epcc

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

Feasibility Study

Nabil Shadman

19 May, 2023

1. Introduction

Deep Reinforcement Learning (DRL) and High-Performance Computing (HPC) technologies are applied to solve problems in areas such as video games, large language models, autonomous driving, and automated trading. DRL models, especially when applied in a data-intensive field such as finance, require a vast amount of computational power. For example, AbdelKawy et al. developed a multi-stock DRL algorithm for automated trading that takes around 13 hours to train initially [1].

It is essential for traders to experiment with different models to discover profitable strategies. We believe utilising the computational power of HPC systems can make the process time efficient. In this study, we discuss the feasibility to build DRL models for automated trading on heterogeneous HPC systems (i.e., with a combination of CPUs and GPUs in our case). We introduce the essential concepts of DRL and HPC, discuss proof-of-concept (POC) implementation, and propose a project to develop, parallelise, and test automated trading models on our target HPC system (i.e., Cirrus). Our study validates that it is feasible to execute the project on Cirrus, and that we can achieve our project goals within the specified timeline (i.e., September 2023 – May 2024).

2. Background and Literature Review

In this section, we discuss the essential background and discuss past research in related areas.

2.1. Basic Concepts

Algorithmic trading is the use of computers to execute and monitor trades. The algorithms are designed to analyse financial data, detect trading opportunities, and execute trades based on predefined logic [1].

Reinforcement learning (RL) is an area of machine learning concerned with how intelligent agents ought to take actions in an environment to maximise cumulative reward (see Figure 1) [2]. A neural network is a machine learning model composed of multiple layers of interconnected nodes (or neurons), which enable it to learn complex representations of the data [2]. **Deep reinforcement learning (DRL)** combines RL algorithms with neural networks to enable artificial agents to learn and make decisions based on their environment [2].

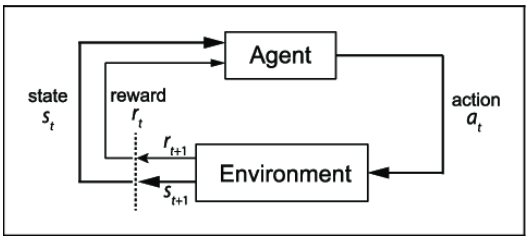

Figure 1: A typical RL architecture consisting of an agent taking actions in an environment and observing states and rewards [3].

High-performance computing (HPC) refers to the use of supercomputers and computer clusters to solve computation problems. A heterogeneous HPC system consists of different types of processors (e.g., CPU, GPU, FPGA) in a single cluster. GPU computing is the use of a graphics processing unit (GPU) to perform highly parallel independent calculations. GPU computing offloads the processing needs from the central processing unit (CPU) to achieve better performance. DRL models involve processing large amounts of data, such as images or sequences of states, and involve operations that can be performed in parallel, such as matrix multiplications [4]. GPUs have many cores that can perform these operations simultaneously, allowing for significant speedup in the training process. For instance, Stooke and Abbeel used a combination of CPUs and GPUs to speed up DRL models for video games [5].

2.2. Models

In this subsection, we discuss the hyperparameters of neural networks and the different types of DRL algorithms.

2.2.1. Hyperparameters

There are some hyperparameters (e.g., layers, activation function) in a neural network that can be tuned to optimise its performance. We introduce two commonly tuned hyperparameters: number of layers, and activation function.

A **layer** is a collection of neurons that perform a specific type of computation on the input data. Each layer receives input from the previous layer, performs a set of operations on it, and then passes the output to the next layer [6]. The number of hidden layers in a neural network is a hyperparameter that determines the depth of the network. Deeper networks can represent more complex functions but may also be prone to overfitting [6].

An **activation function** (e.g., sigmoid, ReLU, and tanh) is a mathematical function applied to the output of a neuron, which determines whether the neuron should activate or not [6]. The activation function introduces non-linearity into the output of the neuron, enabling neural networks to model complex relationships in the data [6].

2.2.2. Value-based Algorithm

In value-based DRL algorithms, an agent calculates a value function for the current state and greedily chooses the next action that results in the maximum expected return [1]. Q-value represents the expected cumulative reward an agent can receive by taking a particular action in each state and following the optimal policy thereafter [2]. **Deep Q-Network (DQN)** is a value-based algorithm, which uses a neural network to approximate, given a state, the different Q-values for each possible action at that state [2]. The pseudocode of DQN is provided in Appendix A.

2.2.3. Policy-based Algorithm

In policy-based DRL algorithms, the agent searches for an optimal policy that maximises the expected return [1]. A policy is represented by a neural network that takes a state as input and outputs a probability distribution of possible actions [2]. **Proximal policy optimization (PPO)** is a policy-based algorithm, which alternates between sampling data through environmental interaction and optimising a clipped surrogate objective function using stochastic gradient descent [2]. The clipped surrogate objective function improves training stability by limiting the size of the policy change at each step [2]. The pseudocode of PPO is provided in Appendix A.

2.2.4. Actor-critic Algorithm

Actor-critic algorithms combine value-based and policy-based approaches by maintaining two neural networks, one for the actor (i.e., policy) and one for the critic (i.e., value function) [1]. The actor network generates the policy, while the critic network estimates the expected return of the current policy and provides feedback to the actor to update its weights [1]. **Deep Deterministic Policy Gradient (DDPG)** is an actor-critic algorithm that combines an actor network that generates actions and a critic network that estimates values of actions, enabling it to learn from both the immediate reward and the expected future reward [2].

2.2.5. Model-based Algorithm

All the models discussed above are called **model-free algorithms** [2]. In model-free algorithms, an agent collects data, and uses the data to directly learn a Q function, or a policy, or both. There is another class of DRL algorithms called **model-based algorithms** [2]. In here, an agent learns a dynamics model of the world that enables it to simulate the behaviour of the world. The algorithm uses this world model to find a good Q function or a good policy. By doing so, it can do a lot of learning without needing to collect new data from the real environment [2]. As this is relatively a new area of research, the literature and development tools available are lesser than those of model-free algorithms. Hence, we do not investigate model-based algorithms in our project.

2.3. Related Literature

There are several researchers who have applied DRL algorithms in trading. For instance, AbdelKawy et al. proposed a DRL model to extract discriminant features of reduced dimensionality from high-dimensional raw financial data, and to generate trading decisions [1]. The researchers experimented with different agents (e.g., DDPG, DQN) in the study [1]. Ma et. al proposed a DRL model to learn both current state and long-term trends of the market, and to generate trading decisions [7]. The researchers employed the Double DQN (DDQN) algorithm, which is an improvement to the DQN algorithm, as the agent [7]. Yang et. al experimented with an ensemble DRL model (with A2C, PPO, and DDPG agents) to generate trading decisions [8]. Some of the models that we reviewed in the literature lack in-depth considerations of risk management. Large drawdowns due to market crashes can be hazardous for profitability [9].

The literature lacks in-depth analysis of DRL models for automated trading on HPC systems. Often, the researchers do not explicitly state the computing systems where they ran the models. The studies focus mostly on profitability compared to some baseline models (e.g., buy-and-hold). Therefore, it presents an opportunity for us to develop, parallelise, and test these models on an HPC system to evaluate potential advantages and challenges. There is, however, literature on parallelising DRL models in the video games domain. For instance, Kopel and Szczurek parallelised DQN and PPO algorithms for video games with both CPU-GPU and GPU only approaches [10]. Clemente et al. developed a framework to parallelise both value-based and policy-based DRL models for video games on a GPU [11]. We can use the findings from these studies in our investigation.

3. Preliminary Investigations and Findings

In this section, we discuss our POC work. We detail the hardware and software environments used, the experimental method, and the results and lessons learnt from preliminary investigations.

3.1. Environment

We used Cirrus, a Tier 2 UK National HPC system, in our POC work [12]. Table 1 details the hardware of the system [13]. We used Cirrus in our study because of its availability of GPU nodes, which is one of the targets of this project.

	Compute nodes	***GPU nodes***
Nodes	*280*	*36 (+2 older nodes)*
GPUs	*-*	*4 NVIDIA Tesla V100*
CPUs	*2 Intel Xeon E5-2695*	*2 Intel Xeon Gold 6248 (on 36 nodes), 2 Intel Xeon Gold 6148 (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 36 nodes), 256 GB (2x128) main memory (on 2 nodes), 64 GB (4x16) GPU memory (on all nodes)*

Table 1: Hardware details of Cirrus [13].

In our experiments, we used an open-source implementation of a DQN algorithm for multi-stock trading developed with the Python programming language [14][15]. There are implementations of the model in PyTorch and in TensorFlow, which are two widely used machine learning (ML) frameworks. We ran the models on the backend nodes of Cirrus, on both CPU and GPU nodes.

We used the pre-installed **pytorch/1.12.1** on CPU nodes and **pytorch/1.13.1-gpu** (optimised for GPUs) on GPU nodes when running the PyTorch implementation as these are the latest versions available on Cirrus. We used **tensorflow/2.10.0** on CPU nodes and **tensorflow/2.11.0-gpu** on GPU nodes when running the TensorFlow implementation. We were interested in observing the performance metrics of the application such as execution time and memory consumption. So, to profile the application, we used the built-in **profilers** of the respective frameworks [16][17]. Also, we were interested to observe whether the application was using the GPU, and if yes, to what extent. So, to monitor GPU metrics, we used **nvidia-smi**, which is a tool to measure metrics such as GPU utilisation, memory usage, and power usage.

3.2. Methodology

For our POC work, we had two main goals: (1) demonstrate that we can run available DRL model for trading with our hardware, and (2) learn the foundations of DRL to guide us in our development effort during dissertation.

In our first goal, the main objective was to ensure we have the capability to run available DRL model on both CPUs and GPUs and understand its performance. Therefore, we searched for pre-existing open-source models. The open-source DQN model experiment is relevant in our study as it informs us about the runtime, memory usage, CPU utilisation, GPU utilisation, and other metrics. We can use the metrics to estimate the performance metrics in the dissertation stage when we will scale the problem size. The profilers of the two frameworks and nvidia-smi were useful for observing these metrics. Also, the two versions (using the different frameworks) of the same algorithm enables us to understand any differences in performance between the two frameworks. Both implementations of the DQN model used the same dataset, which is historical stock prices of Apple, Motorola, and Starbucks [14][15].

Financial data typically consists of time-series data of a financial security (e.g., daily closing price of a company's stock traded in the New York Stock Exchange). Typically, the datasets are available freely from numerous APIs such as Yahoo Finance, IEX Cloud, Alpaca, and EODHD [4]. The first step is to fetch the required data from an API. Then, the data must be cleaned (e.g.,

imputing missing values, normalising data). This is done typically with a data processing tool such as the **pandas** library in Python. Typically, model training time increases as we increase the size of dataset. In our POC work, we use historical stock prices of Apple, Motorola, and Starbucks, which is included with the open-source model used in our experiments [14][15]. The data was clean and ready to be used out of the box. So, we did not have to process the data any further.

In our second goal, we reviewed multiple resources to understand the background of our research and identify opportunities where we can contribute to the financial engineering field. This involved researching academic papers, books, documentations of machine learning frameworks, open-source software, online tutorials, and other materials. For instance, a lecture series on DRL by Abbeel helped us understand the theoretical foundations of DRL [2]. Some open-source implementations of the DRL algorithms helped us map the pseudocodes to the implementations to develop practical understanding [19]. These resources will guide us in the dissertation phase when implementing the DRL algorithms.

All our data, code, and output are included in the code repository associated with the project [20]. The repository includes information on the files, instructions on reproducing the results, and a wiki to store notes.

3.3. Analysis of Experiments

Overall, we do not expect a significant difference in ease of development between the two machine learning frameworks. There is adequate support for both frameworks on the web. We observed that TensorFlow provides more verbose output (by default) both when logging results and when profiling performance of the application. However, we can use certain flags to stop verbose logging.

One challenge was to select which version of the frameworks to use as Cirrus has multiple versions available. For simplicity, we chose the latest available versions of both PyTorch and TensorFlow. However, we ran into a runtime error when using tensorflow/2.11.0-gpu on the GPU node. We worked with the Cirrus helpdesk team to resolve the issue, which required setting a certain flag to the module file. One observation was that the execution time of **tensorflow-2.9.1-gpu** was 20% less than that of **tensorflow/2.11.0-gpu**. The difference could be due to differences in how the framework's versions are compiled, and which specific libraries the versions are using. In the dissertation stage, we aim to configure and choose the version that returns the best performance for our models.

Table 2 details the execution times per episode of running the models out of the box on both CPU and GPU nodes. We observe that there is a significant difference in execution times between TensorFlow and PyTorch implementations. The relatively slower performance of TensorFlow could be due to suboptimal configurations of the library (e.g., TensorFlow version, specific math library used, versions of other libraries used in the framework) for the model. In the dissertation stage,

we aim to perform low-level configurations to reduce execution times as much as possible. Another reason for the difference in performance could be difference in number of threads the libraries are using by default. It was not clear to us how many threads the programs are using when running on both CPU and GPU nodes. We attempted to observe it using both **top** command and the profilers but did not conclusively determine it. In the dissertation stage, we aim to look deeper into this uncertainty as the number of threads will be important for accurately measuring scalability. We are currently investigating the documentations of the frameworks [21][22].

Model/Version	***Time per episode (seconds) (CPU)***	***Time per episode (seconds) (GPU)***
DQN TensorFlow	*93.0*	*62.2*
DQN PyTorch	*1.4*	*0.9*

Table 2: Execution times of training DQN models across different frameworks and across different nodes of Cirrus. We are currently investigating the cause of the considerable difference in performance between TensorFlow and PyTorch versions.

We ran the application on a GPU node of Cirrus, configuring to use only one GPU on the node. TensorFlow, by default, uses the GPU automatically when running the program on the GPU node [23]. We confirm this is the case by monitoring the program with nvidia-smi (see Figure 2). Also, TensorFlow enables us to print which operations are computed by which device (i.e., whether CPU or GPU), which is the useful to observe GPU usage. We observe that TensorFlow uses most of the GPU memory, i.e., around 15 GB of the total 16 GB.

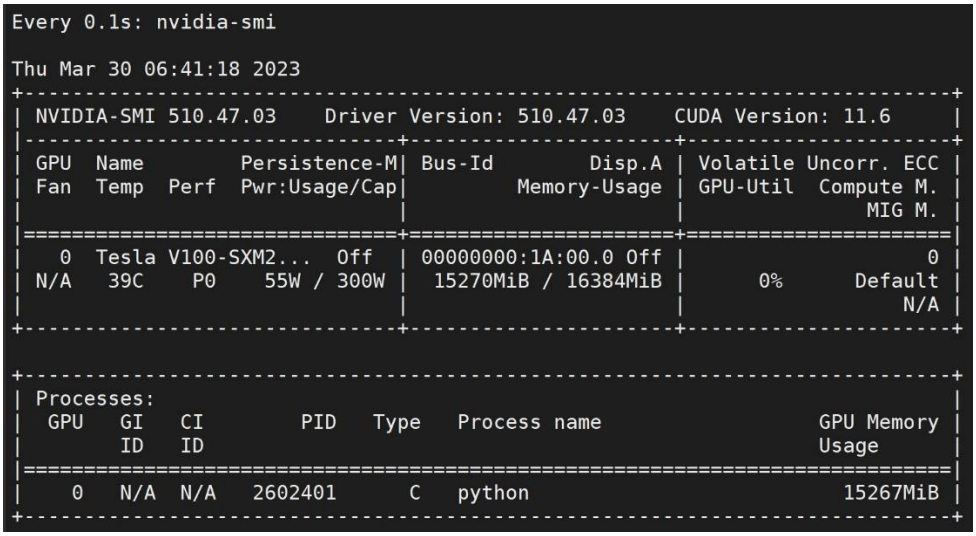

Figure 2: Output from **nvidia-smi** showing DQN TensorFlow version using most of the GPU memory.

PyTorch, by default, does not use the GPU. Both the data (i.e., tensors) and the model must be transferred to the GPU device for GPU computing [23]. When the computations are complete, the results can be transferred back to the CPU. We confirm that the GPU is not being used, based

on output from nvidia-smi and the PyTorch profiler. In the dissertation stage, we will manually transfer the relevant data and the model back and forth from the CPU to GPU to parallelise it. We observe that the relatively newer CPU on the GPU node (e.g., Intel Sky Lake) outperformed older CPU (e.g., Intel Broadwell) on the CPU node with the same PyTorch application, even though we did not tune the code for using GPUs on the GPU node (see Table 2).

4. Project Proposal

In this section, we discuss our project proposal in detail based on learnings from the literature review and the POC experiments. We categorise our requirements in accordance with the MoSCoW method. It is a technique in project management to prioritise requirements (in decreasing order of priority) according to **must have (M)**, **should have (S)**, **could have (C)**, and **would not have (W)** categories [24]. We omit requirements in the **W** category as there could be several possibilities in the category.

4.1. Must Have

These requirements form our minimum viable product.

- **Develop two variants of DRL models: one value-based (e.g., DQN), one policy based (e.g., PPO), each with a PyTorch and a TensorFlow version (i.e., 4 programs in total)**

DQN and PPO are good candidates for this requirement as there is adequate support (e.g., literature, open-source implementations) on the web, which will guide us in our development work. We provide the pseudocodes of the algorithms in Appendix A, which we will use in the dissertation phase. As we already have an open source DQN model implemented in both PyTorch and in TensorFlow, we will not develop DQN implementations from scratch. Instead, we will tune the configurations of the frameworks to minimise execution times of the sequential implementations. The PPO implementations will be developed from scratch.

- **Test each model on a CPU node and on a GPU node and measure performance**

We aim to measure expected return, execution time, speedup, memory consumption, power consumption, and cost of the models. The combination of metrics would be useful to compare the models from a broader perspective. The cost of the models will be calculated by hypothetical assumptions of running the models on a cloud HPC server (e.g., Amazon Web Service) with a similar hardware architecture of Cirrus.

We hypothesise that there is no difference in expected return between TensorFlow and PyTorch implementations. However, PyTorch implementations are expected to outperform TensorFlow implementations in execution time. Also, we hypothesise that PPO model will outperform DQN

model (in both expected return and in execution time) based on results observed in previous research from the video games domain [10].

We aim to compare CPU parallelisation (i.e., with multithreading) with GPU parallelisation. The parallelisation will be configured by using TensorFlow and PyTorch features. Our hypothesis is that GPU parallelisation will outperform CPU parallelisation.

- **Tune hyperparameters and measure performance**

We aim to tune the hyperparameters of the models (e.g., activation function, layers) and measure the performance metrics discussed above. This will enable us to choose the optimal hyperparameters for the trading models.

4.2. Should Have

These are requirements that we intend to execute if we find more time in the dissertation phase after carrying out the **M** requirements.

- **Develop an actor-critic DRL model (e.g., DDPG) with PyTorch and TensorFlow versions**

It would be sensible to build and test a DRL model of the third variety from the class of model-free algorithms. DDPG is a good candidate for this requirement as there is adequate literature on the model to guide us [25].

- **Test model on (CPU, GPU) nodes and measure performance**

We hypothesise that DDPG will outperform DQN in expected return based on performance observed in previous research [1]. However, we expect DDPG to not outperform PPO (in expected return) based on performance observed in previous research [12]. We do not have sufficient information to hypothesise on the execution time performance of DDPG.

4.3. Could Have

These are requirements we intend to execute if we find even more time after executing requirements from the **S** category.

- **Build risk management module to prevent large drawdowns**

It is best practice to have risk management built into the system. For example, the models can have a stop loss module that liquidates positions when drawdowns reach a certain level, e.g.,

10% [26]. Another alternative is to buy way out-of-the money put options, which can protect portfolios from large drawdowns in the event of a market crash [9]. We can use one such approach to build a risk management module for the models.

- **Forward test models to provide another layer of safety**

Forward testing models is another best practice in the financial industry. This is where we run a model in a simulated environment using live real-time data. This enables traders to detect further risks in the models (e.g., price slippage) and enables traders to take precautions [27]. There are APIs in the financial industry that allow traders to simulate models (e.g., Alpaca) [28]. We can use an API to forward test our models.

5. Workplan

In this section, we discuss the workplan of our project. We detail the time estimates and provide a Gantt chart of deliverables. The project is achievable within the imposed timeline. We have kept a 25% buffer in our time estimates to deal with any unforeseen circumstances. Also, the Gantt chart will enable us to evaluate progress regularly against planned timeline, and to make any changes to stay on track during the dissertation stage.

5.1. Time Estimates

Table 3 details the time estimates of our **Must Have** requirements. The time estimates of other categories are detailed in **Appendix B**. The time estimates are based on our experience from previous work. However, the project is dynamic in nature. The scope can be adjusted based on interest or progress in the dissertation phase. For example, we can experiment with an alternative DRL model if we find it more suitable for our research. The scope may be reduced or increased depending on progress.

Must have	***Hours***
Prepare data	
Fetch data	*10*
Preprocess data	*10*
Develop two DRL models (DQN, PPO), each with (PyTorch, TensorFlow) versions	
DQN-PyTorch	*20*
DQN-TensorFlow	*20*
PPO-PyTorch	*80*
PPO-TensorFlow	*80*

Test each model on (CPU, GPU) nodes and measure performance (expected return, execution time, speedup, memory, power, cost)	*60*
Tune hyperparameters (activation function, layers) and measure performance	*60*
Prepare documentation	
Report	*80*
Repository	*30*
Presentation	*30*
Subtotal	***480***
Buffer	***120***
Total	***600***

Table 3: Time estimates of the **M** category of requirements.

5.2. Gantt Chart

The Gantt chart in Figure 3 focuses on the broader requirements in the dissertation phase. It is to be noted that the requirements are not exactly dependent on one another. For instance, some components of the documentation (i.e., background, literature review) can be started earlier in the timeline.

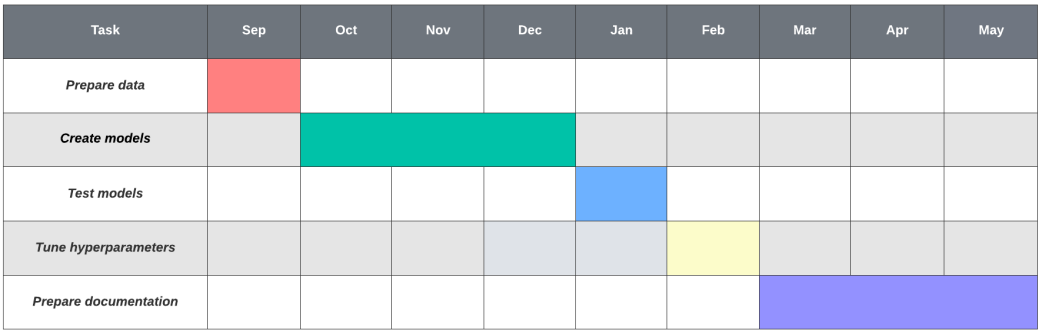

Figure 3: Gantt chart of the project deliverables. The timeline of the project is from September 2023 to May 2024.

6. Resources Estimation

In this section, we estimate the resources (in Table 4) that we need to execute our project.

Resources Estimation	*Formula*	*Subtotals*	*Total*
CPU (hours)			***1215***
TensorFlow	*1.250 second per episode x 1000 episodes x 30 runs x 6 hyperparameters x 3 data scaling x 3 models x 1.2 debugging*	*675.0*	
PyTorch	*1.000 second per episode x 1000 episodes x 30 runs x 6 hyperparameters x 3 data scaling x 3 models x 1.2 debugging*	*540.0*	
GPU (hours)			***405***
TensorFlow	*0.625 second per episode x 1000 episodes x 20 runs x 6 hyperparameters x 3 data scaling x 3 models x 1.2 debugging*	*225.0*	
PyTorch	*0.500 second per episode x 1000 episodes x 20 runs x 6 hyperparameters x 3 data scaling x 3 models x 1.2 debugging*	*180.0*	
Storage (GB)			***26***
TensorFlow	*20 MB per file x 50 runs x 6 hyperparameters x 3 models*	*17.2*	
PyTorch	*10 MB per file x 50 runs x 6 hyperparameters x 3 models*	*8.6*	
Data	*75 MB per file x 6 copies*	*0.4*	

Table 4: Resource estimates of the project.

We estimated resources based on data observed in the preliminary investigations. For instance, compute resources (i.e., CPU or GPU hours) are estimated by:

compute resources = time per episode x episodes x runs x hyperparameter changes x data scaling factor x models x debugging factor

We do not expect any additional costs (i.e., in monetary terms) in our project. We do not need any permission to access data as they will be sourced from open data APIs. After considering ethical issues of the project, we confirm that we do not need any ethical approvals for the project. The ethical considerations discussed were: (1) environmental impact (e.g., energy cost of using GPUs), (2) risk to readers (e.g., one may try our model in live trading), (3) conflict of interest (e.g., there is a chance that we can profit from the models).

7. Risk Analysis

In the final section, we assess the risks of the project and discuss risk controls (Table 5).

Risk Description	***Likelihood***	***Impact***	***Risk Rating***	***Control***
Schedule slippage	*Medium*	*High*	*High*	*Review progress weekly, avoid scope creep*
Resource overrun	*Low*	*High*	*Medium*	*Prevent waste (e.g., set reasonable wall time, delete unnecessary files)*
Loss of work	*Low*	*High*	*Medium*	*Update repository (i.e., GitLab) and report (i.e., Overleaf) regularly*
System outage	*Low*	*Low*	*Low*	*Work on other parts (e.g., development, report)*

Table 5: Risk analysis of the project.

References

[1] R. AbdelKawy, W. M. Abdelmoez, and A. Shoukry, “A synchronous deep reinforcement learning model for automated multi-stock trading,” *Progress in Artificial Intelligence*, pp. 83–97, Jan. 2021. doi:10.1007/s13748-020-00225-z

[2] P. Abbeel, “The Foundations of Deep RL Series,” YouTube, https://www.youtube.com/watch?v=2GwBez0D20A&list=PLwRJQ4m4UJjNymuBM9RdmB3Z9N5-0IlY0 (accessed May 19, 2023).

[3] ResearchGate, “Reinforcement learning schematic,” ResearchGate, https://www.researchgate.net/figure/Reinforcement-learning-schematic-Reinforcement-learning-RL-can-be-formulated-as-a_fig4_322424392 (accessed May 19, 2023).

[4] N. Sanghi, *Deep Reinforcement Learning with Python*. New York, NY: Apress, 2021.

[5] A. Stooke and P. Abbeel, “Accelerated methods for deep reinforcement learning,” arxiv.org, https://arxiv.org/pdf/1803.02811.pdf (accessed May 19, 2023).

[6] A. Géron, *Hands-On Machine Learning with Scikit-Learn & TensorFlow*. Sebastopol, CA: O’Reilly, 2017.

[7] C. Ma, J. Zhang, J. Liu, L. Ji, and F. Gao, “A parallel multi-module deep reinforcement learning algorithm for stock trading,” *Neurocomputing*, vol. 449, pp. 290–302, Apr. 2021. doi:10.1016/j.neucom.2021.04.005

[8] H. Yang, X.-Y. Liu, S. Zhong, and A. Walid, “Deep Reinforcement Learning for Automated Stock Trading: An ensemble strategy,” *SSRN Electronic Journal*, Nov. 2020. doi:10.2139/ssrn.3690996

[9] M. Spitznagel, *Safe Haven*. Hoboken, NJ: Wiley, 2021.

[10] M. Kopel and W. Szczurek, “Parallelization of reinforcement learning algorithms for video games,” *Intelligent Information and Database Systems*, pp. 195–207, Apr. 2021. doi:10.1007/978-3-030-73280-6_16

[11] A. V. Clemente, H. N. Castejón, and A. Chandra, “Efficient parallel methods for deep reinforcement learning,” arXiv.org, https://arxiv.org/abs/1705.04862 (accessed May 19, 2023).

[12] EPCC, Cirrus, https://cirrus.readthedocs.io/en/main/ (accessed May 19, 2023).

[13] EPCC. Cirrus Hardware. Cirrus.ac.uk, 2022. [Online]. Available: https://www.cirrus.ac.uk/about/hardware.html. [Accessed: 19- May- 2023].

[14] lazyprogrammer, “machine learning examples pytorch,” GitHub, https://github.com/lazyprogrammer/machine_learning_examples/tree/master/pytorch (accessed May 19, 2023).

[15] lazyprogrammer, “machine learning examples tensorflow 2.0,” GitHub, https://github.com/lazyprogrammer/machine_learning_examples/tree/master/tf2.0 (accessed May 19, 2023).

[16] PyTorch, “PyTorch Profiler,” PyTorch, https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html (accessed May 19, 2023).

[17] TensorFlow, “Optimize TensorFlow performance using the Profiler,” TensorFlow, https://www.tensorflow.org/guide/profiler (accessed May 19, 2023).

[18] AI4Finance-Foundation, “FinRL: Financial Reinforcement Learning,” GitHub, https://github.com/AI4Finance-Foundation/FinRL (accessed May 19, 2023).

[19] C. Huang, “CleanRL (Clean Implementation of RL Algorithms),” GitHub, https://github.com/vwxyzjn/cleanrl (accessed May 19, 2023).

[20] N. Shadman, “Comparison of Deep Reinforcement Learning Models for Automated Trading on Heterogeneous HPC System,” GitLab, (accessed May 19, 2023).

[21] PyTorch, “CPU threading and TorchScript inference,” PyTorch 2.0 documentation, https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html#tuning-the-number-of-threads (accessed May 19, 2023).

[22] TensorFlow, “Module: tf.config.threading,” TensorFlow, https://www.tensorflow.org/api_docs/python/tf/config/threading (accessed May 19, 2023).

[23] M. Liu, “A Comparison of PyTorch and TensorFlow for Reinforcement Learning,” The University of Edinburgh, (accessed May 19, 2023).

[24] Wikipedia, “MoSCoW method,” Wikipedia, https://en.wikipedia.org/wiki/MoSCoW_method (accessed May 19, 2023).

[25] T. P. Lillicrap et al., “Continuous control with deep reinforcement learning,” arXiv.org, https://arxiv.org/abs/1509.02971 (accessed May 19, 2023).

[26] U.S. Securities and Exchange Commission, “Stop Order,” sec.gov, https://www.sec.gov/answers/stopord.htm (accessed May 19, 2023).

[27] J. Folger, “Backtesting and forward testing,” Investopedia, https://www.investopedia.com/articles/trading/10/backtesting-walkforward-important-correlation.asp (accessed May 19, 2023).

[28] Alpaca, “Paper Trading,” Alpaca Docs, https://alpaca.markets/docs/trading/paper-trading/ (accessed May 19, 2023).

[29] V. Mnih et al., “Playing Atari with deep reinforcement learning,” arXiv.org, https://arxiv.org/abs/1312.5602 (accessed May 19, 2023).

[30] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” arXiv.org, https://arxiv.org/abs/1707.06347 (accessed May 19, 2023).

Appendix A: Pseudocodes

In this appendix, we detail the pseudocodes of the **DQN** and **PPO** algorithms.

Algorithm 1: DQN [29]

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 [29]. The replay memory stores experience tuples to be randomly sampled for training the Q-network, which improves sample efficiency and reduces correlation between consecutive experiences [29]. The epsilon-greedy policy balances exploration and exploitation during action selection, and the Bellman equation is used to compute the target Q-values, which are updated during training to improve the accuracy of the Q-network [29]. The C parameter determines how often the Q-network weights are copied to the target Q-network to stabilise the training process [29].

Algorithm 2: PPO [30]

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 steps 3 for a specified number of iterations or until convergence.

In this algorithm, the policy network learns to output a probability distribution over actions given a state, while the value network estimates the expected return for each state [30]. The advantages

represent the advantage of taking an action compared to the average value of actions in that state [30]. The surrogate objective function measures how the policy performance improves based on the advantages and the ratio between new and old policy probabilities [30]. Clipping the ratio ensures that the policy update does not deviate too much from the old policy, adding stability to the learning process [30]. 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 [30].

Appendix B: Time Estimates of Additional Investigations

In this appendix, we provide the time estimates of the requirements from the **S** and **C** categories of our prioritisation.

Should have	***Hours***
Develop an additional DRL model (e.g., DDPG)	
DDPG-PyTorch	*80*
DDPG-TensorFlow	*80*
Test model on (CPU, GPU) nodes and measure performance	*30*

Table 6: Time estimates of the **S** category of requirements.

Could have	***Hours***
Build risk management module to prevent large drawdowns	
DQN-PyTorch	*50*
Forward test models to provide another layer of safety	
DQN-PyTorch	*50*

Table 7: Time estimates of the **C** category of requirements.