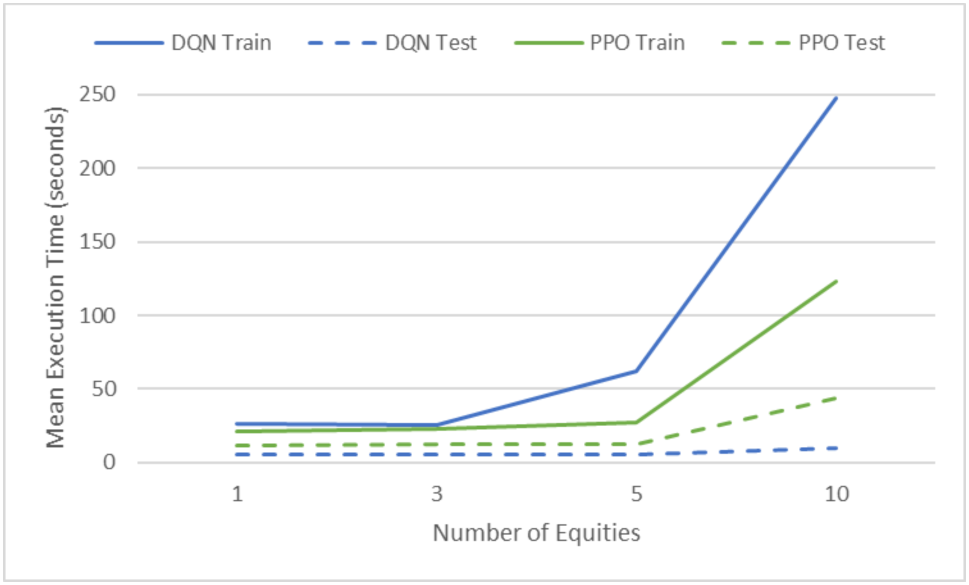

Figure 5.15: Mean execution time (of 20 episodes) versus number of equities in portfolio (on CPU node).

DQN takes longer in training compared to PPO due to its experience replay mechanism with per-step updates. PPO, on the other hand, has an update frequency set at 128, with multiple training epochs (4) at each update. As a result, PPO requires significantly fewer training iterations, benefiting from the sample efficiency inherent in the model. However, during testing, DQN shows an advantage due to its single network architecture. PPO, with its dual network architecture (actor and critic), performs two forward passes per action during testing, whereas DQN requires only one. This explains why DQN scales better in testing than PPO, as it requires significantly fewer neural network computations.

Figure 5.16 displays the mean execution time for DQN and PPO models across different numbers of equities on a GPU node. On the GPU node, DQN's training time follows a similar trend to that observed on the CPU node, with relatively stable times up to 5 equities and a sharp increase at 10 equities. However, the increase is less dramatic on the GPU node compared to that of the CPU node, indicating that DQN benefits more from GPU acceleration as the portfolio size grows. In testing, DQN's execution times remain stable across different equity counts, similar to its behaviour on the CPU node, but with slightly slower processing than that of the CPU node up to 5 equities. At 10 equities, DQN’s testing performance is very close to its CPU performance, suggesting it gains from large portfolio sizes in GPU parallelisation.

PPO's training time on the GPU, unlike on the CPU, remains almost constant across all equity levels, demonstrating its strong scalability on GPU hardware. This differs from the more gradual increase seen on the CPU. In testing, PPO’s GPU performance is slightly slower than that of the CPU up to 5 equities. At 10 equities, PPO’s GPU performance is significantly faster than that of the CPU. Similar to DQN, PPO also benefits from GPU parallelisation with a larger portfolio size during testing.

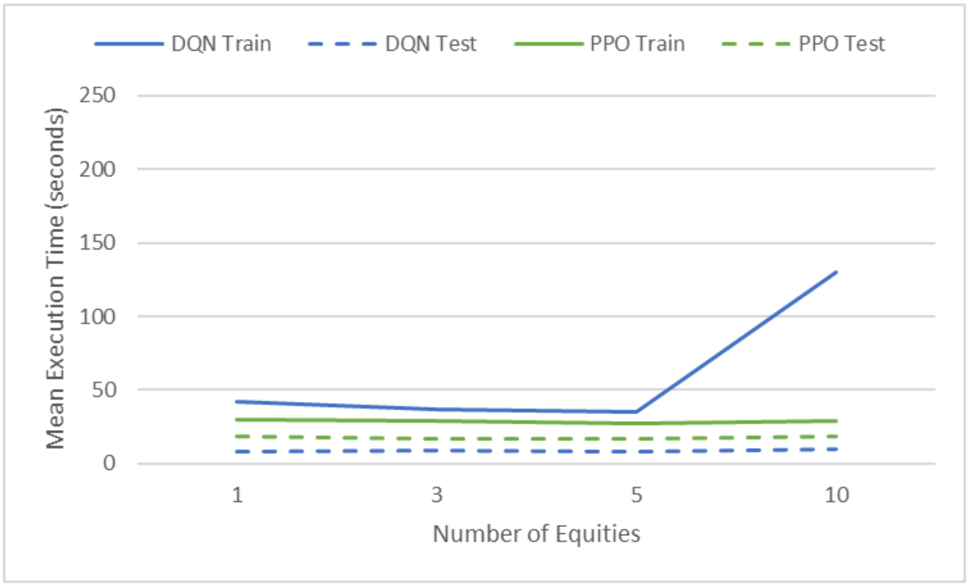

Figure 5.16: Mean execution time (of 20 episodes) versus number of equities in portfolio (on GPU node). The vertical axis scale is consistent with the CPU node’s figure for direct comparison.

Comparing both figures, the GPU accelerates both models, especially PPO, to handle larger portfolios more efficiently. In training, both models begin to benefit from GPU parallelisation at 5 equities, while in testing, they start gaining from it at 10 equities. This is because, with a smaller number of equities, the GPU overheads are significant, and there is insufficient parallelism to offset these costs. The key GPU-related overheads include **cudaLaunchKernel** (which launches GPU kernels), **cudaMemcpyAsync** (which transfers data between the CPU and GPU), **aten::to** and **aten::_to_copy** (which move tensors between devices), and **cudaStreamIsCapturing** (which checks a stream's capture status) [42][43].

As the number of equities increases, the neural network computations (such as **Optimizer.step#Adam.step**, **aten::addmm**, and **aten::linear**) grow in complexity, which requires more computational power. This increased complexity makes the models benefit more from GPU parallelisation, as GPUs are particularly efficient at handling large-scale matrix operations and parallel computations [12]. The larger the input size (number of equities), the more significant the benefit from distributing these operations across multiple GPU cores.

Although DQN gains from GPU parallelisation at 10 equities, the gap between DQN and PPO execution times remains significantly wide. This suggests that DQN could benefit from optimising CPU-GPU communication overheads. The performance difference may be attributed to DQN's more frequent memory transfers between CPU and GPU, particularly due to its experience replay mechanism. Reducing these transfers or implementing more efficient batching strategies could potentially improve DQN's GPU utilisation and overall execution time, bringing it closer to PPO's performance on the GPU.

It is worth noting that our models do not scale on a single GPU when doubling the number of equities from 10 to 20, as we run out of single GPU memory. This is because the action space, which is 3^N, demands significant memory at 20 equities. This suggests that for very large portfolios, our implementations would necessitate GPUs with larger memory capacities than 16 GB (available on the Tesla V100 GPUs on Cirrus) [2]. Alternatively, we could explore distributed training across multiple GPUs to utilise additional GPU memory [23].

From an algorithmic perspective, we could explore techniques to reduce the scaling of the action space from exponential ($O(3^N)$) to linear ($O(N)$). One potential approach is to employ neural networks that output an action per asset (e.g., equity) given a state, rather than selecting a single action from a pool of 3^N possible actions. Additionally, to accelerate our algorithms on CPU nodes with larger portfolio sizes, we could consider the use of multiprocessing or multithreading techniques [12]. However, these are complex propositions and fall beyond the scope of this project.

5.4 Experiment 4: Robustness and Generalisation Results

In section 5.4, we focus on evaluating the robustness and generalisation capabilities of the DQN and PPO models. We examine how well the models perform when exposed to new, unseen datasets, providing insights into their ability to maintain performance outside of the training environment.

Figure 5.17 compares the performance of DQN and PPO models in various scenarios: training, testing, similar dataset, and commodities ETFs dataset. In training, DQN outperforms PPO significantly, showing its stronger ability to learn from the training data. However, in the testing phase, the gap between DQN and PPO narrows, with PPO slightly outperforming DQN, suggesting that PPO may generalise better to unseen data.

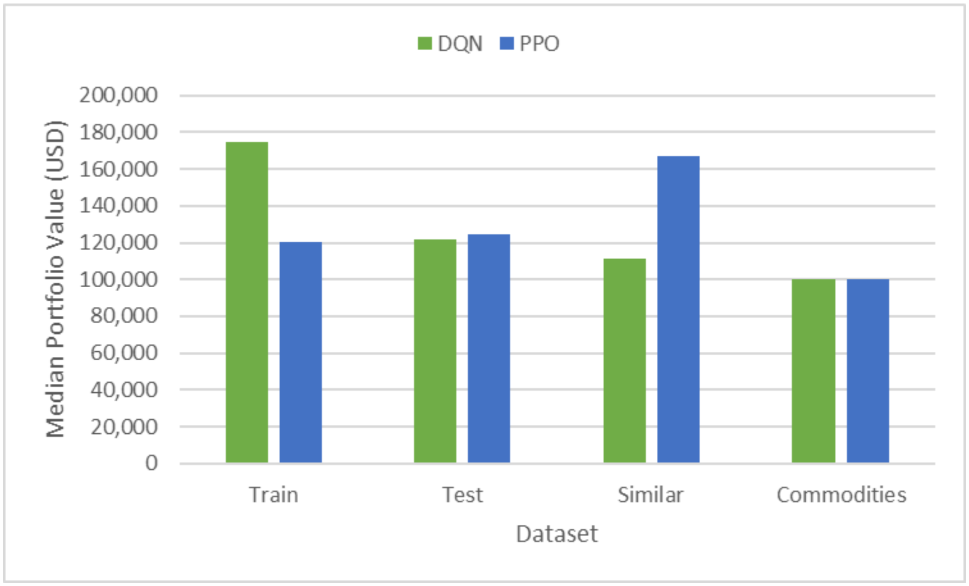

Figure 5.17: Median portfolio value of DQN and PPO across different datasets.

When tested on a similar dataset, PPO significantly outperforms DQN, indicating that PPO is better at adapting its learned policies to new but related environments. This demonstrates PPO's stronger robustness in generalising to similar scenarios. Finally, on the commodities

ETFs dataset, both models struggle to maintain or grow the initial investment. DQN marginally outperforms PPO, but both essentially break even. This suggests neither model has a significant advantage in this domain, likely because commodities ETFs trading differs significantly from equities trading.

Table 5.4 compares the performance of DQN and PPO against the buy-and-hold strategy (our benchmark) across four datasets: baseline equities in training, baseline equities in testing, similar equities, and commodities ETFs, measured by Compound Annual Growth Rate (CAGR). CAGR is a measure of the annual growth rate of an investment over a specified time period, assuming the investment grows at a steady rate. It is calculated using the formula **CAGR = (Ending Value / Beginning Value)$^{(1/n)}$ - 1**, where n is the number of years [45]. CAGR provides a smoothed rate of return that accounts for the compounding effect of growth over time, making it a practical metric for comparing the performance of different investments [45].

Dataset	Buy-and-Hold	DQN	PPO
Train	24.1%	20.5%	6.3%
Test	9.3%	6.7%	7.6%
Similar	30.3%	3.7%	18.6%
Commodities	6.7%	0.1%	0.0%

Table 5.4: Compound Annual Growth Rate (CAGR) Percentage Comparison Across Different Investment Strategies Using Various Datasets.

In the training dataset, buy-and-hold achieves a higher return (24.1%) than both DQN (20.5%) and PPO (6.3%). While DQN performs closer to the benchmark, PPO significantly underperforms, indicating weaker learning during training compared to the benchmark.

In the testing dataset, PPO slightly outperforms DQN, with a 7.6% CAGR compared to DQN’s 6.7%, though both models fall short of the buy-and-hold strategy, which records a 9.3% return. This suggests that neither model surpasses the benchmark in out-of-sample performance, but PPO shows slightly better generalisation than DQN.

In the similar equities’ dataset, buy-and-hold outperforms both models by a wide margin, achieving a 30.3% CAGR. PPO demonstrates a better ability to adapt to similar market conditions, with 18.6%, whereas DQN struggles, achieving only 3.7%. PPO's performance being closer to the benchmark highlights its stronger adaptability to related environments compared to that of DQN.

For commodities ETFs, while the buy-and-hold strategy shows moderate growth (6.7%), both DQN (0.1%) and PPO (0.0%) struggle to generate any meaningful returns. This highlights the challenge both models face in transferring strategies learned on one asset class to a fundamentally different one.

Overall, this comparison indicates that while the DRL models show varying degrees of effectiveness, they consistently struggle to outperform the buy-and-hold benchmark across different datasets. This suggests that the DRL models could benefit from advanced ML techniques, such as improved feature engineering, optimising neural network architectures, ensemble methods (combining different agents), or more thorough hyperparameter tuning [1][27]. Also, training the DRL models on more extensive historical data, spanning several decades, will enable them to learn from a wider range of market regimes and economic cycles, potentially enhancing their adaptability to diverse scenarios. The application of these techniques could enhance the performance of the DRL models, potentially enabling them to outperform the buy-and-hold strategy.

5.5 Comparative Analysis and Discussion

In section 5.5, we present a comprehensive comparative analysis of the DQN and PPO algorithms' performance in automated trading scenarios. This discussion synthesises the findings from our experiments, examining the strengths and weaknesses of each approach across various metrics including portfolio value, execution time, and scalability. We also consider the implications of these results for real-world trading applications and potential areas for future improvement.

5.5.1 Cross-Experiment Comparison

Section 5.5.1 compares the performance of DQN and PPO models, highlighting their distinct strengths and weaknesses. DQN performs well in trading during the training phase but faces challenges with generalisation, while PPO demonstrates more consistent performance across both training and testing phases, showcasing its adaptability to new market conditions.

Hyperparameter optimisation emphasises the necessity of careful tuning for both algorithms, as specific parameters significantly impact performance. Scalability analysis reveals PPO's superior scalability, especially with GPU acceleration, making it potentially more suitable for large-scale trading. However, both algorithms struggle to consistently outperform the buy-and-hold strategy in terms of trading performance, underscoring the need for further refinement.

5.5.2 Comparison with Existing Approaches

In section 5.5.2, we compare the performance of our DQN and PPO models to existing approaches. In our findings, PPO generally outperformed DQN in unseen datasets but struggled to consistently exceed the buy-and-hold strategy across all datasets. Théate and Ernst reported that their TDQN model slightly outperformed the buy-and-hold strategy on average [14]. In contrast, Xia et al. demonstrated that their PPO model achieved an annualised return of 40.8%, outperforming the Dow Jones Index benchmark by 13.8% [16]. Similarly, Yang et al. found that PPO delivered the highest cumulative return among tested algorithms, outperforming both A2C and DDPG [18]. While these studies show PPO’s capacity to exceed traditional benchmarks, our results indicate the need for further optimisation, as neither DQN nor PPO could consistently outperform buy-and-hold strategy in diverse market conditions. It is important to note that the models in the related studies were tested on different assets and over varying time frames, making a direct one-to-one comparison infeasible.

In terms of computational performance, both our DQN and PPO models benefited from GPU acceleration, particularly with larger portfolio sizes, though DQN displayed slower training times compared to PPO. This is related to the findings from Kopel and Szczurek, who observed that whilst DQN achieved higher throughput on GPUs, it did not correlate with better convergence times due to inefficiencies in handling parallel environments [19]. PPO, in contrast, scaled more efficiently and exhibited faster convergence times on GPU. Additionally, WarpDrive's fully GPU-based simulations showed the advantage of eliminating CPU-GPU data transfers, which could further improve the efficiency of DRL models like ours [20]. It is worth noting, however, that these papers use different parallelisation techniques, datasets, and environments, so a direct one-to-one comparison is not feasible.

5.5.3 Insights and Implications

Our experiments provide several key insights into the application of DRL for automated trading in an HPC context. Firstly, while both DQN and PPO show profitable trading outcomes, they consistently underperform the buy-and-hold strategy, highlighting the challenges of applying DRL to complex financial markets. PPO demonstrates better generalisation and adaptability, particularly in unseen market conditions, suggesting its potential for real-world trading scenarios.

The scalability analysis reveals that GPU acceleration becomes increasingly beneficial as portfolio size grows, with PPO showing strong scalability. This implies that for large-scale trading operations, PPO on GPU hardware could be the more efficient choice. However, the underutilisation of GPU resources, as evidenced by low GPU utilisation percentages, suggests there's significant room for optimisation in both algorithms. The performance disparity between training and testing phases, especially for DQN, underscores the importance of robust cross-validation techniques in financial applications. These findings indicate that while our DRL implementations hold significant potential for automated trading, further advancements are necessary to fully realise its capabilities. Refinements such as more advanced reward functions, enhanced feature engineering, and improved algorithms are essential to develop models that can consistently outperform traditional trading strategies across various market conditions.

Chapter 6

Conclusion

Chapter 6 presents the conclusions drawn from our analysis and experimentation with the DQN and PPO models. It summarises the key findings, discusses the limitations of the study, and offers recommendations for future work to further improve the performance and scalability of DRL models in automated trading applications.

6.1 Summary of Key Findings

In this study, we conducted a comprehensive evaluation of DQN and PPO algorithms for automated trading on a heterogeneous HPC system. Our findings reveal that both algorithms can achieve profitable trading outcomes, but they often fall short of buy-and-hold strategy, particularly in unseen market conditions. PPO, with its on-policy learning and clipped objective, generally exhibited better generalisation and adaptability compared to DQN, especially when faced with similar datasets.

Our scalability analysis highlighted the advantages of GPU acceleration, particularly for larger portfolios, with PPO demonstrating strong scalability on GPU hardware. However, both algorithms showed room for improvement in GPU utilisation, suggesting potential for further optimisation. Hyperparameter tuning proved vital for both DQN and PPO, with specific parameter choices significantly impacting performance. Overall, our research underscores the potential of DRL for automated trading, but also emphasises the need for continued advancements in algorithm design, feature engineering, and optimisation techniques to fully realise its capabilities in real-world trading scenarios.

6.2 Reflection on Research Objectives

We addressed our core research objectives in this study. We implemented and optimised both DQN and PPO algorithms for automated trading using PyTorch on the Cirrus HPC system. We conducted a thorough comparative analysis of these algorithms, considering both trading performance and computational efficiency across CPU and GPU architectures. Through hyperparameter tuning and scalability analysis, we gained valuable insights into the factors influencing their effectiveness in real-world trading scenarios. Furthermore, our investigation of model transferability across different datasets shed light on their robustness and adaptability to new market conditions. While our DRL implementations did not achieve consistent outperformance of the buy-and-hold strategy, it laid a solid foundation for future research in this domain, highlighting potential areas for improvement and demonstrating the feasibility of applying DRL to automated trading using HPC resources.

6.3 Limitations of the Study

While this study provides valuable insights into the application of DRL for automated trading on HPC systems, it is essential to acknowledge its limitations. Firstly, our focus on daily data, rather than high-frequency data, might not fully capture the complexities of real-time trading environments where split-second decisions are essential [9]. Secondly, our implementation and analysis were limited to the PyTorch framework, potentially excluding insights that could be gained from other deep learning libraries. Thirdly, our experiments were conducted on a specific HPC system (Cirrus), and the findings might not directly translate to other HPC environments with different architectures or resource constraints. Finally, while we explored the scalability of our models with increasing numbers of assets, we did not investigate their performance in scenarios with significantly larger portfolios or more complex trading strategies, which could pose additional challenges.

6.4 Future Research Directions

One key area for further exploration is the integration of TorchRL, which was used in the early stages of our research for prototyping. Our preliminary work with TorchRL highlighted its potential in advancing RL-based automated trading. Future efforts should focus on exploiting its capabilities, including parallel environment execution and optimised GPU utilisation, to enhance the efficiency and scalability of DRL models in trading applications [40].

Exploring DRL models that handle continuous action spaces could be a fruitful area of research [1]. In real-world trading, decisions such as adjusting portfolio weights or determining order sizes often require fine-tuned, continuous actions [15]. This approach allows for more nuanced control over trading decisions, such as specifying the exact quantity of shares to buy or sell. Implementing continuous action spaces could lead to the development of more adaptable trading strategies, potentially improving the performance of DRL models in dynamic financial markets.

The rise of AI-specific hardware, such as Cerebras Systems' wafer-scale engine, offers promising avenues for accelerating DRL model training and inference [47]. Future research could explore the performance gains and efficiency improvements provided by these specialised architectures in the context of automated trading. Leveraging such powerful hardware could enable the development of more advanced and computationally intensive DRL models, potentially enhancing the effectiveness of trading strategies.

While our study focused on maximising portfolio value, future research should consider incorporating risk management techniques, such as stop-loss orders or tail risk hedging, directly into DRL models [9][45]. This integration would enable the development of more robust trading systems capable of withstanding market volatility and protecting against significant losses, ensuring greater resilience in practical applications.

Although our research primarily relied on backtesting, future work could focus on forward testing or simulated live trading environments to evaluate the real-world performance of DRL models [4]. This approach would provide a more accurate assessment of model effectiveness under dynamic market conditions and help identify any limitations in practical trading scenarios, leading to further refinements and improvements.

Appendix A

Glossary of Terms

This appendix provides concise definitions of key concepts related to trading, machine learning, deep reinforcement learning, and high-performance computing [1][6][9][11][12][27][44][45]. It is intended to offer a clear understanding of essential terms for readers who may be new to these fields.

Action Space: The set of all possible actions an agent can take in a reinforcement learning environment.

Agent: An entity that interacts with an environment by taking actions and receiving rewards, with the goal of learning to maximise its cumulative reward.

Asset: A financial instrument or resource, such as stocks, bonds, or commodities, that holds economic value and can be traded or invested in.

Backtesting: The process of evaluating a trading strategy using historical data to simulate its performance.

Buy-and-Hold Strategy: An investment strategy where assets are purchased and held for a long period, regardless of market fluctuations.

Compound Annual Growth Rate (CAGR): A measure of an investment's annual growth rate over time, accounting for compounding effects.

CPU (Central Processing Unit): The primary component of a computer that performs most of the processing inside the system, particularly suited for serial processing tasks.

Equity: A stock or share representing ownership in a company, giving holders a claim on part of the company's assets and earnings.

Exploration-Exploitation Trade-off: In reinforcement learning, the balance between exploring new actions to gather more information and exploiting known actions that yield high rewards.

Exchange-Traded Fund (ETF): A type of investment fund that tracks an index, commodity, or basket of assets and is traded on a stock exchange like a single stock.

Feature engineering: The process of selecting, transforming, and creating relevant features from raw data to improve the performance of machine learning models.

Forward Testing: A method of evaluating a trading strategy's performance by applying it to out-of-sample data in real-time market conditions, to assess its effectiveness and robustness after initial backtesting.

GPU (Graphics Processing Unit): A specialised processor designed to accelerate graphics rendering, also used for parallel computing tasks.

High-Performance Computing (HPC): The use of powerful processors and parallel computing techniques to perform complex computations more efficiently than standard systems.

Hyperparameter: A parameter whose value is set before the learning process begins, influencing the learning process and model performance.

Learning Rate: A hyperparameter in machine learning that controls how much to change the model in response to the error it observes during training.

Model-Based Reinforcement Learning: A type of reinforcement learning where the agent builds and uses a model of the environment to predict future states and rewards.

Model-Free Reinforcement Learning: A type of reinforcement learning where the agent learns directly from experience without building an explicit model of the environment.

Off-Policy Reinforcement Learning: A type of reinforcement learning where the agent can learn from experiences generated by a behaviour policy different from the target policy it is trying to improve.

On-Policy Reinforcement Learning: A type of reinforcement learning where the agent learns from experiences generated by the same policy it is currently evaluating and improving.

Policy: In reinforcement learning, the strategy used by an agent to determine its actions based on the current state of the environment.

Portfolio: A collection of financial investments held by an individual or organisation.

Proximal Policy Optimisation (PPO): A reinforcement learning algorithm that improves the stability of policy updates through clipping, ensuring more reliable learning in complex environments.

Reinforcement Learning: A type of machine learning where an agent learns to make sequential decisions by interacting with an environment to maximise a cumulative reward signal.

Replay Buffer: A memory bank used in reinforcement learning algorithms to store past experiences for later use in training.

Reward Function: A function that defines the reward an agent receives for taking a particular action in a given state, guiding its learning process.

Scalability: The ability of a system or algorithm to handle increasing amounts of data or computations efficiently as more resources are added.

State: A representation of the current situation or observation in an environment, providing information to the agent for decision-making.

Stop-Loss Order: A trading order placed to automatically sell an asset when its price falls to a certain level, limiting potential losses.

Tail Risk Hedging: A strategy to protect against extreme market movements that occur in the "tails" of a probability distribution, typically representing rare but high-impact events.

Tensor: A multi-dimensional array, fundamental to many machine learning algorithms, especially in deep learning.

Transaction Cost: The expense incurred when buying or selling a security, often represented as a percentage of the transaction value.

Value Function: A function that estimates the expected future reward an agent can obtain from a given state or state-action pair, guiding its decision-making process.

Bibliography

[1] M. Lapan, *Deep Reinforcement Learning Hands-On*. Birmingham: Packt Publishing, Limited, 2018. [Online]. Available: ProQuest Ebook Central. Accessed on: Sept. 5, 2024.

[2] EPCC. Cirrus User Documentation. The University of Edinburgh. [Online]. Available: https://docs.cirrus.ac.uk/. Accessed on: Sept 5, 2024.

[3] C. Ma, J. Zhang, J. Liu, L. Ji, and F. Gao, "A Parallel Multi-Module Deep Reinforcement Learning Algorithm for Stock Trading," *Neurocomputing (Amsterdam)*, vol. 449, pp. 290-302, 2021. doi: 10.1016/j.neucom.2021.04.005.

[4] Real Trading, "Forward Testing," *Real Trading*, 2023. [Online]. Available: https://realtrading.com/trading-blog/forward-testing/

[5] S. Russell and P. Norvig, *Artificial Intelligence: a Modern Approach, Global Edition*. Harlow: Pearson Education, Limited, 2021. [Online]. Available: ProQuest Ebook Central. Accessed on: Sept. 5, 2024.

[6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, Massachusetts: The MIT Press, 2018.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," arXiv preprint arXiv:1312.5602, 2013. doi: 10.48550/arxiv.1312.5602.

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," arXiv preprint arXiv:1707.06347, 2017. doi: 10.48550/arxiv.1707.06347.

[9] B. Johnson, *Algorithmic Trading & DMA: An Introduction to Direct Access Trading Strategies*. London: 4Myeloma Press, 2010.

[10] S. Jansen, *Machine Learning for Algorithmic Trading: Predictive Models to Extract Signals from Market and Alternative Data for Systematic Trading Strategies with Python*, 2nd ed. [Place of publication not identified]: Packt Publishing, 2020.

[11] M. M. López de Prado, *Advances in Financial Machine Learning,* Hoboken, New Jersey: Wiley, 2018.

[12] R. Robey and Y. Zamora, *Parallel and High Performance Computing*. New York: Manning Publications Co. LLC, 2021.

[13] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, 1st ed., Boca Raton, FL: CRC Press, 2011.

[14] T. Théate and D. Ernst, "An Application of Deep Reinforcement Learning to Algorithmic Trading," *Expert Systems with Applications*, vol. 173, p. 114632, 2021. https://doi.org/10.1016/j.eswa.2021.114632.

[15] R. AbdelKawy, W. M. Abdelmoez, and A. Shoukry, "A Synchronous Deep Reinforcement Learning Model for Automated Multi-Stock Trading," *Progress in Artificial Intelligence*, vol. 10, no. 1, pp. 83-97, 2021. https://doi.org/10.1007/s13748-020-00225-z.

[16] Z. Xia, M. Shi, and C. Lin, "Stock Trading Strategy Developing Based on Reinforcement Learning," in *Proc. ICBIS 2023*, 2023, pp. 156–164. [Online]. Available: https://doi.org/10.2991/978-94-6463-198-2_18

[17] J.-H. Park, J.-H. Kim, and J.-H. Huh, "Deep Reinforcement Learning Robots for Algorithmic Trading: Considering Stock Market Conditions and U.S. Interest Rates," *IEEE Access*, vol. 12, pp. 20705–20725, 2024. doi: 10.1109/ACCESS.2024.3361035.

[18] H. Yang, X.-Y. Liu, S. Zhong, and A. Walid, "Deep Reinforcement Learning for Automated Stock Trading: An Ensemble Strategy," in *ICAIF 2020 - 1st ACM International Conference on AI in Finance*, New York, NY, USA: ACM, 2020, pp. 1–8. doi: 10.1145/3383455.3422540.

[19] M. Kopel and W. Szczurek, "Parallelization of Reinforcement Learning Algorithms for Video Games," in *INTELLIGENT INFORMATION AND DATABASE SYSTEMS, ACIIDS 2021*, vol. 12672, Cham: Springer International Publishing, 2021, pp. 195–207. doi: 10.1007/978-3-030-73280-6_16.

[20] T. Lan, S. Srinivasa, H. Wang, and S. Zheng, "WarpDrive: Extremely Fast End-to-End Deep Multi-Agent Reinforcement Learning on a GPU," *arXiv*, 2021. doi: 10.48550/arxiv.2108.13976.

[21] M. Balaz and P. Tarabek, "Tensor Implementation of Monte-Carlo Tree Search for Model-Based Reinforcement Learning," *Applied Sciences*, vol. 13, no. 3, p. 1406, 2023. doi: 10.3390/app13031406.

[22] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, et al., "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures," *arXiv*, 2018. [Online]. Available: https://doi.org/10.48550/arxiv.1802.01561.

[23] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, et al., "Massively Parallel Methods for Deep Reinforcement Learning," *arXiv*, 2015. [Online]. Available: https://doi.org/10.48550/arxiv.1507.04296.

[24] S. Huang, R. F. J. Dossa, C. Ye, J. Braga, D. Chakraborty, K. Mehta, and J. G. M. Araújo, "CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms," *Journal of Machine Learning Research*, vol. 23, no. 274, pp. 1-18, 2022. [Online]. Available: http://jmlr.org/papers/v23/21-1342.html.

[25] L. Programmer, "Machine Learning Examples," GitHub repository, 2023. [Online]. Available: https://github.com/lazyprogrammer/machine_learning_examples. [Accessed: 12-Sep-2024].

[26] P. Tabor, "Youtube Code Repository," GitHub repository, 2020. [Online]. Available: https://github.com/philtabor/Youtube-Code-Repository. [Accessed: 12-Sep-2024].

[27] A. Géron, *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, Sebastopol, CA: O'Reilly Media, Incorporated, 2017.

[28] L. Programmer, "PyTorch: Deep Learning and Artificial Intelligence," Udemy course, 2023. [Online]. Available: https://www.udemy.com/course/pytorch-deep-learning/. [Accessed: 12-Sep-2024].

[29] P. Tabor, "Proximal Policy Optimization (PPO) is Easy With PyTorch | Full PPO Tutorial," YouTube, 25 Dec. 2020. [Online]. Available: https://www.youtube.com/watch?v=hlv79rcHws0. [Accessed: 12-Sep-2024].

[30] N. Shadman, "Deep Reinforcement Learning for Automated Trading on HPC," GitLab repository, hosted by The University of Edinburgh, 2024. [Online] (Private repository).

[31] Intel, *Intel® Core™ i7-8550U Processor Product Specifications*, 2024. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/122589/intel-core-i7-8550u-processor-8m-cache-up-to-4-00-ghz.html [Accessed: 12-Sep-2024].

[32] Python Software Foundation, "Welcome to Python.org," 2023. [Online]. Available: https://www.python.org/ [Accessed: 12-Sep-2024].

[33] PyTorch Foundation, "PyTorch," [Online]. Available: https://pytorch.org. [Accessed: 12-Sep-2024].

[34] Ran Aroussi, "yfinance," Python Package Index (PyPI), [Online]. Available: https://pypi.org/project/yfinance/. [Accessed: 12-Sep-2024].

[35] TensorFlow, "TensorFlow," [Online]. Available: https://www.tensorflow.org/. [Accessed: 12-Sep-2024].

[36] OpenAI, "Baselines," GitHub repository, [Online]. Available: https://github.com/openai/baselines. [Accessed: 12-Sep-2024].

[37] A. Hill, "Stable Baselines," GitHub repository, [Online]. Available: https://github.com/hill-a/stable-baselines. [Accessed: 12-Sep-2024].

[38] DLR-RM, "Stable Baselines3," GitHub repository, [Online]. Available: https://github.com/DLR-RM/stable-baselines3. [Accessed: 12-Sep-2024].

[39] Ray Project, "RLlib: Scalable Reinforcement Learning," GitHub repository, [Online]. Available: https://github.com/ray-project/ray/tree/master/rllib. [Accessed: 12-Sep-2024].

[40] PyTorch, "PyTorch RL," GitHub repository, [Online]. Available: https://github.com/pytorch/rl. [Accessed: 12-Sep-2024].

[41] Fidelity, "Special rules for commodity ETFs," Fidelity, [Online]. Available: https://www.fidelity.com/learning-center/investment-products/etf/special-rules-commodity-etfs. [Accessed: 12-Sep-2024].

[42] The PyTorch Foundation, "PyTorch Profiler Recipe," *PyTorch Tutorials*, Sep. 14, 2023. [Online]. Available: https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html

[43] NVIDIA Corporation, "CUDA Toolkit Documentation v11.6.0," *NVIDIA Developer Documentation*, Feb. 2022. [Online]. Available: https://docs.nvidia.com/cuda/archive/11.6.0/

[44] N. N. Taleb, "Statistical Consequences of Fat Tails: Real World Preasymptotics, Epistemology, and Applications," arXiv:2001.10488 [stat.OT], 2022. [Online]. Available: https://arxiv.org/abs/2001.10488 [Accessed: September 14, 2024].

[45] Spitznagel, M. (2021). *Safe Haven: Investing for Financial Storms*. Hoboken, New Jersey: John Wiley & Sons.

[46] Zhang, J., Zhang, Z., Han, S., & Lü, S. (2022). Proximal Policy Optimization via Enhanced Exploration Efficiency. *Information Sciences*, 609, 750–765. https://doi.org/10.1016/j.ins.2022.07.111

[47] Cerebras Systems, "Cerebras Wafer-Scale Engine (WSE)," *Cerebras Systems*, 2023. [Online]. Available: https://cerebras.ai/product-chip/