|epcc|

A Comparison of PyTorch and TensorFlow for Reinforcement Learning

Mingyang Liu

August 19, 2022

MSc in High Performance Computing with Data Science

The University of Edinburgh

Year of Presentation: 2022

Abstract

With the development of artificial intelligence technology, reinforcement learning, a new branch of machine learning, has more and more research results and applications[8]. For example, it has proven its worth in the field of games and robotics. However, building and training algorithmic models by hand takes a lot of work every time. Therefore, many frameworks have emerged in recent years to improve the efficiency of developing algorithmic models. For example, machine learning frameworks such as TensorFlow, PyTorch, Caffe, CNTK, etc., have been rapidly developed and widely used. TensorFlow and PyTorch are two of the most widely used machine learning frameworks. However, for those who want to train reinforcement learning algorithms, the question is which framework to choose for training their algorithms in which situation. Therefore, this project compares the performance of the frameworks TensorFlow and PyTorch in training reinforcement learning algorithms using performance metrics such as return with episodes, return with training time, memory with time and CPU consumption. The performance tests were conducted on three different operating systems: Windows 10, Ubuntu 20.04, and Cirrus. The performance of the two frameworks was also compared in Cirrus with a different number of CPUs and GPUs. Furthermore, this dissertation analyses the usability of TensorFlow and PyTorch. The result is a set of qualitative and quantitative recommendations for the choice of framework to be used for training reinforcement learning algorithms.

Contents

1 Introduction **1**
- 1.1 Project Motivation . . . 1
- 1.2 Project Objectives . . . 2
- 1.3 Chapters Breakdown . . . 3

2 Background theory **4**
- 2.1 Machine Learning . . . 4
- 2.2 Reinforcement Learning . . . 5
- 2.3 Key Concepts in RL . . . 5
- 2.4 Kinds of RL Algorithms . . . 7
  - 2.4.1 Value-based and Policy-based Methods . . . 7
  - 2.4.2 Monte Carlo and Temporal Difference Methods . . . 8
  - 2.4.3 On-policy and Off-policy Methods . . . 9
- 2.5 PPO . . . 9
- 2.6 DQN and Rainbow DQN . . . 10
- 2.7 Benchmark . . . 11
  - 2.7.1 OpenAI Gym . . . 11
  - 2.7.2 CartPole-v0 . . . 11
- 2.8 PyTorch, TensorFlow and Related Work . . . 11

3 Methodology **13**
- 3.1 RL Baseline Algorithms Selection . . . 13
  - 3.1.1 Changes to PPO . . . 13
  - 3.1.2 Changes to Rainbow DQN . . . 14
- 3.2 RL Testing Benchmark Selection . . . 14
- 3.3 Metrics for Performance Comparison . . . 15
  - 3.3.1 Return . . . 15
  - 3.3.2 Training Time . . . 16
  - 3.3.3 Memory Usage and GPU Usage . . . 16
  - 3.3.4 CPU Usage . . . 16
- 3.4 Operating Environments Selection . . . 17
  - 3.4.1 Operating Systems . . . 17
  - 3.4.2 Virtual Workstation . . . 17
  - 3.4.3 Environment Manager . . . 18
- 3.5 Experimental Design . . . 19

3.5.1 Hardware Resources . . . 19
3.5.2 Controls . . . 20
3.5.3 Automation of Testing Scripts . . . 21
3.5.4 Evaluation of the data . . . 22
3.6 Usability . . . 24

4 Experimental Works **25**
4.1 Implementation of Code . . . 25
4.1.1 PPO and Rainbow DQN . . . 25
4.1.2 Output . . . 25
4.2 Test Steps . . . 26
4.2.1 Build Operating Systems . . . 26
4.2.2 Build Environment Manager . . . 27
4.2.3 Hyperparameters Tuning . . . 27
4.2.4 Running Test Scripts . . . 29
4.3 Automation of Testing Scripts . . . 29
4.3.1 `YAML` and Requirements File . . . 29
4.3.2 `SLURM` and Shell Scripts of Running Job . . . 29
4.3.3 Job Chaining . . . 30

5 Results and Analysis **31**
5.1 Performance analysis . . . 31
5.1.1 PyTorch Versus TensorFlow . . . 31
5.1.2 Cirrus Versus Ubuntu20.04 . . . 34
5.1.3 Differences with Different CPUs . . . 37
5.1.4 Differences with Different GPUs . . . 39
5.2 Usability Analysis . . . 41
5.2.1 Ease of learning . . . 41
5.2.2 Loading Data . . . 42
5.2.3 Device management . . . 42
5.2.4 Visualisation tools . . . 43
5.2.5 Graph Calculations . . . 43
5.2.6 Parallel execution and training . . . 44
5.2.7 Deployment . . . 44
5.2.8 Serialization . . . 45
5.2.9 Compatibility and Scalability . . . 45
5.3 Summary . . . 46
5.3.1 Performance . . . 46
5.3.2 Usability . . . 46

6 Conclusions **48**

List of Tables

3.1 Hardware Information of Windows 10 19
3.2 Hardware Information of Ubuntu 20.04 19
3.3 Hardware Information of Cirrus with 2 CPU Cores 20
3.4 Hardware Information of Cirrus with 32 CPU Cores in Two Nodes 20
3.5 Hardware Information of Cirrus with 40 CPU Cores and 1/2/3/4 GPU(s) . . . 21
3.6 Test Parameters under Different OS 21
3.7 Test Parameters under Different Numbers of CPU 22
3.8 Test Parameters under Different Types of Algorithms 22
3.9 Test parameters under Different Numbers of GPU 23
3.10 The Packages Versions in the Virtual Test Environment 23

List of Figures

1.1 Paper Implementations grouped by framework (from 2017 to 2022) 2

2.1 Subdisciplines of Machine Learning 5
2.2 Learning Process of Reinforcement Learning 6
2.3 Interaction and communication between agent and environment[10] 6
2.4 Value-based and Policy-based RL 7
2.5 Value-based and Policy-based RL Algorithms 8
2.6 Monte Carlo and Temporal Difference RL Algorithms 9
2.7 Kinds of RL Algorithms 10
2.8 The `CartPole` Environment Scenario 11

3.1 MontainCar[7] 15
3.2 CartPole[6] 15
3.3 Topology of Operating Systems for Testing 18

4.1 Tensorboard Visualisation Tool 26

5.1 Return with Episodes of PPO 32
5.2 Return with Episodes of Rainbow 32
5.3 Return with Time of PPO 33
5.4 Return with Time of Rainbow 33
5.5 CPU with Time of PPO 33
5.6 CPU with Time of Rainbow 33
5.7 Memory with Time of PPO 34
5.8 Memory with Time of Rainbow 34
5.9 Return with Episode of PPO 35
5.10 Return with Episode of Rainbow 35
5.11 Return with Episode of PPO 35
5.12 Return with Episode of Rainbow 35
5.13 Return with Time of PPO 36
5.14 Return with Time of Rainbow 36
5.15 Return with Time of PPO 36
5.16 Return with Time of Rainbow 36
5.17 CPU with Time of PPO 37
5.18 CPU with Time of Rainbow 37
5.19 CPU with Time of PPO 37

5.20 CPU with Time of Rainbow 37
5.21 Time with Virtual Memory of PyT-PPO 38
5.22 Time with Virtual Memory of PyT-Rb 38
5.23 Time with Virtual Memory of TF-PPO 38
5.24 Time with Virtual Memory of TF-Rb 38
5.25 CPU Cores with Integral Return 39
5.26 CPU Cores with Integral GPU 39
5.27 CPU Cores with Integral Virtual Memory 39
5.28 CPU Cores with CPU per Step 39
5.29 CPU Cores with Virtual Memory per Step 39
5.30 CPU Cores with Integral Return 40
5.31 CPU Cores with Integral GPU 40
5.32 CPU Cores with Integral Virtual Memory 40
5.33 CPU Cores with CPU per Step 40
5.34 CPU Cores with Virtual Memory per Step 40
5.35 GPU with Integral CPU Percent 41
5.36 GPU with Integral Virtual Memory 41
5.37 GPU with Integral Return 41
5.38 GPU with CPU per Step 41
5.39 GPU with Virtual Memory per Step 41

Acknowledgements

First, I would like to express my gratitude to my two mentors, Dave McKay and Charaka Palansuriya. Despite their busy schedules, they met with me weekly to exchange ideas and discuss solutions to problems. So, Dave and Charaka often took more time to help me with the project than the half-hour weekly meetings. Even during spring teaching holiday, Dave kept meeting me at home to help me and Charaka and Dave gave me very valuable advice when testing algorithms and writing my dissertation, which is very touching. They also helped me a lot in completing this project, including the phase of setting up the environment, algorithm training problems, performance testing ideas, automation solutions for training, scheduling, and writing the paper. I can say that I could not have completed this work without their help.

Secondly, I would like to express my sincere gratitude to the course project manager Ben Morse, the course assistant Jemma Auns, the dissertation team, and the support team at Cirrus. They were always very helpful and patient when I encountered difficulties due to lack of knowledge and information during the project preparation and dissertation stages. For example, they provided me with course materials, requested resources and helped me with any problems I encountered during my Cirrus project. Without their help, I would not have been able to get through the project preparation and dissertation phases so smoothly.

I am also grateful to my EPCC classmates for guiding me and communicating with me during my studies. During the dissertation period, we talked to each other, met and travelled together, which took the stress away, made the time more enjoyable and provided a good learning environment for me to successfully complete my studies.

Finally, I would be remiss if I did not mention my parents and wife. Thanks to their encouragement and support, I was able to complete my studies at postgraduate level.

Chapter 1

Introduction

1.1 Project Motivation

In the last two decades, machine learning has become a multidisciplinary discipline, widely used in areas such as computer vision, recommender systems, natural language processing, strategic gaming and robotics[8]. In layman's terms, machine learning is a cutting-edge technology that enables computers to learn as much or more than humans and extract valuable knowledge from large, complex data. The process of human learning can be summarised as input, integration and output. It can be understood that human learning is the process of developing a certain understanding or summarising certain rules for a class of problems based on experience and then using this knowledge to make judgements about new problems. However, there are many limitations to the human learning process, such as slow learning speed, easy forgetting and limited life expectancy. The computational speed of computers combined with various machine learning algorithms that quickly learn from data and calculate valuable predictions overcomes the limitations of human learning. Currently, machine learning methods are divided into three main categories: supervised learning, unsupervised learning and reinforcement learning, based on the type of "signal" or "feedback" available to the learning system.

As machine learning research becomes more popular, there is an explosion of machine learning tools. The development of machine learning algorithms is complex and time-consuming. Machine learning frameworks such as TensorFlow, PyTorch, Caffe, Torch, etc. have emerged and are widely used to improve development efficiency and simplify the process of building algorithmic models. Figure 1.1 shows the proportion of frameworks used to implement algorithms in machine learning work from 2017 to 2022, as well as the proportion of repository time used to create algorithms in implemented work[11]. The main comparisons are for the mainstream languages TensorFlow, PyTorch, Caffe2, JAX, MXNet, PaddlePaddle, Torch, MindSpace and other languages and frameworks. Since Google released TensorFlow 1.0 in 2015 and PyTorch from FaceBook on GitHub in 2016, these two frameworks have gradually become two of

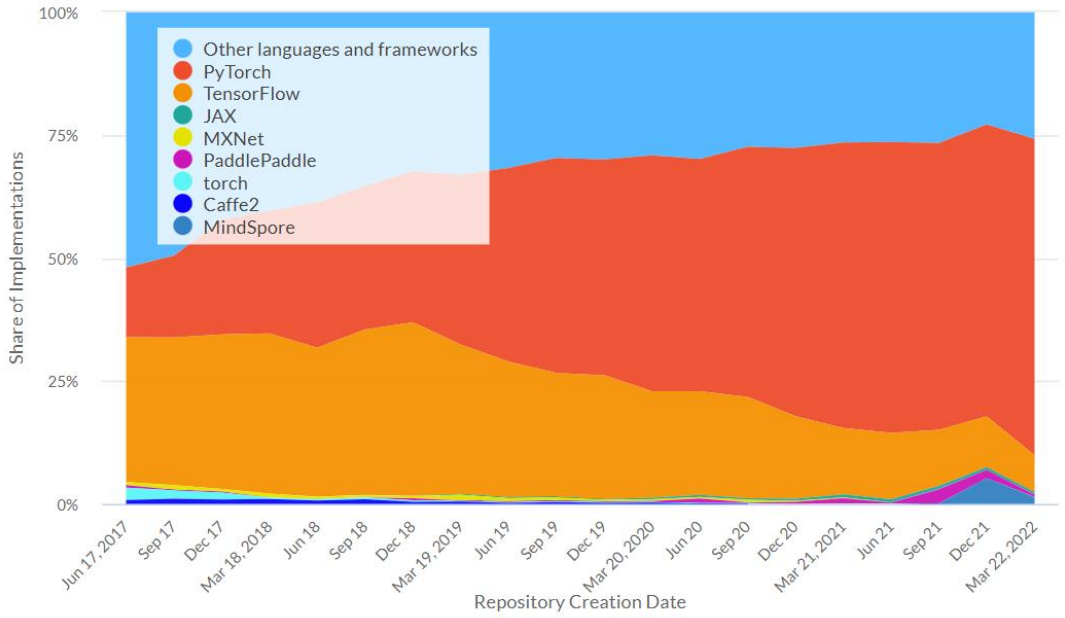

Figure 1.1: Paper Implementations grouped by framework (from 2017 to 2022)

the most popular frameworks for implementing machine learning algorithms. In recent years, some companies have experienced the phenomenon of mutual migration of TensorFlow and PyTorch algorithm code. At the same time, the choice of which of these frameworks to use before training an algorithm often leaves researchers, collaborators, and learners undecided. As a result, the question was whether we should use TensorFlow or PyTorch as the algorithmic framework.

1.2 Project Objectives

Reinforcement learning, one of the three major areas of research in machine learning, is being used in applications such as strategy games and robotics. In 2017, AlphaGo, developed by Google DeepMind in London, UK, shocked the world with its victory over Go master Ke Jie, whose core algorithm is a reinforcement learning algorithm[4]. Reinforcement learning algorithms, like other machine learning algorithms, need a proper framework to improve the efficiency of developing, building and training models by building models and training the algorithms. Therefore, there is also an urgent need for a set of informed recommendations on whether to choose TensorFlow or PyTorch as the framework for training reinforcement learning algorithms. In 2019, Simmons' team conducted an artificial neural networks (ANNs) analysis to test and compare the training time, memory consumption and usability of the two frameworks. It concluded that TensorFlow and PyTorch have the same accuracy, with PyTorch requiring less training time and TensorFlow requiring less memory, and PyTorch being easier to programme and TensorFlow being more flexible in terms of customisation. In the field of reinforce-

ment learning, there are no publications so far. Therefore, this work aims to compare the performance of the two frameworks based on two reinforcement learning algorithms, PPO and Rainbow DQN, in terms of returns over action, returns over training time, throughput, memory consumption and CPU consumption. The tests were conducted on three different operating systems, Windows 10, Ubuntu 20.04, and Cirrus. The usability of the two frameworks was also compared. Finally, the results are analysed to provide a set of recommendations for choosing TensorFlow or PyTorch based on reinforcement learning algorithms.

1.3 Chapters Breakdown

This dissertation is divided into six main chapters. Chapter 2 presents the basic knowledge required for this thesis and the relevant research background. Chapter 3 presents the ideas for the selection of the algorithms in the project, the ideas for the selection of the test environment, the choice of the performance metrics, the running environment, and the specific experimental design and the perspectives for the usability analysis. Chapter 4 presents the practical work, such as the specific changes to the algorithm under test, the processing of the training results, and the steps for testing, including setting up the test environment, tuning the hyperparameters, and writing the automation script. Chapter 5 contains the results and detailed analysis of the performance tests of the two frameworks, as well as an analysis of the usability of the two frameworks. Finally, Chapter 6 briefly summarises some recommendations for deciding between TensorFlow and PyTorch when someone wants to train a reinforcement learning algorithm, some of the difficulties encountered in implementing the project and how they were overcome, and suggestions for further work in the future.

Chapter 2

Background theory

2.1 Machine Learning

It is helpful to introduce machine learning before we turn to reinforcement learning. It was first introduced in 1952 when Arthur Samuel, an American engineer IBM, developed a chess programme that beat many famous players and introduced the term "Machine Learning". Machine learning is a very broad term, a discipline that uses neural networks or other mathematical models to learn how to interpret complex patterns. Simply put, machine learning means learning patterns from historical data and then applying those patterns to the future[16]. Currently, machine learning is divided into three main categories: supervised learning, unsupervised learning, and reinforcement learning, depending on the type of "signal" or "feedback" available to the learning system, as shown in Figure 2.1.

The three branches of machine learning are distinguished by the study of problems in different scenarios. Supervised learning, in Layman's terms, uses labelled data to derive a predictive model and then uses the trained model to compute new data for predictive purposes. For example, the model parameters are trained using past weather data such as humidity, temperature and wind direction at a given time and the current weather data for the next few hours. Then the trained model is used to predict the weather for the next few hours based on the current humidity, temperature, and wind direction. Second, unsupervised learning, where unlabelled data is trained to make inferences, finds the implicit structure. And then the trained model can be used to predict the outcome of new data. For example, in classical cluster analysis, the hypothetical scenario is to classify a bunch of fruit, find out what are apples, oranges and bananas, and use a trained model to classify the new fruits. In reinforcement learning, however, the data comes exclusively from the environment in question. The reward resulting from the state of the environment and the feedback received after a series of actions is used to correct the next action and finally reach the process that leads to the best reward. For example, if the scenario is to balance a pencil vertically in our hand, we need to constantly move our hand to keep it the pencil balanced as best we can. By observing

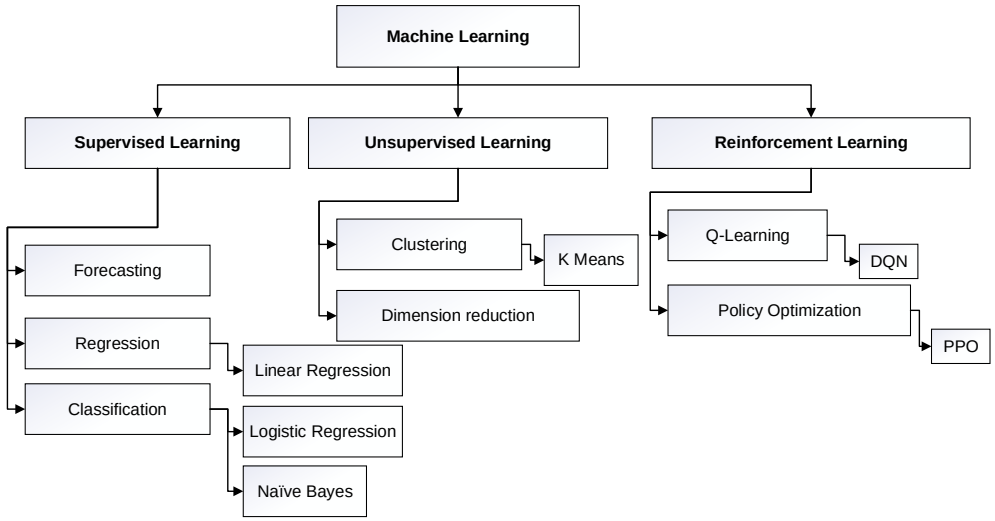

Figure 2.1: Subdisciplines of Machine Learning

the state of the pencil and the feedback we receive from the hand after some actions, we correct our next action to eventually reach a state of absolute or relative balance, absolute balance being the ideal goal in this scenario.

2.2 Reinforcement Learning

Reinforcement learning is a new subfield of machine learning, abbreviated RL. The concept of reinforcement learning became known to the public when Alpha Go defeated the world's number one Ke Jie at the Wuzhen Go Summit 2017. Theoretically, as shown in Figure 2.2 , reinforcement learning finds the best course of action for any given environmental condition through a simple trial-and-error process. The machine learning model introduces a random rule at the beginning and receives a certain number of points called rewards for each action. This process is repeated until the model reaches a goal, such as winning or losing a game. Then the round or episode ends, and the game is reset to its initial state. As the model goes through several rounds, it remembers which behaviours are more likely to lead to a better outcome and thus finds the best behaviour for a given state, i.e., the best strategy. The following four sections introduce reinforcement learning in terms of algorithm classification, agent and environment, and the PPO and Rainbow DQN algorithms, which introduce reinforcement learning from a classification and scenario-based perspective.

2.3 Key Concepts in RL

To understand the basics of reinforcement learning, the algorithms and the corresponding code implementations, an understanding of the key concepts of reinforcement learn-

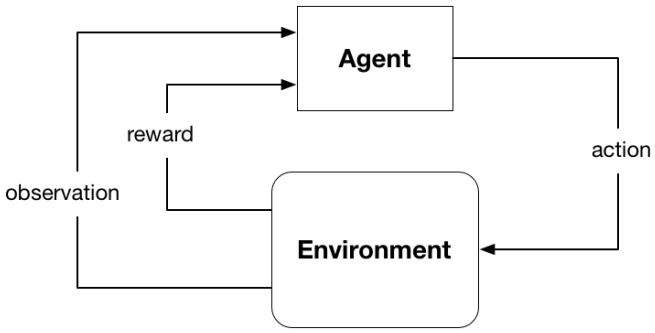

Figure 2.2: Learning Process of Reinforcement Learning

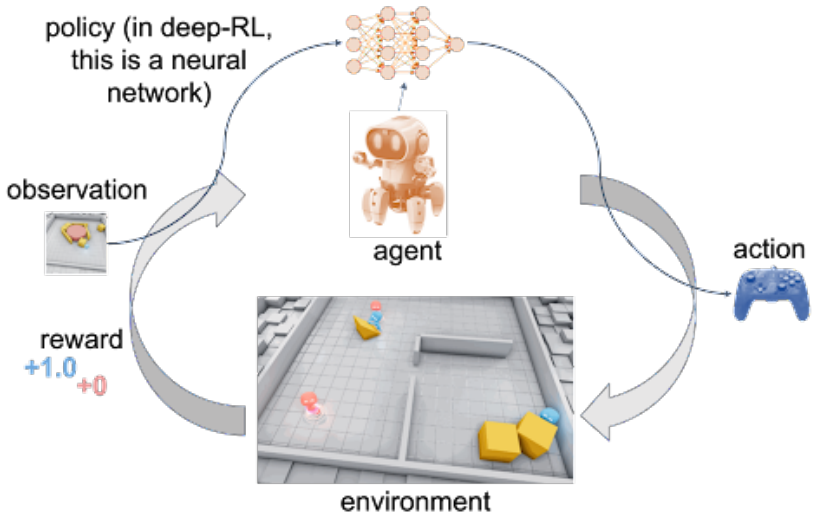

Figure 2.3: Interaction and communication between agent and environment[10]

ing is required. One of the most important features of reinforcement learning is the "agent" and the "environment". The Figure 2.3 below shows the interaction and communication between the agent and the environment. The environment is the place where the agent interacts and exists. In reinforcement learning, one or more agents interact in an environment, which can be a simulated scenario such as Cart Pole. At each step of the interaction, the agent observes the state of the environment (i.e. the "state" of the environment) and then decides on the next "action" to take. At the same time, the agent perceives the environment's "reward signal", i.e. a certain number that tells the agent how good or bad the current state of the environment is. The agent learns from this repeated cycle of trials in a sequence known as an "episode", which begins with the initial observation of a "successful" or "unsuccessful" "action" that leads the environment to its final state. The agent's goal is to maximise the cumulative reward, referred to as the "payoff". Reinforcement learning algorithms are, among other things, a way for agents to learn behaviours to achieve their goals. In addition, continuous learning leads to an optimal solution, i.e. the rules that the agent uses to decide which actions to take, called "policies".

2.4 Kinds of RL Algorithms

There are many types of reinforcement learning algorithms as well as many perspectives on the classification of reinforcement learning algorithms. This dissertation does not deal with model-based algorithms, so they are not covered here. This section focuses on three common classifications based on model-free algorithms, namely value-based and policy-based methods, Monte Carlo and Temporal Difference methods, and on-policy and off-policy methods.

2.4.1 Value-based and Policy-based Methods

There are two main types of policy optimisation: value-based methods and policy-based methods. In addition, the combination of value-based and policy-based methods has led to actor-critic type algorithms and other algorithms, as shown in Figure 2.4 .

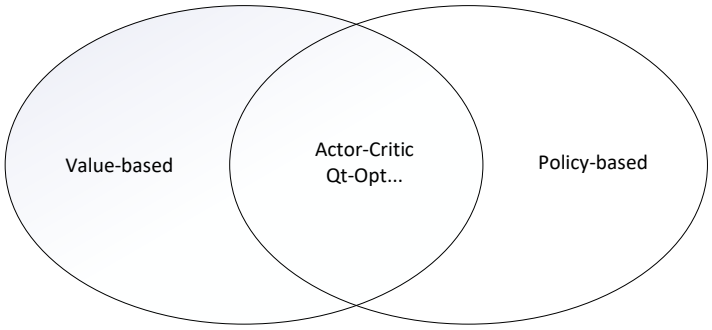

Figure 2.4: Value-based and Policy-based RL

The value-based class of algorithms in reinforcement learning is a class of algorithms that search for the value function without having an explicit policy. In this concept, it is worth noting that the algorithm is trained to focus on both current and long-term gains. So with Value Based, the value is not the reward given by the environment, but is calculated by a complex formula, usually using the Bellman equation, which is not explained here. Common value-based algorithms include Q-learning, Deep Q-Network (DQN) and its variants: e.g. Prioritised Experience Replay (PER), Dueling DQN, Double DQN, Retrace, Noisy DQN, Distributed DQN, Rainbow DQN, etc.

The policy-based class of reinforcement learning algorithms is a class of algorithms that rely on a policy without a value function. That is, the policy is directly optimised and the cumulative reward is maximised by iterative updates of the policy. Common policy-based algorithms include Policy Gradient (PG), Trust Region Policy Optimisation (TRPO), Proximal Policy Optimisation (PPO), etc.

The combination of value-based and policy-based algorithms has led to the class of Actor-Critic algorithms, where the current policy is used to estimate the value and improve the policy. (AC) algorithm and a number of improvements: Asynchronous Domi-

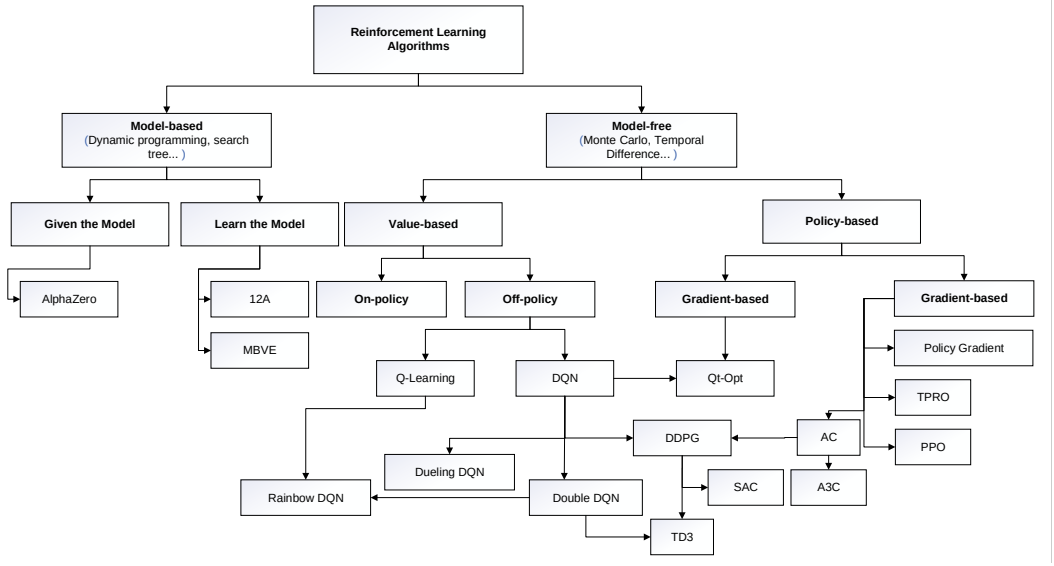

Figure 2.5: Value-based and Policy-based RL Algorithms

nance Actor-Critic (A3C), Deep Deterministic Policy Gradient (DDPG), Twin Delayed Deep Deterministic Policy Gradient (TD3), Soft Actor-Critic (SAC), etc.

In summary, the classification trees for both types of algorithms are shown in Figure 2.5 below.

2.4.2 Monte Carlo and Temporal Difference Methods

The Monte Carlo algorithm is a sampling-based learning algorithm and is based only on past experience. When using Monte Carlo in machine learning, the main approach is to average the reward values for each state-action pair from different segments. Algorithms that fall into this category are the Policy Gradient algorithms, such as PPO and TRPO.

Temporal Difference combines the ideas of dynamic programming and Monte Carlo methods. Similar to dynamic programming, Temporal Difference uses bootstrapping in the estimation process and, like Monte Carlo, does not require complete information about the environment in the learning process. Temporal Difference uses difference values, i.e. the difference between the target value and the estimated value at different time steps, for learning. It uses bootstrapping because it has to construct its goal from the observed payoffs and the evaluation of the next state. Temporal Difference is mainly represented by algorithms such as DDPG, Q-Learning and DQN, which belong to the Actor-Critic class.

In summary, the classification trees of the two classes of algorithms are shown in Figure 2.6.

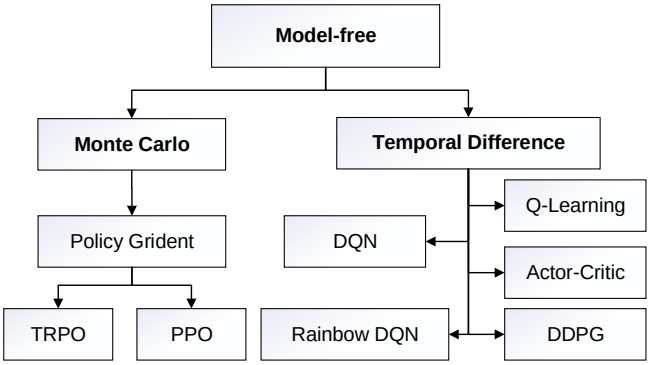

Figure 2.6: Monte Carlo and Temporal Difference RL Algorithms

2.4.3 On-policy and Off-policy Methods

The on-policy approach and the off-policy approach classify reinforcement learning algorithms based on how policies are learned, as shown in Figure 2.7. The on-policy approach attempts to evaluate and improve the policy that interacts with the environment to generate the data, while the off-policy approach evaluates and improves a different policy than the one that generates the data. This means that the on-policy approach requires that the policy with which the intelligence interacts with the environment and the policy that is to be improved must be identical. Off-policy methods, on the other hand, do not have to observe this restriction and can use the data obtained from the interaction of other intelligences with the environment to improve their own policy. A common approach for on-policies is for the PPO to select and execute an action based on the current policy, and then update the current policy based on the data returned by the environment. The PPO thus interacts with the environment and updates the same policy. Q-learning and DQN, on the other hand, are typical off-policy methods.

In summary, when the three categories of classification methods are combined, the following classification tree emerges in Figure 2.7.

2.5 PPO

The Proximal Policy Optimisation (PPO) algorithm has the same motivation as Trust Region Policy Optimisation. It solves the problem of how to make the maximum possible improvement steps to a policy with the data currently available without accidentally causing a performance crash and deviating too far. The PPO approach is much easier to implement and seems to work empirically at least as well as TRPO. Currently, there are two main variants of PPO: PPO with Adaptive KL Penalty and PPO with Clipped Objective. The PPO-Penalty approximation solves the TRPO-like KL-forced update, but penalises the KL-scatter in the objective function instead of making it hard, and automatically adjusts it during training. PPO-Clip has no KL dispersion term in the objective function and no constraints at all. Instead, it relies on ad hoc adjustment of the

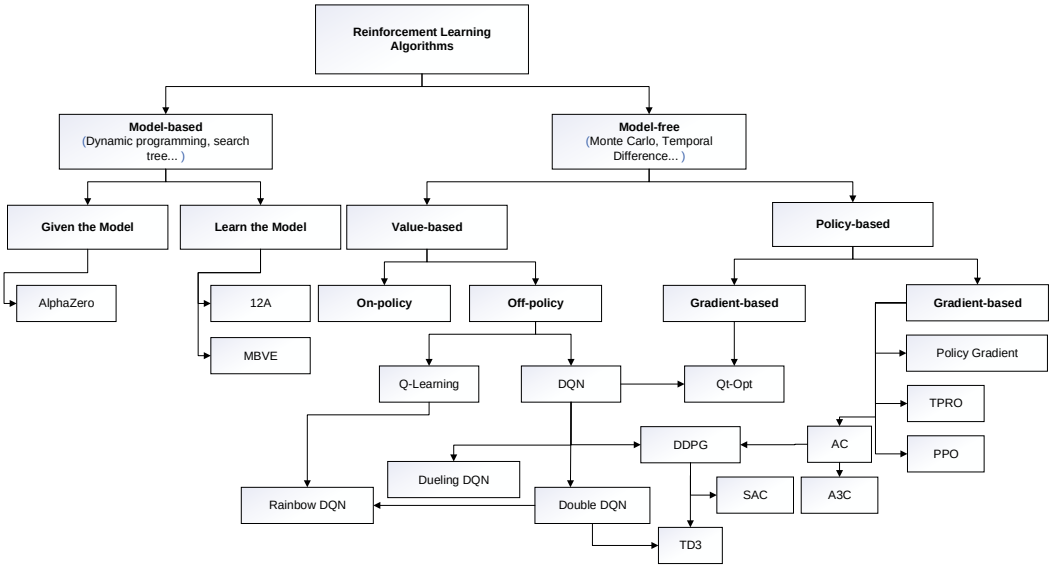

Figure 2.7: Kinds of RL Algorithms

objective function to remove the incentive for new policies to move away from old ones. This article focuses only on PPO with Clipped Objective as this is the main variant used by OpenAI.

2.6 DQN and Rainbow DQN

Deep Q-Networks (DQN) was published by Mnih's team in 2015. The algorithm combines Q-Learning and Deep Learning to tackle instability through two key techniques, Replay Buffers and Target Networks, and has made significant progress on Atari games. DQN addresses the approximation of learning state action value functions and the problem of convergence instability. Since then, many evolutionary algorithms for DQN have emerged.

For example:

- Prioritised Experience Replay (PER) weighted sampling of data based on TD errors to improve learning efficiency;
- Dueling DQN improves network structure by decomposing action value function Q into state value function V and dominance function A to improve function approximation;
- Double DQN uses various network parameters to select and evaluate actions to solve the overestimation problem;
- Rainbow includes three other DQN extensions, including Multi-Step Learning, Noise Network and Value Distribution Reinforcement Learning, in addition to Double Q-Learning, Dueling structure and PER.

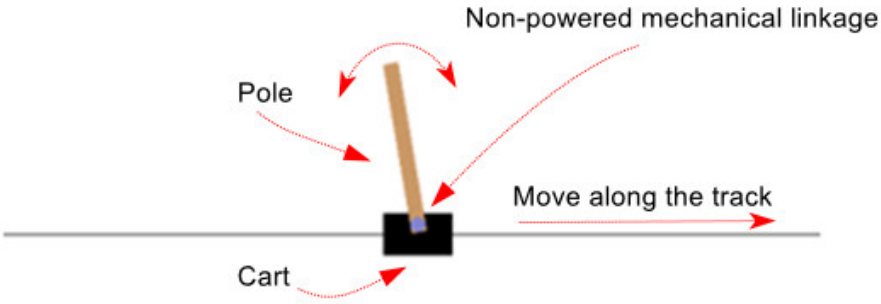

Figure 2.8: The `CartPole` Environment Scenario

2.7 Benchmark

2.7.1 OpenAI Gym

OpenAI is designed to address the lack of standardisation in papers while creating better benchmarks by providing a variety of easy-to-set-up environments. A number of benchmark scenarios are provided, such as the commonly used `LunarLander-v2`, `CartPole-v0`, `Mountain Car`, `CartPole-v1`, etc.

2.7.2 CartPole-v0

`CartPole` is primarily a reference environment for testing algorithms. The scenarios are the same in the different versions. The difference lies in the threshold values of the parameters, e.g. the threshold values for the parameters `max_episode_steps` and `forward_threshold` are different in the two versions. For example, the thresholds for the `max_episode_steps` and `forward_threshold` parameters are different in the two versions.

The `CartPole` Problem, as shown in Figure 2.8 , is, in short, the problem of keeping a pole upright by moving a cart. The pole is connected to the cart by a driveless joint and the cart moves along a frictionless path. The pendulum starts upright and the goal is to prevent it from tipping over by increasing and decreasing the speed of the cart.

2.8 PyTorch, TensorFlow and Related Work

This paragraph briefly summarises the important release nodes of TensorFlow and PyTorch in their respective developments. TensorFlow is the result of further simplification and restructuring based on Disbelief, The first version was published in 2015 under

the Apache 2.0 licence and the first-generation machine learning system developed by the Google Brain team in 2011. Version 1.0.0 of TensorFlow was released in February 2017, marking the stable version. In early 2019, Google released version 2.0 of TensorFlow with a much smaller CPU footprint, but with significantly lower computational speed due to the use of dynamic graphs instead of static graphs in version 2.0. In contrast, PyTorch was born later than TensorFlow. In 2017, a Torch-based version of PyTorch was launched by FaceBook. In March 2018, Caffe2 was integrated with PyTorch and in May, PyTorch version 1.0 was officially released.

TensorFlow and PyTorch have attracted a lot of scientific attention and research. The number of users of both frameworks is increasing and they are becoming the most widely used machine learning frameworks. The question of which framework is better and when you should choose one to train your algorithms is becoming a hot topic in data science. 2.0 still does not offer the same ease of use compared to PyTorch. PyTorch offers more complex dataflow processing but has the advantage of allowing you to define custom options, while PyTorch is faster to train and run than TensorFlow [3]. The simplicity of learning is perhaps one of the main reasons for the increase in PyTorch users, and execution speed is also an important reference factor for training algorithms. In addition, Scott attributes the usability and deployment improvements driven by the PyTorch community to the broadening of the range of applications [2]. Data scientists such as Horace He have analysed the current state of TensorFlow and PyTorch in research and industry and have concluded that TensorFlow has a clear advantage in the industrial domain, but PyTorch has a clear advantage in all five top conference journals with usage rates above 50[14]. Mobile apps have a huge user base and are one of the most important endpoints in the industry, which is perhaps one of the reasons why the industry is adopting TensorFlow. Simmons' team conducted an artificial neural network (ANN) analysis to compare the performance and usability of the two frameworks. Both quantitative and qualitative analyses of TensorFlow and PyTorch were compared [12] and conclusions were drawn describing the strengths and weaknesses of the two frameworks. In short, the above researchers compared the frameworks in terms of usability, community support, user roles, features of the frameworks themselves, scalability, training speed and performance.

Currently, there are no publications on the topic of when to choose TensorFlow and PyTorch for training reinforcement learning algorithms. Therefore, this article compares TensorFlow and PyTorch in terms of performance and usability using PPO and RainBow DQN algorithms for reinforcement learning. Performance tests were conducted simultaneously on Windows 10, Ubuntu 20.04, and Cirrus systems, with Additional GPU-accelerated performance tests and tests with different CPU core counts are conducted on Cirrus to account for users' varying hardware resources. Ultimately, a number of recommendations are made on when to choose TensorFlow or PyTorch based on both types of reinforcement learning algorithms to reduce the time required to select a framework and increase training efficiency.

Chapter 3

Methodology

3.1 RL Baseline Algorithms Selection

3.1.1 Changes to PPO

The implementation of the PyTorch version of the PPO algorithm is largely based on the original PPO code. The TensorFlow version of the PPO algorithm, on the other hand, is based on PyTorch code and other people's DQN code. I had to write my own heavily modified TensorFlow code because there was no existing code that I could have modified.

No changes were made to the theoretical logic of the PPO algorithm itself. The changes and additions were made to make the code more testable. The specific changes are, first, changing the parameters from fixed to configurable to make it easier to pass parameters on the command line; second, recording the performance parameters and necessary configuration information, mainly to evaluate the convergence of the algorithm and analyse performance; third, updating the protocols and model paths to make the whole project clearer; fourth, adding a network layer for the NN. The main reason for adding a layer is to strengthen the differences appropriately, considering that algorithm implementations generally use 2 to 4 layers of networks and the original code implementation had too few 2-layer networks, which might make it difficult to show the differences between PyTorch and TensorFlow.

The GPU usage simply uses the original functionality of PyTorch and TensorFlow, without any special design. More specifically, the PyTorch version of the `cuda` code uses the GPU and copies the tensor from CPU to the GPU. TensorFlow, on the other hand, uses the GPU by default. If it does not need to use the GPU, but only the CPU, it needs to set the value of `os.environ["CUDA_VISIBLE_DEVICES"]` to -1.

3.1.2 Changes to Rainbow DQN

The changes to the Rainbow DQN in the code are mainly for two reasons. First, section 2.6 described how the Rainbow DQN was optimised from the DQN, so the code has been optimised in the same way. Furthermore, compared to the PPO algorithm, Rainbow DQN does not add another network layer to the NN, apart from changing the way parameters are configured, recording performance and configuration parameters and updating logs, which is mainly due to the optimisation of the network structure on top of the DQN itself and the extension of the noise network. The reason for this is that the DQN had difficulty converging during the debugging hyperparameters. Considering the short time available for the project, the Rainbow DQN was chosen as the derivative instead of the classical DQN algorithm so that the algorithm could converge. Finally, the added GPU code is the same as the PPO code and is not repeated here.

3.2 RL Testing Benchmark Selection

The benchmark used in this paper is the `CartPole-v0` test reference scenario in OpenAI Gym. Currently, there are many environments that support reinforcement learning training, and the goodness and performance of algorithms need to be evaluated in such environments. Examples include DeepMind Lab, Google Research Football, OpenSpiel and OpenAI Gym. The main reasons for choosing OpenAI Gym as the benchmarking agent platform for this project are its compatibility with the Windows 10 operating system, its compatibility with the TensorFlow and PyTorch frameworks, its relative ease of installation and referencing in code, its classic test scenarios (e.g. cart pole and mountain car) and the availability of more resources on the internet.

The simulated scenarios used for training in this work are `MontainCar-v0` and `CartPole-v0`, which were learned from the classical controller in OpenAI Gym. When `MontainCar-v0` was used for training, both the PPO and DQN algorithms did not converge significantly at first and it was difficult to see that the algorithm gave an optimal or even stable result. When we replaced the `MontainCar-v0` scenario with `CartPole-v0`, the results improved significantly and the training time to reach the optimal solution became shorter. The reason for the different results of the two simulated scenarios is that the goal of the `MontainCar` scenario is to reach the highest point (see Figure 3.1), while the goal of the `CartPole` scenario is to reach equilibrium (see Figure 3.2). Intuitively, the highest point is one point, while the equilibrium point can be in several places, making the equilibrium point easier to reach. Theoretically, `MontainCar` is a low-reward environment where it is difficult for the agent to obtain positive rewards during exploration, leading to a problem of slow or even impossible learning. This problem is common in the real world, e.g. Go, where it is difficult to determine the reward for each intermediate move and the state space is so large that using global rewards would introduce the problem of sparse and delayed rewards. The `CartPole` scenario, on the other hand, is a mainline reward whose task is to be

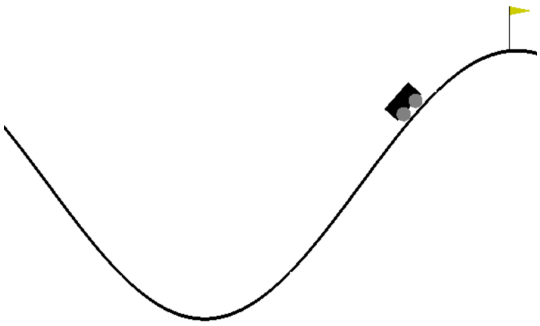

Figure 3.1: MontainCar[7]

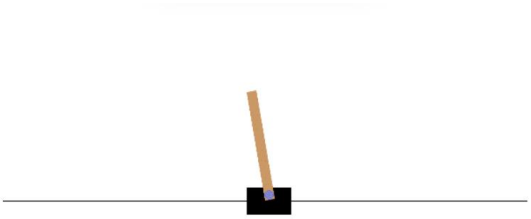

Figure 3.2: CartPole[6]

balanced in one area.

Apart from testing the `MontainCar-v0` and `CartPole-v0` benchmark scenarios, no other benchmark scenarios were selected for testing in this project. The main reason for this is that with the testing of `MontainCar-v0` and `CartPole-v0` the project had already found a benchmark scenario with which the project could be completed and the use of further benchmark scenarios would have meant additional time spent on tuning. In addition, the use of further scenarios would have had the aim of checking the quality of the optimisation of the algorithm and seeing whether it is more robust in different benchmark scenarios. However, the topic of this project is mainly testing the performance differences between TensorFlow and PyTorch and not the optimisation of the algorithm.

3.3 Metrics for Performance Comparison

3.3.1 Return

This project uses the rate of return as a sign of the convergence of the algorithm. This is because the goal of intelligence can be reduced to maximising the cumulative sum of scalar reward signals received by the algorithm, also known as return on investment. The algorithm's test environment for this work was set up with a `max_step` value of 100, meaning that for each sequence of episodes, 100 action steps are taken. Since each of these environments receives the same `max_step` value for all test environments of the algorithm, the target value for the maximum return is also the same. As the number of episodes increases, the return should show an upward trend and eventually converge at a certain level and along a certain trajectory, which is the metric against which reinforcement learning is usually compared.

Accuracy is not used here as a marker for assessing convergence. Since reinforcement learning deals with Markov decision process problems (MDPs), especially in deep reinforcement learning, it is generally a POMDP problem. MDPs are a mathematically

idealised form of the reinforcement learning problem for which exact theoretical statements can be made, and the returns are one of the key elements [1]. In reinforcement learning, the values that can be approximated are state or action state values, but the same action state may have different values depending on the strategy at that time, so accuracy cannot be used to define convergence at that time, and return is generally used to account for training.

Loss is also not used as a marker to assess convergence. Although we call it a loss function, it is not a loss function in the typical sense of supervised learning. One of the most important differences with the standard loss function is that it does not measure performance. Here we are only concerned with expected return [13].

3.3.2 Training Time

When training the algorithm, the programme stores the start time of each step. On the one hand, it is possible to record the return over time, which helps to compare TensorFlow with PyTorch in terms of temporal convergence, i.e. better learning performance, based on CPU or GPU acceleration.

3.3.3 Memory Usage and GPU Usage

Memory usage is determined using the `psutil.virtual_memory()`.percent method. It is worth noting that the percentage of virtual memory determined here includes the sum of main memory (RAM) and a logical memory block virtualised using disc space, which is uniformly referred to as memory consumption in this article. However, the memory and GPU consumption is influenced by the parameters of the algorithm itself, e.g. the maximum `batch_size` allowed for training, since the memory consumption is equal to the sum of the model's memory consumption and the result of the memory consumption per sample multiplied by the `batch_size`.

3.3.4 CPU Usage

The usage of CPU is determined via `psutil.cpu_percent()`. By comparing the usage of CPU over time. It is useful to compare TensorFlow and PyTorch to see which programme uses more computational resources. However, it should be noted that a CPU utilisation value of 100% is misleading and does not mean that as many CPU cores as available are taking up all the running CPU cores. In the same scenario, if you increase the number of CPU cores by a moderate amount, the algorithm will take up more CPU cores while the CPU utilisation decreases.

3.4 Operating Environments Selection

3.4.1 Operating Systems

Choosing different operating systems to test the performance of reinforcement learning algorithms in TensorFlow and PyTorch is necessary because the roles of the users training the algorithms may be different. For example, they may be hobbyists with only a laptop running Windows, or students with a computer running Ubuntu or Windows, or engineers or researchers with a corporate or school server. Therefore, this article will cover the mainstream operating systems Windows 10, Ubuntu 20.04 and EPCC's Cirrus systems as operating system test platforms.

In this test, the online platform Google CoLab, a product developed by the Google Research team, was not preferred. The advantages are mainly the immediate availability of a hosted `Jupyter` notebook service and the availability of free GPUs. However, there are a number of reasons why we do not prioritise testing in CoLab further in this article for now. For one, CoLab can be a problem with too many folders and timeouts, which can disrupt the testing process. Also, CoLab can fail in the middle of a test due to memory problems. Also, the way it works is different from a traditional operating system and there is less experience and community resources, which means you may need a long time to learn how to use CoLab before you can use it. It is also important to note that CoLab is essentially a virtual machine. When it interacts with local discs to read and write data, the IO footprint is large, affecting performance test results. However, using a remote computing platform is certainly a good alternative to operating systems when the user's computer does not have the hardware to train its own algorithms.

3.4.2 Virtual Workstation

Ubuntu 20.04 and Windows 10 systems are installed as virtual machines in a virtual workstation. There are four main reasons for this. First, it is easy to adjust the number of processors and the memory size of the system. You can change the number of processors and the size of the memory directly in the Workbench settings after the virtual machine has been hung or shut down. Secondly, it is possible to set up the environment in a separate virtual environment, avoiding the tedious and time-consuming installation and uninstallation of the operating system. Thirdly, suppose the system fails for operational reasons. In this case, you can save time by quickly installing a new virtual machine at any time or directly restoring the state of the specified time node through snapshots using ISO image system files. Another important reason is that Python's psutils is a cross-platform library for retrieving information about running processes and system load [11]. This library detects data generated by all processes on the system, not by the algorithm programme itself. Therefore, to minimise the impact of other processes on the performance metrics, Virtual Workstation installs a relatively clean

Figure 3.3: Topology of Operating Systems for Testing

environment without other applications to reduce the error in the measurement results.

The most common virtual workstations are VMWare Workstation and VM VirtualBox, a stable and mature virtualisation platform developed by VMWare in 1999. free managed virtual machine hypervisor. The two platforms have been compared by many researchers. The Deepak team evaluated the performance of both platforms, comparing the read, write and rewrite performance of various virtual machines installed on them and discussing the performance differences between the two platforms themselves, with VMWare performing better overall [15]. It is also worth noting that when trying to use the Ubuntu virtual machine in VirtualBox for this project, the problem of not being able to display the GUI due to insufficient disc space was common and difficult to solve. For these reasons, VMWare Workstation was chosen. As shown in Figure 3.3, the topology of the three operating systems used in this work is shown below.

3.4.3 Environment Manager

The required environments such as Python, PyTorch and TensorFlow must be created before the algorithms can be tested on the already selected operating system. There are three easy-to-implement and common options: The first is to create the required test environment directly in a virtual machine, the second is to create the test environment in Anaconda3, and the third is to create the test environment in Miniconda3. Installing the environment directly on the system is prone to problems such as lack of compatible dependencies and lack of flexibility in changing the environment during installation. In addition, Anaconda3 is a heavyweight environment management system containing more than 180 scientific packages such as `conda` and Python and their dependencies. In contrast, Miniconda3 is a lightweight environment management system that contains only `conda`, Python and pip. At the same time, Anaconda3 and Miniconda3 can create multiple virtual `conda` environments and easily switch between environments for different types of tests. Finally, for this report, the solution chosen was to use Miniconda3 to create the test environment to reduce disc overhead and make it easier to switch between test environments.

3.5 Experimental Design

3.5.1 Hardware Resources

As shown in Table 3.1, Table 3.2, Table 3.3, Table 3.4, and Table 3.5, the hardware information is for Windows with 2 CPU cores, Ubuntu with 2 CPU cores, Cirrus with 2 CPU cores, Cirrus with 40 CPU cores and 0 GPU and Cirrus with 40 CPU cores and 1/2/3/4 GPU(s). Furthermore, the hardware information of Cirrus with 4, 8, 16 and 32 CPU cores is the same as that of Cirrus with 2 CPU cores. Windows systems can be viewed with the command DOS `systeminfo`, Ubuntu systems with `free -m` and `lscpu` for memory and CPU information respectively, and Cirrus can be viewed in the Cirrus documentation [5].

Types	**CPU node Information**
OS Name	Microsoft windows 10 Pro
OS Version	10.0.19044 N/A Build 19044
System Type	X64-based PC
Model Name	Intel(R) Core(TM) i7-10875H CPU @ 2.30GHz
Processor(s)	2
CPU cores	2
GPUs	0
Physical Memory	8191MB
Virtual Memory	8412MB
Memory Usage Error	22%
CPU Usage Error	1.5%

Table 3.1: Hardware Information of Windows 10

Types	**CPU node Information**
OS Name	Microsoft windows 10 Pro
OS Version	10.0.19044 N/A Build 19044
System Type	X64-based PC
Model Name	Intel(R) Core(TM) i7-10875H CPU @ 2.30GHz
Processor(s)	2
CPU cores	2
GPUs	0
Physical Memory	8191MB
Virtual Memory	2047MB
Memory Usage Error	13.69%
CPU Usage Error	1.5%

Table 3.2: Hardware Information of Ubuntu 20.04

Types	CPU node Information
OS Name	Red Hat Enterprise Linux release 8.4 (Ootpa)
OS Version	8.4
System Type	core-4.1-amd64
Model Name	Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
CPU cores	2
GPUs	0
Physical Memory	256GB
Memory Usage Error	0%
CPU Usage Error	0%

Table 3.3: Hardware Information of Cirrus with 2 CPU Cores

Types	CPU node Information
OS Name	Red Hat Enterprise Linux release 8.4 (Ootpa)
OS Version	8.4
System Type	core-4.1-amd64
Model Name	Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
CPU cores	32 (which is 16*2 in two nodes)
GPUs	0
Physical Memory	512GB
Memory Usage Error	0%
CPU Usage Error	0%

Table 3.4: Hardware Information of Cirrus with 32 CPU Cores in Two Nodes

3.5.2 Controls

Controlled variable

In all the experiments in this article, the main method was to control variables. The controlled variables are the types of operating system, the types of algorithms, the numbers of CPU or the numbers of GPU. Therefore, four types of experiments were divided into two separate tests, the first with different types of operating system as shown in Table 3.6, the second with different numbers of CPU as shown in Table 3.7, the third with different algorithms as shown in Table 3.8, and the fourth with different numbers of GPU as shown in Table 3.9. It should be noted that in the experiments RAM is considered sufficient and no specific values are set as constants.

Software Version

The packages versions used in the virtual test environment are the same as shown in Table 3.10, except for the hardware as a variable. This is to have the same benchmarks so that it makes sense to compare the results.

Types	CPU node Information
OS Name	Red Hat Enterprise Linux release 8.4 (Ootpa)
OS Version	8.4
System Type	core-4.1-amd64
Model Name	Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz
CPU cores	40
GPUs	1/2/3/4
Physical Memory	768GB (=384*2)
Memory Usage Error	0%
CPU Usage Error	0%
nvidia/nvhpc	22.2
gcc	8.2.0
nvidia	NVIDIA Corporation GV100GL [Tesla V100 SXM2 16GB] (rev a1)

Table 3.5: Hardware Information of Cirrus with 40 CPU Cores and 1/2/3/4 GPU(s)

OS	GPU	CPU Cores	Memory Available
win10	0	2	8GB
ubuntu	0	2	8GB
Cirrus	0	2	256GB

Table 3.6: Test Parameters under Different OS

The two frameworks selected for this article are based on the principle of minimal changes to the source code. The version of PyTorch was selected based on the GitHub algorithm source code and uses version 1.4. TensorFlow version 1.15.0 was used instead of TensorFlow version 2.x. The main reasons for this were firstly that the performance of 1.x was better than the performance of 2.x, which was generally considered to be significantly slower in terms of computational speed. Secondly, it makes sense to use TensorFlow 1.x because the open source algorithms for reinforcement learning are essentially implemented in TensorFlow 1.x and not TensorFlow 2.x. Third, TensorFlow 2.x's more complex code syntax and longer write cycle make it unsuitable for the near-term completion of this project, and it does not fit the project's focus on performance testing rather than testing programming skills.

3.5.3 Automation of Testing Scripts

Use automated scripts to simplify the steps of creating the environment, training and drawing. It is time-consuming and error-prone to execute all commands manually. The project uses automation scripts in three ways: First, the steps to create the `conda` environment and install the toolkit can be configured with pip via a `YAML` file that can be run with a specific command line to create the virtual `conda` environment. Secondly, the `SLURM` file can be configured for submitting jobs in Cirrus, which not only reduces

OS	GPU	CPU cores	Memory Available
Cirrus	0	4	256GB
Cirrus	0	8	256GB
Cirrus	0	16	256GB
Cirrus	0	32	256GB

Table 3.7: Test Parameters under Different Numbers of CPU

OS	Algorithms	CPU cores	Memory Available
Cirrus	TensorFlow PPO	32	256GB
Cirrus	TensorFlow Rainbow	32	256GB
Cirrus	PyTorch PPO	32	256GB
Cirrus	PyTorch Rainbow	32	256GB

Table 3.8: Test Parameters under Different Types of Algorithms

the number of steps to be performed, but more importantly eliminates the need to log in to the job node. Secondly, the `SLURM` file for submitting a job in Cirrus not only reduces the number of steps to be executed, but more importantly, it also reduces the need to log in to the job node.

Using `SLURM` scripts in the Cirrus server to execute code via CPU nodes and GPU nodes requires, among other things, attention to the priority of the command tags. First, the `--exclusive` command tag is exclusive to the memory on the CPU node, and if this command tag is not used, the system allocates memory based on resources. If a CPU tag is not specified at the same time, it means that a node's resources are exclusive to both CPU and memory. And for the GPU level, `--exclusive` means that both CPU and memory are exclusive. However, if the numbers of GPU are not specified, it also means the exclusive use of all GPUs of the node in addition to CPU cores and the memory. Also, it is important to specify the command tag `--time` wisely, as the numbers of CPU hours per account are limited and specifying too long a time for convenience not only wastes resources but also requires frequent requests for CPU hours.

Therefore, nine `SLURM` scripts must be configured in Cirrus. This is equivalent to test 2, 4, 8, 16 and 32 CPU cores scenarios and to test 1 to 4 GPU(s) scenarios each. The details of the test scripts are described in 4.3.2.

3.5.4 Evaluation of the data

The output of the data results is divided into three main areas: Output of the protocols, Output of the results and further graphical representation of the results.

OS	GPU	CPU cores	Memory Available
Cirrus	1	40	384GB
Cirrus	2	40	384GB
Cirrus	3	40	384GB
Cirrus	4	40	384GB

Table 3.9: Test parameters under Different Numbers of GPU

Packages	Versions
gym	0.19.0
matplotlib	3.3.4
numpy	1.19.2
pandas	1.1.5
pip	21.2.2
psutil	5.8.0
pynvm	11.4.1
tensorboard	1.15.0
tensorboardX	1.6
tensorflow	1.15.0
torch	1.4.0
cuda	11.6

Table 3.10: The Packages Versions in the Virtual Test Environment

Logs

The logs are mainly the real-time training logs, which are written to a file, and the logs, which are written to a `SLURM-out` file. The former is used to record training times and network errors using the Tensorboard component to observe training in real time or offline, mainly for efficient tuning purposes. However, in this work, using Tensorboard to monitor convergence in real time during testing is not recommended because a large percentage of hardware resources are consumed during the detection process, which in turn affects the performance metric values. The latter are mainly used for debugging and monitoring progress.

Results

The output of results consists of two phases. The first phase is the output of the performance data, and the second phase draws a graph based on this data. The results of the training include episode values, returns, training time, virtual memory usage, CPU, GPU usage and device status, which are stored serially with Pickle and used to form the conclusions of the report. In addition, the character chart is based on the data in the Pickle file. The independent and dependent variables for the charts and the types of charts are mainly created with reference to all the metrics mentioned in section 3.3.

3.6 Usability

The usability assessment is mainly done from the user's point of view. This report examines almost all articles and discussions in Google Search Engine that involve a comparison between TensorFIow and PyTorch and summarises and categorises user perspectives when evaluating the two programmes. In this report, both are discussed in terms of learning difficulties, data loading, device management, visualisation tools, graph computations, distributed execution and training, deployment, serialisation, compatibility, and scalability. It is important to emphasise that firstly, learning difficulties are discussed from the perspective of programming style, syntax and programming ideas, backwards compatibility, community, and ecology; secondly, deployment comparison is mainly discussed from the perspective of stability, deployment methods and types of supported end devices.

Chapter 4

Experimental Works

4.1 Implementation of Code

4.1.1 PPO and Rainbow DQN

The changes to the PPO code are more of a supporting code to facilitate testing. The changes to the Rainbow DQN are similar and are described in detail in section 3.1 and are not repeated here.

4.1.2 Output

Logs of Output

TensorFlow uses Tensorboard via a built-in function, while PyTorch's log files are mainly generated via the tensorboardX tool. Specifically, TensorFlow collects data via the `tf.summary` module and stores it in a log using `tf.summary.FileWriter`. PyTorch uses the tensorboardX tool to create a `tensorboardX.writer` and then stores it in a log via the `tensorboardX.add_scalar` method to load the data to be stored in the log. For example, to access the log directory in the command line, type `tensorboard --logdir=logs-0` in the command line to access `http://localhost:6006/` in a search engine and use the Tensorboard tool on this website. This is shown in Figure 4.1.

Results of Output

The results of the test are obtained by creating a list of records and appending the results of each measurement as an element to the records. Finally, the configuration of the test and the data sets are saved in a pickle file.

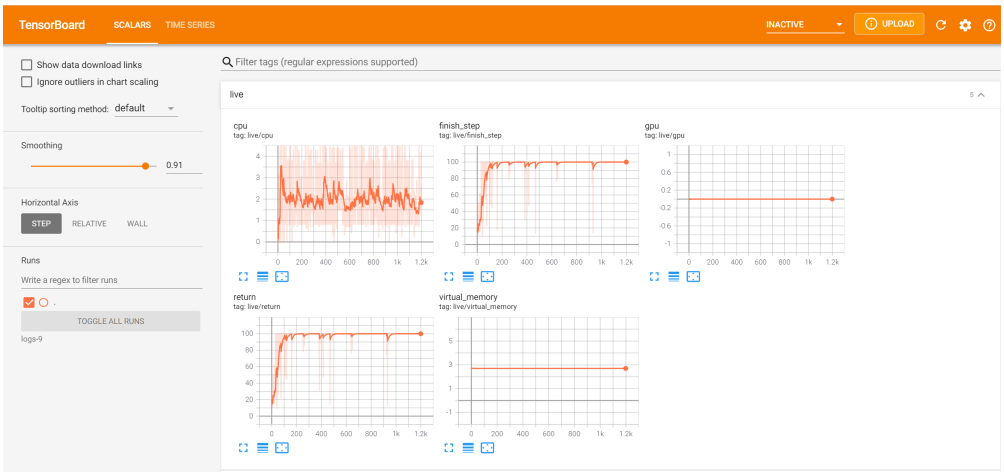

Figure 4.1: Tensorboard Visualisation Tool

Plots of Output

The code for drawing plots is created by reading the pickle file of the result set. The main reference for the plot is the type described in section 3.3. When you run the `plot.py` script, the plot is saved in the specified directory.

4.2 Test Steps

The test steps consist of creating the operating system environment, creating the Miniconda3 virtual environment on the system and installing the Python environment needed for the tests, tuning the algorithm, testing the algorithm, and drawing the plot with the test results.

4.2.1 Build Operating Systems

Windows 10 and Ubuntu 20.04 are virtual machines created on a virtual platform. First, the VMWare Workstation virtual platform was downloaded from the official website and installed on a hard disc partition with sufficient space, as two virtual machines were to be installed later. Then download the ISO image files for Windows 10 and Ubuntu 20.04 from the official website and install them on a hard disc partition with sufficient space. Now configure the number of processors, CPU cores, RAM, and hard disc space according to the hardware resources of your computer. For this project, the settings are 2 processors, 1 CPU core per processor, 8GB of RAM and 100G of disk space. When installing the virtual machine, we recommend that you do not tick any boxes that require personalised recommendations and unnecessary protocol shares, etc. that may cause additional consumption of hardware resources. Once the virtual machine is installed, also download a Portable Git, configure the GitLab repository and clone

your own code into the virtual machine. Finally, before testing, uninstall unneeded applications and disable unneeded services as much as possible to reduce the risk of test failures due to their resource consumption.

It is not necessary to create a special operating system for Cirrus. It just needs to edit the required hardware resources via the command and the system will allocate the resources automatically.

4.2.2 Build Environment Manager

There are two ways to create a Miniconda3 virtual environment: one is manually and the other is by copying an existing environment. When you create a Miniconda3 virtual environment for the first time, you must create it manually at each step of the command. However, if the same test environment needs to be copied, it is possible to use the `YAML` file to copy the virtual environment to different operating systems. It is worth noting that the virtual environment for CPU tests and the virtual environment for GPU tests of TensorFlow and PyTorch3 are different and need to be created separately, as the former is the CPU-only version and the latter is the `cuda` version. In addition, the `YAML` file used to create the environment cannot replicate all Python working environments and must be supplemented with a `requirements.txt` file to install the packages needed for the tests. The specific test scripts and commands are described in section 4.3.

4.2.3 Hyperparameters Tuning

The goal of tuning the hyperparameters is to make the algorithm converge, the ideal goal of convergence being the achievement of a globally optimal solution. The optimal solution is divided into a global optimum and a local optimum, with the global optimum being the most desirable goal state. To make the algorithm converge, this project focuses on the parameters `env_id`, `gamma`, learning rate `lr`, `batch_size`, `num_episodes`, `max_step`, `capacity`, `hidden_dim` to tune.

`env_id` is the name of the reference environment used for training. Originally `MountainCar-v0` was used for training, but this benchmark reference environment was difficult to debug for DQN training, so it was replaced by `CartPole-v0`. Different algorithms behave differently in different benchmark reference environments. In order to perform subsequent performance tests, it is necessary to find a benchmark reference environment in which the target algorithm can easily converge.

`gamma` is the discount factor, i.e. the weighting of the reward after the t-step. Generally, `gamma` is chosen between 0.88-0.99. The larger `gamma`, the greater the difference between the values of the different states, resulting in a learning divergence. `lr` is the learning rate, which controls the learning speed of the model. `lr` is chosen between 0.01-0.0001, the larger `lr` is, the more likely it is that the best point will be missed and scatter will occur.

batch_size is the number of samples per calculation, i.e. the amount of data used in each update when you use the optimiser. The larger the number of samples per training session, the greater the load on memory and calculations and the more accurate the results. Off-policy methods such as DQN (Rainbow DQN), which have an experience pool, can be trained by drawing a sample from the experience pool once per action, i.e. the larger the value, the better the network is likely to fit. On-policy methods (PPO), on the other hand, do not take experience samples, but the training is repeated several times for each number of experience samples specified in the batch_size. So a larger value means a longer training interval and slower training, but due to the complex structure of their network (two networks at the same time), the number of samples must be large enough to match the training of a complex network, otherwise the network scatters, i.e. it becomes too large, leading to more training cycles to achieve the same computational effect, and poor generalisation[8], while a smaller value leads to more noise.

num_episodes is the number of training rounds. Too small a number will cause the computation to stop before the algorithm can converge, too large a number will cause the algorithm to converge while still computing. However, to ensure that the algorithm converges, the parameter is not set too small and its value is only decreased if the number of convergence steps is known.

max_step is the maximum number of steps with which the environment is executed. This parameter influences the probability of exploration learning. Too small a value will result in less exploration and no convergence, too large a value will result in too long a training time but also no convergence due to other unsuitable parameters.

hidden_dim is the number of nodes in the hidden layer of the network. hidden_dim is related to the complexity of the network. The more complex the task, the more complex the network must be to achieve sufficient but non-redundant cognitive performance. A smaller number will not achieve the targeted cognitive performance and a larger number will result in slower training.

capacity is another parameter of DQN and refers to the size of the experience pool, also called storage capacity, i.e. the maximum capacity of the buffer for replaying experiments. The more samples a policy has, the more accurate the fit to the value function of that policy will be. This also means that we want to keep as many samples as possible for training. However, since reinforcement learning policies change during learning, too old samples are actually irrelevant to the current policy - they are noise - and too many noisy training samples reduce the quality of the training. So a suitable compromise has to be found. This is usually done through experience and several attempts to find a suitable value.

In addition, replace_target_iter is a parameter that DQN also has for updating the target network interval. If the target network is updated too quickly, the algorithm lapses into overfitting. The purpose of updating is similar to adding noise to the data so that the convergence is approximately in the right direction and not constantly oscillating.

The above parameters affect the performance of the algorithm in terms of `batch_size`, `hidden_dim` and `capacity`. Therefore, it is important that these three parameters are consistent when you set the parameters in different frameworks for the same algorithm. The other parameters are parameters of the algorithm itself for the learning situation, such as the learning rate `lr` and the discount factor (`gamma`), which are mainly parameters for whether the algorithm is globally optimal and do not affect the computing power of the framework itself.

4.2.4 Running Test Scripts

Once the operating system is set up, the test environment is set up and the algorithm has been tuned with hyperparameters until the algorithm converges, it is time to run the test scripts for testing. The scripts for testing in Windows 10, Ubuntu 20.04 and Cirrus and the commands in the test scripts are described in detail in section 4.3.

4.3 Automation of Testing Scripts

The automation of scripts mainly includes the replication of the `conda` virtual environment, the test commands in Cirrus and the test commands in the Windows 10 and Ubuntu 20.04 virtual machines.

4.3.1 `YAML` and Requirements File

The steps to create a `conda` environment and a Python environment are simplified by creating a `YAML` file and a `requirements.txt` file respectively. If you are creating a `conda` environment and a Python environment for the first time, you will need to create them manually. Second, after creating the first environment, you can use the export environment command to create the `YAML` file of the `conda` environment and export the `requirements.txt` file of the Python environment. Third, if the same new environment needs to be created, the `YAML` file can be used to allocate the environment by first creating the `conda` environment with the `YAML` file command, then activating the `conda` environment, and then specifying the `requirements.txt` file with the pip command to add the toolkit needed for testing to the Python environment. Details of all the above commands can be found in the GitLab repository[9].

4.3.2 `SLURM` and Shell Scripts of Running Job

Using the `srun` command and `SLURM` scripts are two ways to run your own code on the Cirrus server using CPU nodes and GPU nodes. The `srun` command specifies the resources required and requires a login to the computer node on which these resources

are configured for execution. It is necessary to change to the working directory `/work/` and run your own code. The advantage is that the process is very close to the steps you would take on your own computer and is simple and easy to follow. The disadvantage is that logging into the node consumes memory unnecessarily. Alternatively, `SLURM` scripts are generally considered the better method for testing. The advantage is that you do not need to log in to the test node and simply execute the test code in the node. The commands in the scripts consist of four main parts: first, configuring the parameters of the test node; second, activating the Miniconda environment on the node; third, executing the test code; and fourth, moving the result files to the specified directory. Submitting the `SLURM` file is as simple as using the `sbatch [SLURM-filename]` command to submit the task. The `srun` command to run the code and the `SLURM` automation script with a GPU test as an example can be found in Appendix B: Automatic Scripts and Commands. More detailed commands can be found in the GitLab repository[9]. Using shell scripts on Windows 10 and Ubuntu 20.04 systems simplifies the procedure and is described in detail in the GitLab repository[9].

4.3.3 Job Chaining

Job chaining mainly configures the execution of `SLURM` files and Python character scripts. When you have completed testing on Windows 10, Ubuntu 20.04 and finally on Cirrus. The work chain can be configured for ten `SLURM` files and one character script. The ten `SLURM` scripts correspond to the nine tests in Cirrus described in section 3.5.3. For details, see in the GitLab repository[9].

Chapter 5

Results and Analysis

The analysis in this chapter focuses on the results from five perspectives. The first is a comparison of training reinforcement learning algorithms in PyTorch and TensorFlow on the same platform; the second is a comparison of different frameworks and different algorithms on different operating systems; and the third is a comparison of the differences in training different classes of reinforcement learning algorithms in the two frameworks on the same operating system with a different number of CPU. The fourth is the difference between the two frameworks for training different classes of reinforcement learning algorithms on the same operating system with a different number of GPUs. In addition, the usability of the two frameworks is analysed.

5.1 Performance analysis

In this section, two frameworks, TensorFlow and PyTorch, are analysed using PPO and Rainbow DQN algorithms, respectively, for a comparative analysis of data metrics, including convergence, convergence speed, CPU and memory consumption.

5.1.1 PyTorch Versus TensorFlow

Return with Episodes

Figure 5.1 and Figure 5.2 show that the PPO algorithm and the Rainbow DQN algorithm converge smoothly in both frameworks and converge with almost the same number of time episodes. This result shows that both algorithms themselves obtain better regularised hyperparameters. It is the benchmark for the analysis of the latter metrics. It is worth noting that compared to the other three curves that reach the global optimum, TensorFlow-PPO converges with a score between 20 and 25 and falls into a local optimum, which happens for two reasons. First, the tendency to fall into local optima is a

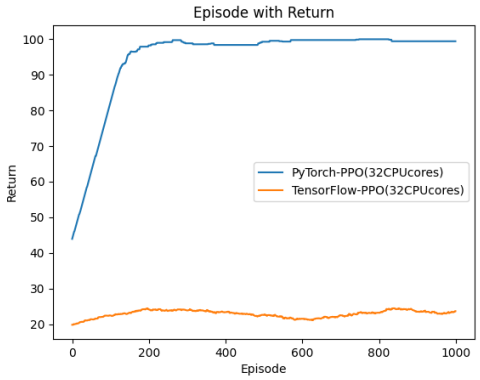

Figure 5.1: Return with Episodes of PPO

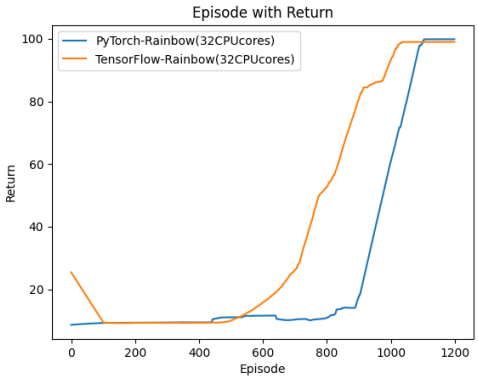

Figure 5.2: Return with Episodes of Rainbow

recognised weakness of the PPO algorithm itself. Second, the differences in the underlying implementation principles between PyTorch and TensorFlow are perhaps another reason for the local optimum. This does not affect the performance comparison, as the algorithm can be considered to be running stably when it reaches a local or global optimum. For example, in the CartPole scenario, the global optimum is to keep the pole vertical and the local optimum is to keep it oscillating in a certain range.

Return with Time

Figures 5.3 and 5.4 show the running times for different cases with the same number of episodes.

Looking first at the two types of algorithms, PPO is clearly faster in computation. The reason for this is that PyTorch first needs to get the results in NumPy, which then need to be exported from NumPy to PyTorch's tensor to run during training, and then exported to NumPy to interact with the environment and count the results when the computation is complete. TensorFlow, on the other hand, can use NumPy to process data and operations directly. In terms of algorithms, Rainbow DQN trains every step it executes, and when the number of strategy optimisation steps increases from 10 to 100, the number of training steps increases from 9 to 99, resulting in a noticeable speed-up, while PPO trains every `batch_size` step it executes, which may not result in a noticeable increase in time due to the smaller number of training steps. This is the main reason for the time difference between the two algorithms. Due to the nature of the Rainbow DQN algorithm, frequent data exchange is required, while PyTorch is computationally more efficient because the tensor calculation is a dynamic graph calculation.

Secondly, training different algorithms for both platforms does not always benefit the two frameworks, but has its own advantages. From the PPO algorithm results, TensorFlow-PPO is about 17.14% faster than PyTorch-PPO. Looking at the Rainbow results, the difference is reversed, with PyTorch-Rainbow being 76.67% faster than TensorFlow-Rainbow. This result may not only be due to the computational principles of the plat-

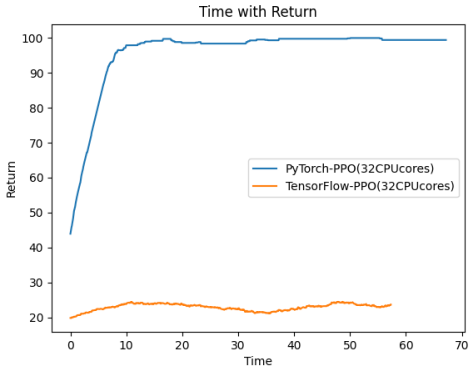

Figure 5.3: Return with Time of PPO

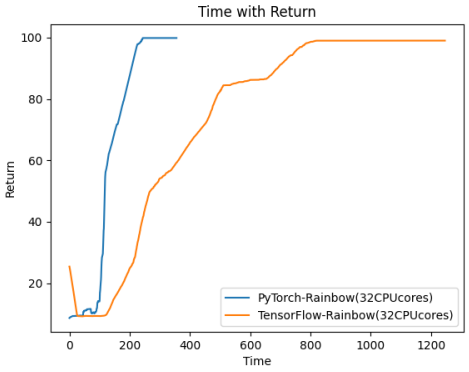

Figure 5.4: Return with Time of Rainbow

form itself, but also a combination of the properties of the algorithms themselves. This is because PyTorch and TensorFlow process data differently and the algorithms themselves are trained and learned differently. For Rainbow DQN, as mentioned in the previous paragraph, frequent data exchange is exactly what PyTorch is good at.

CPU Usage or Memory Usage with Time

Figure 5.5 and Figure 5.6 show that PyTorch consumes even more CPU resources. As mentioned in the previous section, PyTorch performs frequent computations through interactions between NumPy and Tensor, resulting in a large consumption of CPU resources. In contrast, TensorFlow only interacts with NumPy to perform data operations and data storage, and the large number of calls to NumPy to shop and compute data requires the creation of new and more NumPy variables to shop and compute data, resulting in TensorFlow being more memory intensive compared to TensorFlow. The results for Figure 5.7 and Figure 5.8 also confirm this view.

In summary, there is a significant difference in the computational results due to the

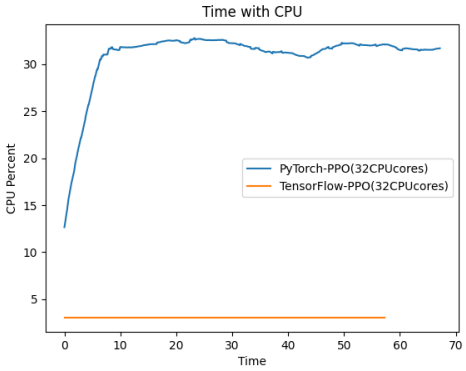

Figure 5.5: CPU with Time of PPO

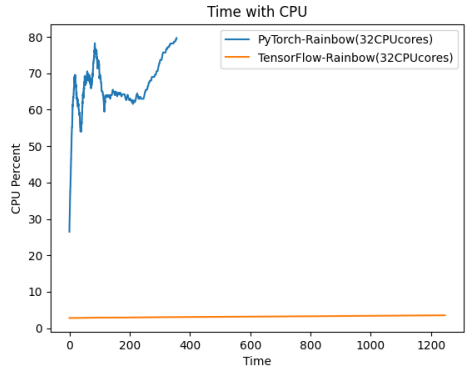

Figure 5.6: CPU with Time of Rainbow

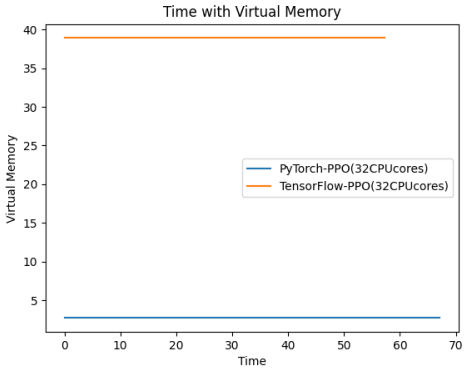

Figure 5.7: Memory with Time of PPO

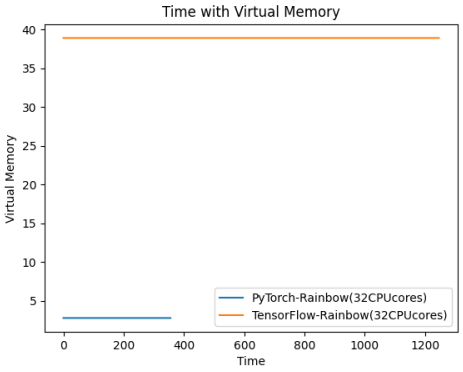

Figure 5.8: Memory with Time of Rainbow

different principles of the training and learning process for different types of algorithms and the different way TensorFlow and PyTorch frameworks handle data. In addition, PyTorch CPU is more memory intensive, while TensorFlow consumes more memory resources. As a result, PyTorch is better suited for scenarios where data interactions are more frequent, while TensorFlow is better suited for environments where the volume of data and computation is higher but data interactions are less frequent.

5.1.2 Cirrus Versus Ubuntu20.04

This summary analyses only the most important metrics of the algorithms running in Cirrus and Ubuntu 20.04. Since the results are almost identical when looking at the performance of the most important metrics in the results of the algorithms running in Ubuntu 20.04 and Windows 10, the comparison is not repeated. The full charts can be found in the corresponding directory in the GitLab repository[9].

Return with Episodes

Figure 5.9, Figure 5.10, Figure 5.11 and Figure 5.12 show that both the PPO algorithm and the Rainbow DQN algorithm achieve global or local optimality on both Cirrus and Ubuntu 20.04 systems, which can serve as a prerequisite for the following comparative analysis.

Return with Time

Figure 5.13, Figure 5.14, Figure 5.15 and Figure 5.16 show that, first, PyTorch does not have much variability in training times for the same class of algorithms on different operating systems for both classes of algorithms. For example, the time curves of the PPO algorithm for both platforms almost overlap, and the Rainbow DQN algorithm

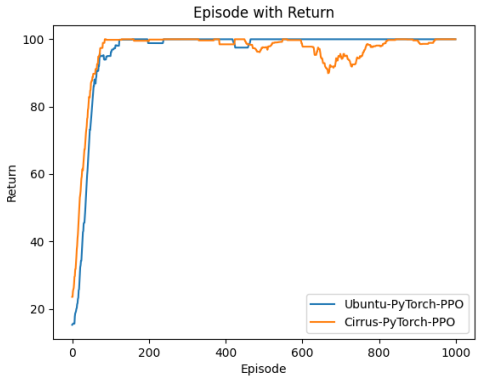

Figure 5.9: Return with Episode of PPO

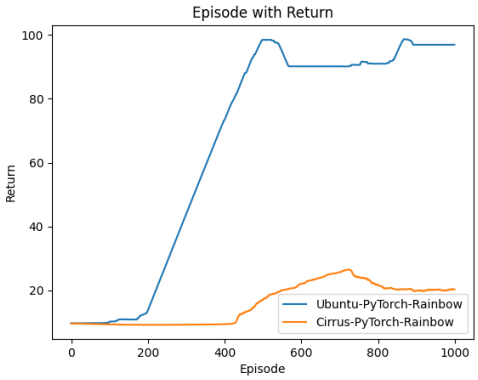

Figure 5.10: Return with Episode of Rainbow

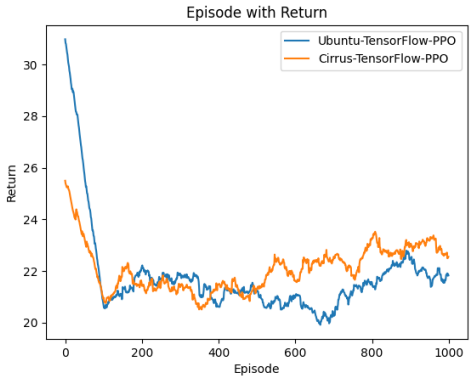

Figure 5.11: Return with Episode of PPO

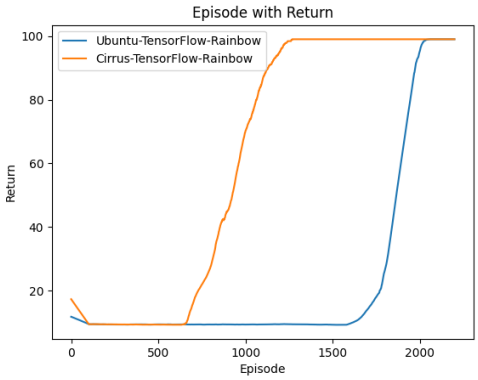

Figure 5.12: Return with Episode of Rainbow

also reaches the optimal solution in about 200 seconds. In a direct comparison, the two algorithms reach a state of convergence that differs in the time dimension by a factor of about 40. The TensorFlow framework, on the other hand, shows considerable differences in performance when the same algorithms are trained on different operating systems. In particular, the PPO algorithm computes 33.33% faster than Cirrus on Ubuntu 20.04. The Rainbow DQN algorithm, on the other hand, was trained with a lead of almost 50% of the time. In direct comparison, the two types of algorithms reach a state of convergence with a difference of about 40 times in the time dimension. Thus, the difference in training time between the PyTorch and Rainbow DQN frameworks for different algorithms is almost the same, i.e. the difference between the two frameworks for different algorithms is almost non-existent between different operating systems. Longitudinal comparisons of the same algorithms between different frameworks were already compared in 5.1.1 and are not repeated here.

It can be concluded that there is no additional difference in training efficiency due to differences in operating system architectures when both PPO and Rainbow DQN algorithms are trained, all other conditions being equal, at least as shown by the results of the tests between Windows 10 and Cirrus' Linux systems.

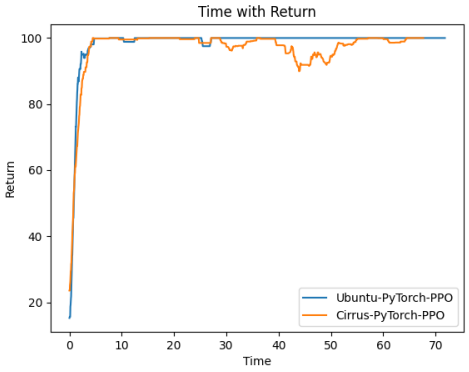

Figure 5.13: Return with Time of PPO

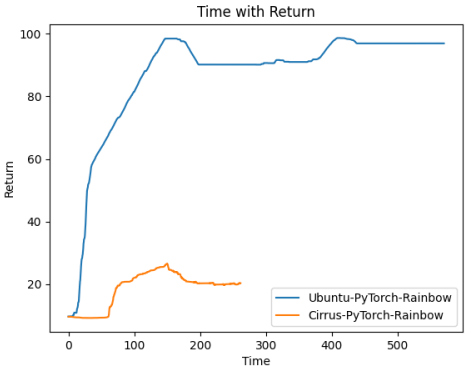

Figure 5.14: Return with Time of Rainbow

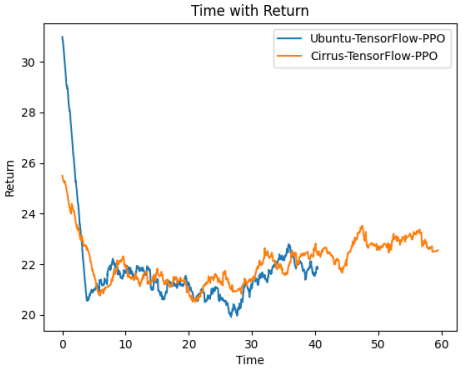

Figure 5.15: Return with Time of PPO

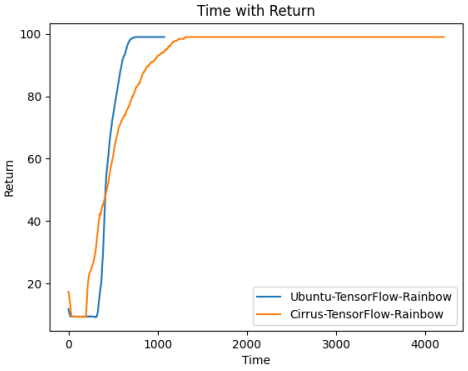

Figure 5.16: Return with Time of Rainbow

CPU Usage with Time

The four figures as shown in Figure 5.17, Figure 5.18, Figure 5.19 and Figure 5.20, that suggest that testing the performance of Cirrus is more beneficial in that one does not need to log in to the node, which reduces the consumption of additional resources due to logins and other enabled services, which in turn leads to inaccurate performance parameters.

However, from another perspective, the difference in CPU consumption between TensorFlow and PyTorch is significant. In a horizontal comparison, the difference in CPU consumption between PyTorch is larger for different algorithms, which is mainly due to the principle of the underlying computation; algorithms that require less data interaction consume significantly less CPU in PyTorch. The same phenomenon is observed in TensorFlow, where the difference between algorithm types is the dominant factor. The vertical comparison shows TensorFlow's lack of dependence on CPU resources, with TensorFlow consuming fewer CPU resources than PyTorch in both Ubuntu 20.04 and Cirrus.

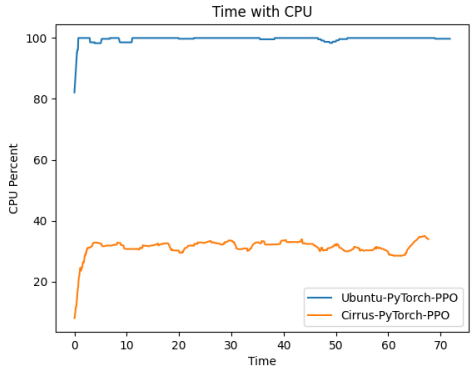

Figure 5.17: CPU with Time of PPO

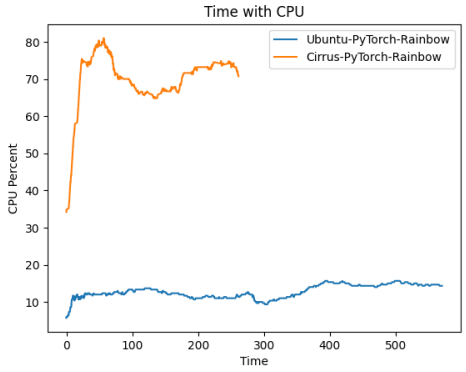

Figure 5.18: CPU with Time of Rainbow

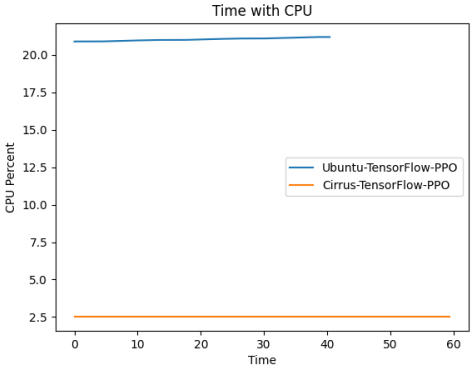

Figure 5.19: CPU with Time of PPO

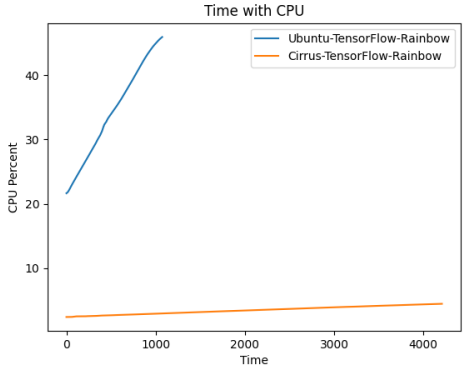

Figure 5.20: CPU with Time of Rainbow

Virtual Memory Usage with Time

As can be seen in all four diagrams Figure 5.21, Figure 5.22, Figure 5.23 and Figure 5.24, the memory consumption is constant. In the vertical comparison, the PPO algorithm, the TensorFlow framework requires more memory resources than PyTorch, and the Rainbow DQN algorithm performs similarly for both frameworks.

5.1.3 Differences with Different CPUs

In this subsection, the integration of important data parameters over time is used to streamline the legend and reduce the number of charts analysed. The idea is to take CPU cores as a horizontal coordinate and integrate each CPU cores dimension against existing data, such as CPU usage or virtual memory usage. The starting point for this is that the results should show some correlation when the number of CPU cores increases by a certain trend.

Indeed, the same conclusions can be drawn for Rainbow as for the PPO analysis. With-

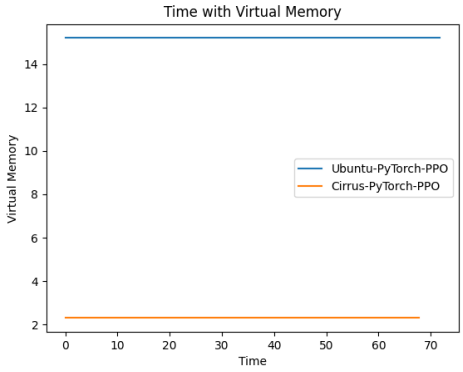

Figure 5.21: Time with Virtual Memory of PyT-PPO

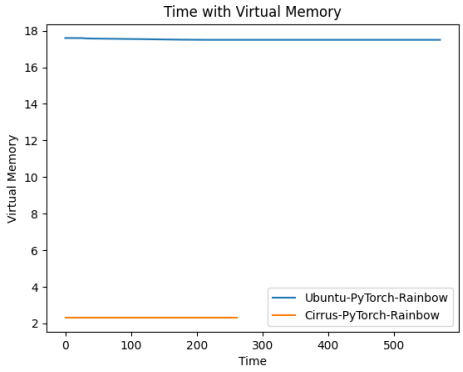

Figure 5.22: Time with Virtual Memory of PyT-Rb

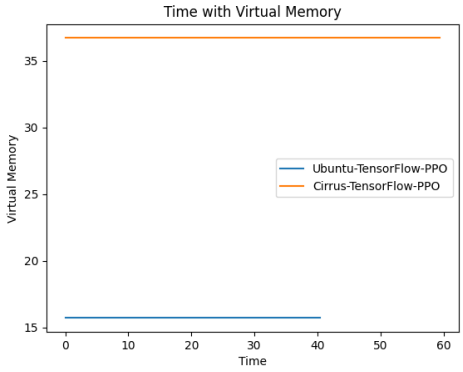

Figure 5.23: Time with Virtual Memory of TF-PPO

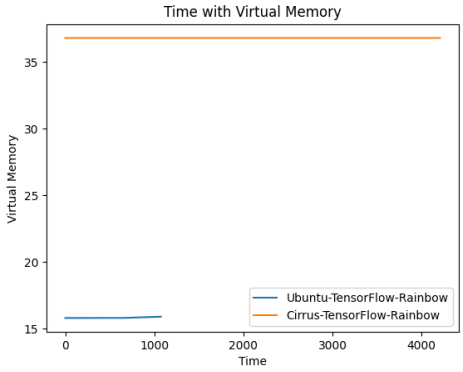

Figure 5.24: Time with Virtual Memory of TF-Rb

out repeating the expression, the results of Rainbow DQN are described in detail in the GitLab repository[9].

Testing PPO on CPU Node

From the first two diagrams, Figure 5.25 and Figure 5.26, it can be seen that regardless of the number of CPU cores, PyTorch always consumes more CPU resources and TensorFlow consumes more memory resources. A look at Figure 5.27 shows that PyTorch performs better because the size of Return is positively correlated with the number of steps executed, which in turn is positively correlated with the actual number of training sessions, but this test controls for the same number of trials to be trained, so in return, the larger the integral Return, the better the framework performs. Figure 5.28 and Figure 5.29 show that PyTorch consumes fewer virtual memory resources and more CPU resources per step for the PPO algorithm. This confirms the results in the Episodes Figure 5.27.

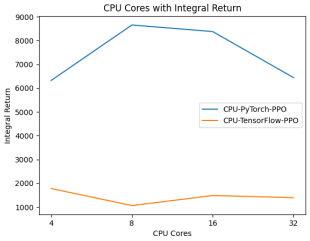

Figure 5.25: CPU Cores with Integral Return

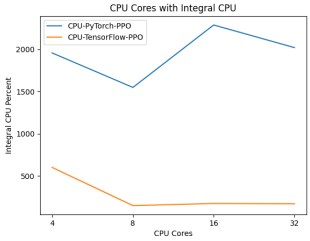

Figure 5.26: CPU Cores with Integral GPU

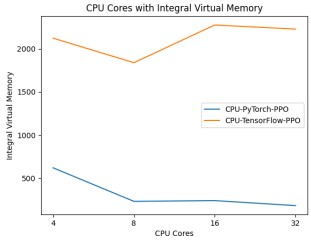

Figure 5.27: CPU Cores with Integral Virtual Memory

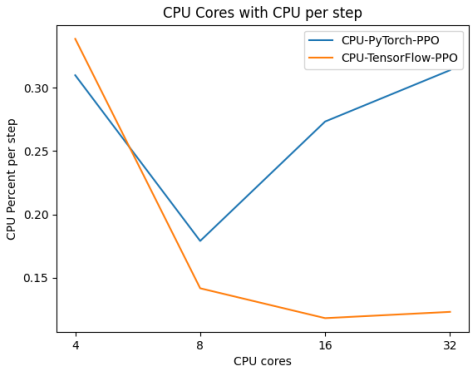

Figure 5.28: CPU Cores with CPU per Step

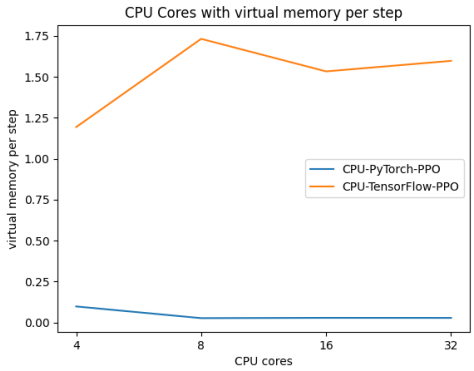

Figure 5.29: CPU Cores with Virtual Memory per Step

5.1.4 Differences with Different GPUs

During GPU acceleration, neither the PPO algorithm nor the Rainbow algorithm changed the conclusions in section 5.1.3. On this basis, a new phenomenon arises from GPU acceleration.

Testing PPO on GPU Node

As you can see from the figures of Figure 5.30, Figure 5.31, Figure 5.32, Figure 5.33 and Figure 5.34, to the increase in GPUs, both frameworks have only a small impact on the CPU consumption and the memory consumption for the computation of the PPO algorithm and no big impact on the convergence of the algorithm. This is probably because the PPO algorithm itself does not have many data interactions and consumes more memory, while the GPUs if there is an additional increase in copying data to the GPU and returning it. Unfortunately, the data interaction time was not tracked in this experiment, but it can be surmised that the data interaction time with the GPU is much greater than the GPU acceleration time. This means that GPU acceleration saves time in the PPO algorithm and more time is spent on data interaction.

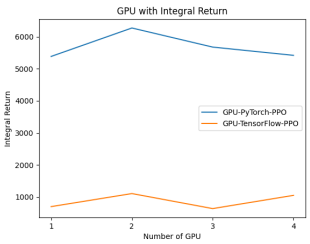

Figure 5.30: CPU Cores with Integral Return

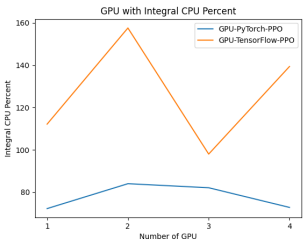

Figure 5.31: CPU Cores with Integral GPU

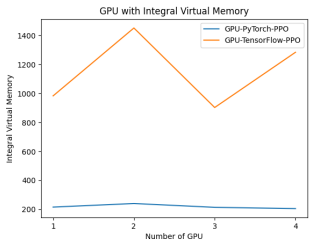

Figure 5.32: CPU Cores with Integral Virtual Memory

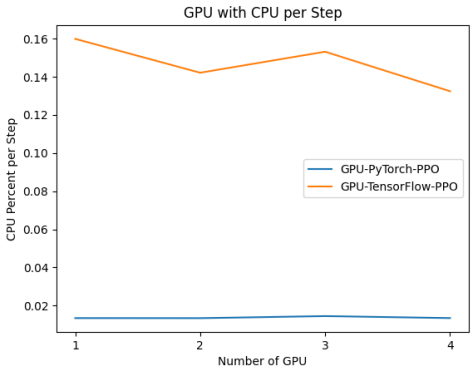

Figure 5.33: CPU Cores with CPU per Step

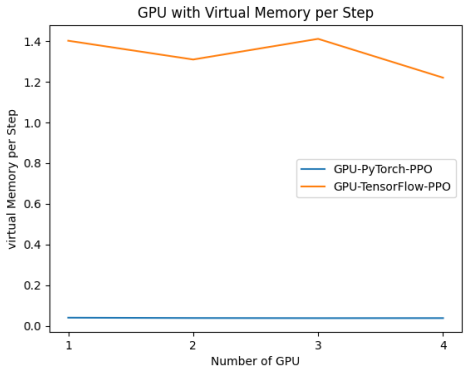

Figure 5.34: CPU Cores with Virtual Memory per Step

Testing Rainbow DQN on GPU Node

Overall, as show in Figures of Figure 5.35, Figure 5.36 and Figure 5.37, TensorFlow always consumes more resources than PyTorch, and the TensorFlow curve tends to decrease in terms of overall consumption, with efficiency increasing with the number of GPUs, while PyTorch does the opposite.

However, given the difference in actual performance, as shown in Figure 5.38 and Figure 5.39, we can see that PyTorch converges better as the number of GPUs increases, while TensorFlow does not.

Therefore, we also need to determine the impact of the different number of actual steps per round on the overall consumption. As we can see from the curve per step, PyTorch shows a decreasing trend, with efficiency increasing with the number of GPUs, while TensorFlow does the opposite. In addition, PyTorch has a significantly higher number of training sessions than TensorFlow for GPU counts greater than or equal to 3, while TensorFlow consumes more than PyTorch both overall and per training session, suggesting that TensorFlow may be more resource intensive when using GPUs.

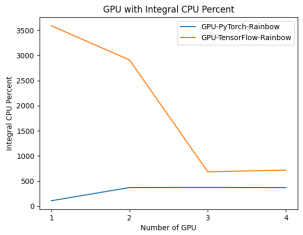

Figure 5.35: GPU with Integral CPU Percent

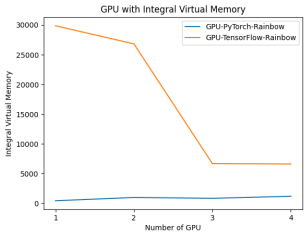

Figure 5.36: GPU with Integral Virtual Memory

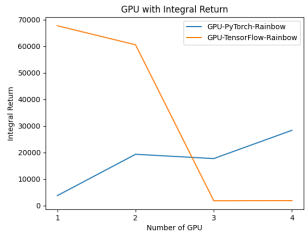

Figure 5.37: GPU with Integral Return

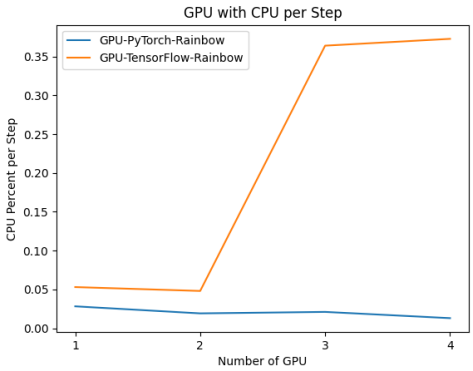

Figure 5.38: GPU with CPU per Step

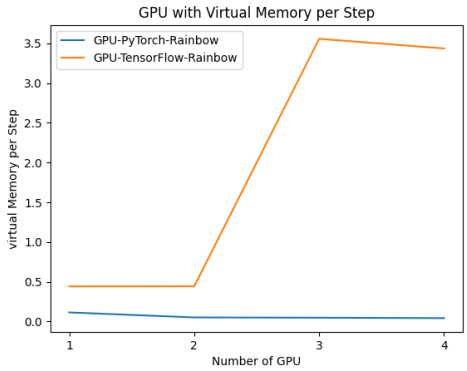

Figure 5.39: GPU with Virtual Memory per Step

5.2 Usability Analysis

5.2.1 Ease of learning

Programming Style

In terms of programming style, PyTorch is easier to learn. PyTorch is Python and Python programmer friendly, and if you are familiar with `NumPy`, PyTorch is easy to learn; perhaps this is why Stanford University, for example, uses PyTorch for teaching Deep Learning. TensorFlow, on the other hand, can be thought of as a programming language embedded in Python. When you write TensorFlow code, it is compiled into a graph by Python and then executed by the TensorFlow execution engine. For this reason, additional learning of placeholders, variable fields, sessions, etc. is also required, which makes learning the language more difficult. In addition, more sample code is required to run the base model. As a result, the pre-learning framework with TensorFlow takes significantly longer than PyTorch.

Syntax and programming ideas

TensorFlow is purely symbolic programming, while PyTorch is imperative programming. TensorFlow's purely symbolic programming is usually executed after the computational process is fully defined, and is therefore more efficient, but more difficult to implement due to its complex syntax. Both the TensorFlow 1.x and TensorFlow 2.x versions suffer from the complexity of the syntax. TensorFlow 2.x can use Eager Execution, but it still requires building low-level components and the complexity of writing code is still high. In contrast, PyTorch has the advantage of being syntactically simple and easily implemented as imperative programming but has the disadvantage of being inefficient in execution as it needs to be compiled during execution.

Backwards Compatibility

TensorFlow 1.x to TensorFlow 2.x has many new features and is like learning a new language, with high costs of reworking old code and poor backwards compatibility. In comparison, PyTorch is better backwards compatible and does not have too many sudden updates.

Community and Ecology

Currently, the community and ecology of the two frameworks are indistinguishable. In the early days, TensorFlow had a stronger community and more resources for experiences and tutorials than PyTorch, which was released later. In addition, both PyTorch and TensorFlow have a rich ecosystem. For example, PyTorch's highlight fast.ai API enables fast model building, has an open-source model server TorchServe and TorchElastic, which uses Kubernetes to train deep neural networks at large scale, etc., while TensorFlow has the Keras Advanced API, the Advanced API model collection Model Garden, Scikit-Learn, and so on.

5.2.2 Loading Data

PyTorch has a much better API for loading data and the PyTorch interface for loading data consists of a dataset, a sampler and a data loader. TensorFlow's approach to loading data is simpler and more straightforward and does not have the same sophisticated data loading mechanism as PyTorch.

5.2.3 Device management

PyTorch must explicitly move all content to the GPU device, even if CUDA is enabled. As a result, writing PyTorch code requires more frequent CUDA checks and more ex-

plicit device management, especially if you are writing code that runs on both CPUs and GPUs.

TensorFlow does not require manual tuning and uses the GPU by default. If you set `CUDA_VISIBLE_DEVICES` to -1, you turn off the GPU and can easily manage it. Memory and video memory can thus be precisely managed and controlled. The disadvantage of TensorFlow device management, however, is that it uses all the memory of all available GPUs by default, even if only one of them is actually in use. So the GPUs always look busy, even if they are actually idle. The solution is to use `tf.device()` to specify the specific GPU device to work on.

5.2.4 Visualisation tools

TensorFlow has a better visualisation tool, Tensorboard, and visualising data and listening to data while training a model can make a significant difference in training efficiency and is key to understanding the performance and work of the model. Tensorboard is the most used tool in TensorFlow and has many features. There is a similar TensorboardX in PyTorch with almost the same functionality as Tensorboard, but TensorboardX is not as efficient at loading tensor data as Visdom, a dedicated visualisation tool for PyTorch. Visdom, however, has the disadvantage of emphasising the concept of an ambient environment. Visdom also currently supports results for cross-environment comparisons but is complicated to set up. Also, saving the Visdom log requires manual saving, and if the results are not saved and the visualisation page is closed, the results data disappears. Both can be used with the standard Python plotting tool `matplotlib`, but it is not suitable for debugging scenarios and is more suitable for discussing the results.

5.2.5 Graph Calculations

Computational graphs are an abstract way of describing computations as directed graphs. Graph computing in both frameworks works with a tensor, and the creation and execution of computation graphs is the biggest difference between the two frameworks. Graph computing can be resource-efficient and computationally intensive. It lends itself well to distributed computing, and computational work can be spread across multiple GPUs, CPUs, processes, threads or multiple devices. And many machine learning models inherently lend themselves to computations organised in a graph format.

Currently, the stable version TensorFlow 1.x uses static graphs, the less stable version TensorFlow 2.x uses dynamic graphs, and PyTorch uses dynamic graphs. Static graphs mean that the graph is compiled and then executed. The structure of the constructed graph cannot be changed, except for numbers, which are less flexible. A dynamic graph means that a new graph can be reconstructed during the computation.

For the initialisation process, the way the two frameworks define the graph is different. TensorFlow for static graphs needs to assign placeholders to the computed graph

dimensions and variables of the tensor and then run the session to compute the graph. Although TensorFlow 2.x has the Eager Execution feature for dynamic graphs, whose operations are executed immediately when called by Python, the performance is not good enough as execution slows down when the convolution is computed. PyTorch, on the other hand, with dynamic building block graphs, does not have to worry about the size tensor at initialisation and can define the size in variables, define nodes, change them and execute them at any time during the run.

TensorFlow is not easy to debug, either by checking variables before running a session or by learning the TensorFlow debugger (`tfdbg`), which is very complex. In contrast, PyTorch is easier to debug. You can use `pdb` and set breakpoints everywhere.

5.2.6 Parallel execution and training

PyTorch supports distributed execution. The code implementation of PyTorch's distributed execution approach is relatively simple, and PyTorch supports distributed execution with the wrapper module `torch.nn.DataParallel`. TensorFlow is much better at distributed training. Although PyTorch 1.8 started supporting distributed training, it has taken longer for more people to use it. Early versions of TensorFlow, on the other hand, supported massive amounts of distributed training and were much better at distributed computing on the GPU, which is much more efficient for Big Data.

5.2.7 Deployment

Stability

TensorFlow is an early version and has good stability and is used in industrial applications. PyTorch, on the other hand, is a latecomer and is less used in industrial applications. Although the algorithms have good performance and more and more companies are moving to the PyTorch architecture, there is currently no guarantee of stability.

Deployment Methods

In TensorFlow, models can be deployed directly into TensorFlow using TensorFlow Serving, which is suitable for large-scale deployments. However, in PyTorch, production deployment does not provide a framework for deploying models directly to the web. You need to use Flask or Django as a backend server. There are also many changes from TensorFlow 1.x to TensorFlow 2.x. Related to deployment is the release of `Tensorflow.js`. With the increasing dominance of web applications, the need to deploy models in the browser has increased dramatically. TensorFlow uses `Tensorflow.js` to allow users to run Python models and re-train them in the browser using Node, while TensorFlow uses Javascript to create and train models without requiring Python.

Supported Terminals

Both TensorFlow and PyTorch simplify model storage and loading, and both support small server-side implementations. However, for mobile (Android and IOS) and embedded implementations, TensorFlow is more efficient. There is support for mobile development with APIs for JavaScript and Swift, and then there is TensorFlow Lite, which allows compression and optimisation of models for IoT devices for mobile devices. Even PyTorch 1.8 is starting to support mobile devices. PyTorch does not support embedded implementations, but the version that does is too new and there are fewer practical validations of stability. Alternatively, in TensorFlow 2.x there is TensorFlow Lite, a lightweight library for deploying models to mobile and embedded devices. With TensorFlow Lite, you can easily convert an existing model into a "compressed flat buffer" and then load it onto a mobile or other embedded device. The main optimisation process is to convert 32-bit floating point values to 8-bit, which is more suitable for embedded devices (less memory required).

Similar to TensorFlow Lite, PyTorch improves on the existing PyTorch Mobile, a framework for quantifying, tracking, optimising and storing models for Android and iOS. A prototype PyTorch Lite interpreter that reduces the size of binary runtimes on mobile devices has also been released.

5.2.8 Serialization

Both are relatively easy to buy and load, but TensorFlow has the advantage of being more cross-language. If you compare the two frameworks in serialisation, TensorFlow's graph can be stored as a log buffer that contains operations and parameters. TensorFlow's graphs can then be easily loaded into other programming languages such as Java and C++. This is particularly important for deployment environments that do not support Python and can be applied in scenarios where older models need to be run. PyTorch, on the other hand, has a simple API that can march in entire classes if needed and also buy in all the property weights of the model.

5.2.9 Compatibility and Scalability

Firstly, TensorFlow supports more languages than PyTorch. TensorFlow is compatible with several languages such as C++, JavaScript, Python, Go, C#, Ruby and Swift as well as R. PyTorch supports Python and C++. If you do not want to write too much low-level code with TensorFlow, you can use Keras instead, which has a high-level API that can significantly shorten the experimental cycle for iteratively building and training algorithm models.

In addition, TensorFlow itself supports a wider range of features and more library features. For example, TensorFlow supports the fast Fourier transform, NaN and infinity

checking of a tensor, the `tf.sequence_mask` used in filling calculations, the inverse hyperbolic tangent function `tf.math.atanh`, and the ability to flip a tensor along any dimension. These functions are currently not supported by PyTorch and fewer functions are implemented.

5.3 Summary

5.3.1 Performance

The performance analysis is divided into several dimensions, namely different operating systems (Ubuntu 20.04, Windows 10), different types of algorithms (PPO, Rainbow DQN) and different computational frameworks (TensorFlow, PyTorch).

With the same hardware configuration and the same runtime environment set up for the different operating systems, the training of the algorithms themselves did not differ due to the different operating systems and also not additionally due to the different frameworks.

With different types of reinforcement learning algorithms and different types of frameworks, the main reason for the differences in performance and convergence between the algorithms is primarily the different training and learning methods. Furthermore, TensorFlow and PyTorch differ in the way they perform their underlying data operations to either increase or decrease the difference, i.e. PyTorch is more efficient mainly due to the advantage of frequent data interactions and dynamic graph tensor computations, while TensorFlow is more memory-intensive as it mainly computes in memory. This means that the performance of algorithms with different functions in both frameworks is reflected in the algorithm functions. In addition, TensorFlow consumes more hardware resources when it uses GPU acceleration.

5.3.2 Usability

PyTorch has many advantages, e.g. it is easier to learn and is suitable for users who need to get into the real world quickly; the data loading mechanism is more perfect, data processing is more convenient and is suitable for dealing with more complex data sources; it has flexible dynamic graph computation, which is suitable for machine vision, RNN, natural language processing and other scenarios that require variable long stack size; it is easy to debug, flexible and efficient; However, TensorFlow's parallel execution method also has disadvantages, such as cumbersome device management, which increases the complexity of code generation; data visualisation is not perfect, but can be compensated by using TenesorboardX; parallel training is not mature enough and training speed is slow, which is not suitable for fast iterative scenarios; online deployment is less practical and stability is yet to be tested in practise.

Similarly, TensorFlow has many advantages, such as easy device management, which facilitates strict management of RAM and video memory; it has a better visualisation tool, Tensorboard, which makes training algorithms more efficient; static graph computation has better parallelism and excellent performance for optimising specific devices and algorithms, but debugging is difficult; parallel training is fast and effectively shortens the training cycle. Parallel execution also performs well and is suitable for industrial scenarios with large amounts of data and high parallelism. Deployment supports hot deployment and can ensure deployment without shutting down the server and supports multiple programming languages.

Chapter 6

Conclusions

In summary, different operating system perspectives, whether for the two frameworks of PyTorch and TensorFlow or for the algorithms themselves, make virtually no difference in terms of performance; different algorithm perspectives, different algorithms make big differences in computational time efficiency and convergence; Different framework perspectives, PyTorch and TensorFlow due to the way they process data Different framework perspectives, PyTorch will consume more CPU resources while TensorFlow will consume more memory resources, affecting the computational performance of algorithms with different characteristics, amplifying or mitigating the differences. Furthermore, GPU acceleration does not affect PyTorch's performance, but adding GPUs has a greater impact on TensorFlow, which can consume more hardware resources. Secondly, PyTorch has better backward compatibility, is suitable for fast training of algorithms, has simpler syntax for easy debugging, dynamic graph computation is more suitable for scenarios that require dynamic tensor computation, but also has drawbacks, less stable online cases, does not support multiple languages and is only suitable for lightweight use; TensorFlow has more stable performance, is suitable for parallel TensorFlow has more stable performance, is suitable for parallel training, has better visualisation tools and can design algorithms in advance to improve performance in scenarios where static graph computation is used as an advantage.

Bibliography

[1] Alex M Andrew. Reinforcement learning: An introduction by richard s. sutton and andrew g. barto, adaptive computation and machine learning series, mit press (bradford book), cambridge, mass., 1998, xviii + 322 pp, isbn 0-262-19398-1, (hardback, £31.95). *Robotica*, 17(2):229–235, 1999.

[2] Scott Carey. Why enterprises are turning from tensorflow to pytorch. *InfoWorld.com*, 2020.

[3] Mihai Cristian Chirodea, Ovidiu Constantin Novac, Cornelia Mihaela Novac, Nicu Bizon, Mihai Oproescu, and Cornelia Emilia Gordan. Comparison of tensorflow and pytorch in convolutional neural network - based applications. In *2021 13th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–6. IEEE, 2021.

[4] DeepMind. Deepmind artificial intelligence alphago. [EB/OL]. `https://www.deepmind.com/research/highlighted-research/alphago` Accessed August 15, 2022.

[5] EPCC. Cirrus user guide. [EB/OL]. `https://cirrus.readthedocs.io/en/main/` Accessed August 1, 2022.

[6] OpenAI Gym. Cart pole. [EB/OL]. `https://www.gymlibrary.ml/environments/classic_control/cart_pole/` Accessed August 16, 2022.

[7] OpenAI Gym. Mountain car. [EB/OL]. `https://www.gymlibrary.ml/environments/classic_control/mountain_car/` Accessed August 16, 2022.

[8] J. Hou, L. Hua, J. Hu, C. Zhao, and P. Quan. A review of the applications and hotspots of reinforcement learning. In *IEEE International Conference on Unmanned Systems*, 2017.

[9] Mingyang Liu. Gitlab repository of dissertation. [EB/OL]. `https://git.ecdf.ed.ac.uk/msc-21-22/s2149684/-/tree/master/` Accessed August 1, 2022.

[10] Sven Mika Michael Galarnyk. Deepmind artificial intelligence alphago. [EB/OL]. `https://www.anyscale.com/blog/an-introduction-to-rei`

nforcement-learning-with-openai-gym-rllib-and-google Accessed August 10, 2022.

[11] Paperswithcode. Paper implementations grouped by framework. [EB/OL]. https://paperswithcode.com/trends Accessed August 16, 2022.

[12] C. Simmons and M. A. Holliday. A comparison of two popular machine learning frameworks. In *33rd Annual Consortium for Computing Sciences in Colleges Southeastern Conference*, 2019.

[13] spinningup comuunity. lntro to policy optimization. [EB/OL]. https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html Accessed August 1, 2022.

[14] tensorflow community. Tensorflow lite for mobile & edge. [EB/OL]. https://www.tensorflow.org/lite/guide Accessed August 1, 2022.

[15] M. S. Vasudevan., B. R. Mohan, and D. K. Damodaran. Performance measuring and comparison of virtualbox and vmware.

[16] Zhi-Hua Zhou. *Machine learning / Zhi-Hua Zhou.* Springer, Gateway East, Singapore, 2021.