Reinforcement Learning for Physics Simulations: A Comparison of A2C, PPO, and TRPO

UNIVERSITY OF
LINCOLN

Eddie Muschamp
MUS19694636

19694636@lincoln.ac.uk

School of Computer Science

College of Science

University of Lincoln

Submitted in partial fulfilment of the requirements for the
Degree of BSc(Hons) Computer Science

Supervisor: Mr Philip Carlisle

May 2023

Acknowledgements

I would like to express my gratitude and appreciation towards several individuals who have played a significant role in the completion of this project. I would like to thank my friends and my partner, whose support and encouragement has been incredible throughout this year. Their support has played a crucial role in maintaining my mood and well-being. Their companionship, empathy, and understanding created a supportive environment that helped alleviate the stress and challenges that often arose during this project.

Abstract

This dissertation presents a comprehensive analysis of three popular reinforcement learning (RL) algorithms, Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C), and Trust Region Policy Optimization (TRPO), in the Acrobot environment. The primary objective is to evaluate and compare the performance and efficiency of these algorithms through testing them in the Acrobot environment.

The research demonstrates that all three RL algorithms, when equipped with carefully tuned hyperparameters, exhibit improved performance compared to their default configurations in terms of task completion in the Acrobot environment. Notably, the A2C algorithm emerges as the top performer, displaying higher reward scores and more consistent scores compared to PPO and TRPO.

The findings highlight the significance of hyperparameter tuning in RL algorithms, emphasizing its role in enhancing learning capabilities and overall performance. Moreover, the study provides valuable insights into the strengths and limitations of each algorithm, which gives valuable results for researchers seeking the most suitable algorithm for similar environments or tasks.

In summary, this dissertation offers a comprehensive comparative analysis of RL algorithms in the Acrobot environment, highlighting the differences how the algorithms are trained as well as how they perform after training. The benefits of hyperparameter tuning for improved performance is also investigated as shows clearly the uses of it. These insights and findings serve as valuable references for other researchers interested in leveraging RL algorithms for similar tasks or applications.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Reinforcement Learning

Reinforcement learning (RL) has gained significant notoriety recently due it being able to allow agents to learn from experience and make decisions based on rewards and punishments. RL algorithms are being successfully used in various areas, including robotics, games, and finance. In robotics specifically, RL has been shown to be effective in learning control policies for both simple and complex systems. Traditional control methods in robotics usually rely on human made rules or algorithms to control the system's behaviour which are designed by a human based on their understanding of the system's dynamics and the task requirements (Kober et al., 2013).

However, these methods may not always the most effective or efficient due to several reasons. Firstly, the system may be nonlinear and difficult to model accurately, therefore making it challenging to manually design a control policy that can handle all possible situations. Secondly, traditional methods may find it difficult to predict uncertain and dynamic environments**.**

RL, on the other hand, learns control policies by directly interacting with the environment, meaning it does not rely on a model of the system but instead learns from experience. This makes RL more adaptable and robust to changes in the environment and system dynamics.

However, there is no one RL algorithm that can solve all problems effectively and efficiently. Therefore, it is crucial to compare and evaluate

different RL algorithms to understand their strengths and weaknesses so that the most appropriate algorithm can be selected for a specific task and environment.

1.2 Acrobot Environment

One of the benchmarks for testing RL algorithms in robotics is the Acrobot environment, which simulates a two-link arm that must swing up and reach past a certain horizontal line above the arm. The Acrobot environment is a challenging problem due to its complex dynamics and high-dimensional state space. By comparing the performance of different RL algorithms in this environment, a better understanding of their capabilities and limitations can be gained for learning control policies in challenging and high-dimensional systems.

1.3 RL Algorithms

In this study, three algorithms will be compared: Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C), and Trust Region Policy Optimization (TRPO). It is useful to compare these algorithms because they represent three popular and successful RL algorithms with many similarities but also key differences which may give one of them a clear advantage or disadvantage in the Acrobot environment.

PPO is a model-free reinforcement learning algorithm that optimizes policies by iteratively improving them through multiple training iterations. It is also known for its stability, sample efficiency, and ability to generalize well to different environments. PPO has demonstrated strong performance in various tasks, making it a popular choice in the field of reinforcement learning.

A2C is another popular model-free reinforcement learning algorithm that has been widely used in various domains. It is an actor-critic algorithm that combines elements of both value-based methods and policy-based methods. It operates by maintaining two separate networks: an actor network (policy) and a critic network (value function). These networks work collaboratively to learn an optimal policy while estimating the value of state-action pairs.

Lastly, TRPO an algorithm that is designed to find an optimal policy for a given reinforcement learning problem. It was introduced by Schulman et al. in 2015 to improve the stability and performance of the traditional policy gradient algorithm.

TRPO, PPO, and A2C are model-free RL algorithms, which means that they do not rely on a model of the environment to learn an optimal policy. Instead, they directly interact with the environment and learn from experience by optimizing a reward function. This contrasts with model-based RL algorithms, which use a model of the environment to learn the optimal policy.

This allows us to evaluate their performance in learning control policies for a specific environment without the need for prior knowledge of the system dynamics. This is particularly important for real-world robotic systems that are often nonlinear and difficult to model accurately. Model-free algorithms are more adaptable and robust to changes in the environment and system dynamics, making them a popular choice in RL applications, including robotics (Kober et al., 2013).

By comparing these three algorithms, insights can be gained into their respective strengths and weaknesses in the context of the Acrobot environment. For example, it may be found that one algorithm is more is able achieve better scores, but another may be more consistent. These

insights can help us better understand how to select the most appropriate algorithm for different types of problems and applications.

1.4 Hyperparameters

Altering the hyperparameters is one way to make the algorithms perform better and more efficiently in an environment. Hyperparameters are settings that are not learned from the data but must be specified by the user, such as the learning rate, batch size, and clip range. Hyperparameters can significantly affect the learning speed and stability an algorithm. Therefore, it is important to carefully tune the hyperparameters the algorithms and environment to achieve optimal performance.

All hyperparameters will be kept the same for each algorithm to ensure that any differences in performance are due to the differences in the algorithms themselves rather than how they are tuned. By controlling for hyperparameters, the impact of each algorithm can be isolated to gain a better understanding of the strengths and weaknesses of each one in the context of the Acrobot simulation.

1.5 Performance Metrics

There are many different performance metrics that can be used to evaluate RL algorithms. Episode rewards is one of the commonly used metrics. It measures the total reward received by the agent in a single episode. This metric is useful for evaluating the performance of the agent in achieving the task objective. However, it may not provide a complete picture of the agent's overall performance, as the agent may achieve a high episode reward but struggle to generalize its behaviour to different situations. Episode rewards also don’t tell you how the algorithms has learnt and how it is making decisions.

Chapter 2

Literature Review

2.1 RL Research

RL is a branch of machine learning that teaches an agent to complete a task by looking at its environment and giving a reward based on how well it has completed an objective. This reward comes from a complex equation which changes depending on the RL algorithm you are using (Hinton, 1990). All RL algorithms aim to maximize the expected reward, as reward maximization lies at the core of the RL objective (Larsen, 2021). I will be using PPO, A2C, and TRPO, which maximize expected reward in different ways.

There are two main types of RL algorithms: model-based and model-free. Model-free algorithms learn through experience and are only able to get better after each iteration, but model-based algorithms learn the environment first and make predictions about the consequences of the actions.

There are also two measures of efficiency to consider when choosing which RL algorithms to use: data efficiency and computing efficiency. Data efficiency measures the amount of data used from the actual controlled system during learning. Computing efficiency measures the amount of computation the learning algorithm requires. In a perfect model-based RL algorithm the data efficiency can be extremely high, as little or no data from the physical system is required to learn optimal behavior. The actual system is only used to execute the result. This means that the learning algorithm may compute for long periods of time before finding good policies, so the computing efficiency can be quite low.

In a model-free reinforcement learning algorithm such as the algorithms I have chosen, the agent must be moved for each time step of learning algorithm execution, so the computing efficiency in terms of number of agent movements considered is reflected directly in the data efficiency (Atkeson, 1997).

For this project I will be comparing model-free algorithms as they are better fitted for the Acrobot environment since the information given at the start of the simulation is lacking, which is what model-based algorithms learn from, as well as the fact that model-free algorithms are suited to more dynamic environments, such as Acrobot. This reason as well as the fact that model-free requires less computational power is why I have chosen model-free algorithms.

Another variable that must be considered in RL is the tradeoff between exploration and exploitation. Agents must explore to improve the state which potentially yields higher rewards in the future or exploit the state that yields the highest reward based on the existing knowledge. Pure exploration degrades the agent's learning but increases the flexibility of the agent to adapt in a dynamic environment. On the other hand, pure exploitation drives the agent's learning process to locally optimal solutions (Yogeswaran, 2012). There are many different learning policies that have been used such as: greedy, ξ-greedy, Boltzmann Distribution (BD), Simulated Annealing (SA), Probability Matching (PM), multi-layer perceptron (MLP), and Optimistic Initial Values (OIV). I will be using MLP which takes in the current state of the environment and outputs a probability distribution over the possible actions the agent can take. It is a good choice due to its flexibility and ability to handle a wide range of environments and action spaces.

2.2 RL Algorithm Research

For this project, three popular reinforcement learning algorithms, A2C, PPO, and TRPO, will be used to train agents in the Acrobot environment. These algorithms were chosen because they are among the most used in reinforcement learning research and have been shown to perform well on a variety of tasks (Henderson et al., 2018).

A2C (Advantage Actor-Critic) is an on-policy algorithm that combines the advantages of both policy gradient and value-based methods. It uses two neural networks: a policy network that outputs the action probabilities, and a value network that estimates the state-value function. A2C is known for its low variance and fast convergence, making it a good choice for environments with high-dimensional state and action spaces (Wu, Y., Mansimov et al., 2017).

PPO (Proximal Policy Optimization) is also an on-policy algorithm that addresses some of the issues with earlier policy gradient methods, such as high variance and slow convergence. It achieves this by constraining the policy update using a clip range and a penalty term. PPO has shown to be robust and stable on a variety of tasks, including continuous control tasks (Schulman et al., 2017).

TRPO (Trust Region Policy Optimization) is an off-policy algorithm that uses a trust region constraint to ensure that policy updates do not deviate too far from the previous policy. This constraint ensures that policy updates are conservative and can help to avoid policy collapse. However, TRPO can be computationally expensive and requires careful tuning of hyperparameters (Schulman et al., 2017).

2.3 Environments

There are many different simple and complex physics simulations and environments used with RL. Some common ones include the cart-pole problem and the swing-up single or double pendulum problem. There is a lack of literature that has explored a multitude of simulations and papers tend to stick to the cart pole or single pendulum problems, such as (6). Therefore, I will use the double pendulum, Acrobot, environment in hopes of finding helpful information for how certain RL algorithms interact with this environment and gain a better understanding of how to apply these algorithms to other domains such as robotics.

2.4 Aims & Objectives

The aim of this project is to compare the performance of three popular reinforcement learning algorithms, A2C, PPO, and TRPO, in the context of physics simulations, in hopes to gain a better understanding of their relative strengths and weaknesses, and to identify which are best and most efficient at the Acrobot environment.

The objectives are as follows:

- Acquire the publicly available physics simulation Acrobot environment software and confirm that is compatible with reinforcement learning algorithms within four weeks.
- Locate and access open-source implementations of PPO, A2C, and TRPO reinforcement learning algorithms within six weeks.
- Implement and test the publicly available reinforcement learning algorithms in the Acrobot environment within five weeks.
- To tune the hyperparameters of each algorithm to identify the optimal values for learning rate, batch size, and clip range within 7 weeks.

- Complete and acquire final testing and results from the algorithm learning and evaluation within three weeks.
- Analyze and present the results of the comparison in multiple tables and graphs showing different performance metrics and statistics withing six weeks.

Chapter 3

Requirements Analysis

3.1 Simulation Software

The project requires access to a publicly available physics simulation software that is compatible with reinforcement learning algorithms. The software must provide a dynamic environment (Acrobot). Additionally, it should offer APIs or libraries that enable seamless integration with popular reinforcement learning frameworks, such as Stable Baselines.

3.2 RL Algorithm Requirements

The project must identify and obtain three publicly available reinforcement learning algorithms that are compatible with the simulation software and can be applied to the environment. These algorithms must be capable of learning from feedback and improving their performance over time.

The hyperparameters of these algorithms must also be able to be altered so that they can tuned to the environment and their performance can be optimised.

The project must execute each algorithm and compare their results in a table showing mean reward and standard deviation, as well as graphs showing multiple performance metrics. The comparison must consider the performance of the algorithms in the environment and their ability to learn and improve over time.

3.3 Gaps in Knowledge

There is a lack of research comparing the performance of specifically PPO, A2C, and TRPO in the Acrobot environment and therefore by conducting this research, this gap in knowledge will hopefully be filled and provide insights into which algorithms are most effective in controlling the Acrobot environment. This will contribute to the broader understanding of the strengths and weaknesses of the different RL algorithms in this comparison and aid in the development of more efficient and effective algorithms for real-world applications.

Another potential gap in knowledge is related to the generalizability of RL algorithms across different environments. While these algorithms have been shown to be effective in a variety of environments, it's unclear how well they will perform specifically in the Acrobot environment. This is particularly relevant given the unique characteristics of the Acrobot environment, due to it being unstable and nonlinear at times because of the second chaotic arm.

Chapter 4

Design & Methodology

4.1 Project Management

The tasks in this project will be managed using a task Gantt chart detailing times and deadlines for completion of key objectives. This will help to keep track of the progress of each task, assign deadlines and ensure that all tasks are completed on time.

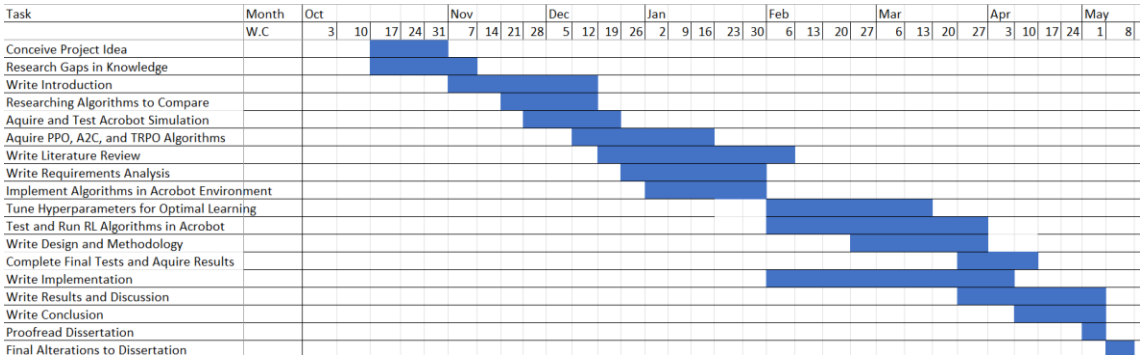


Figure 1: A Gantt chart showing times for key objectives.

Meetings with the supervisor will be held to discuss the progress of the project and to receive feedback and guidance when needed. This will help to ensure that the project is moving in the right direction and that any issues or concerns can be addressed.

For time management, the Pomodoro technique will be used in many writing and coding sessions. The Pomodoro technique is a time management method that uses a timer to break down work into intervals, traditionally 25 minutes in length, separated by short breaks. The technique aims to provide a balance between work and rest, increase productivity, and reduce the likelihood of burnout.

All code, RL algorithms, and documents will be kept in cloud storage to make sure they are easily accessible and secure in case of hardware or software failures.

4.2 Hyperparameter Tuning

Hyperparameters will be changed and tested all in the same way to ensure that the comparison between the algorithms is fair and unbiased. Each algorithm will be tested with a range of hyperparameter values, and the best performing values will be selected based on the performance metrics and reward outputs. The same performance metrics will be used for all algorithms (although some of the algorithms have performance metrics which are specific to them), and the results will be recorded in a systematic and organized manner to ensure reproducibility and ease of access.

These are the hyperparameters that will be manually set. The rest will stay as default.

The learning rate determines how much the algorithm adjusts the parameters based on the difference between predicted and actual values. A higher learning rate can lead to faster learning but may also cause instability.

The batch size determines the number of data points used to update the algorithm at each iteration. A larger batch size can lead to more stable updates but may also require more memory.

Gradient clipping is a technique used to prevent the gradient from becoming too large during training, which can lead to unstable updates. There is a maximum value which is set to stop it from exceeding a certain threshold.

4.3 Algorithm Teaching

Teaching the algorithms is the next key step. 100,000 timesteps has been chosen for the length of training. This choice has been made to try and find a balance between allowing the algorithms to gather sufficient experience and preventing excessive training time that could lead to overfitting or diminishing returns.

By allocating 100,000 timesteps for training, an adequate amount of interaction with the environment can be ensured, enabling the algorithms to explore various states and actions, learn from their experiences, and progressively refine their policies. This extended training duration allows for the accumulation of a diverse set of experiences, facilitating the exploration of different strategies and the discovery of optimal policies within the given task.

It is also essential to consider computational resources and practical constraints when determining the training duration. Longer training periods require more computational power and time. Therefore, the choice of 100,000 timesteps finds a reasonable balance between allowing for significant learning progress and respecting the available resources.

The algorithms will be trained in a vectorized environment, which allows for concurrent interactions with multiple instances of the environment. This parallelization offers several advantages, including increased computational efficiency, accelerated data collection, and potentially improved exploration. Furthermore, the ability to gather experiences from multiple instances allows for a more diverse and representative sampling of the environment, which can enhance exploration and potentially lead to more robust and generalized policies. This vectorized training approach offers a scalable and efficient means of training reinforcement learning

algorithms, enabling accelerated learning, and potentially achieving superior performance compared to sequential training.

As stated previously, all the algorithms will have the same set of hyperparameters to ensure a fair and consistent comparison across the different RL algorithms. By using identical hyperparameter settings, any observed differences in performance can be mainly attributed to the differences in the algorithms rather than the influence of hyperparameter tuning. This controlled experimental setup allows for a more reliable and meaningful evaluation of the algorithms' relative strengths and weaknesses, enabling a comprehensive analysis of their respective learning capabilities and their potential applicability to the given task.

4.4 Algorithm Testing

Next, the algorithms are tested to assess their performance and capabilities in the Acrobot environment. To achieve this, a total of 100 episodes will be executed for each algorithm. The choice of 100 episodes is based on the need for an adequate number of trials to obtain statistically reliable results and capture the algorithms' performance across a range of scenarios caused by the chaotic nature of the chosen environment. By conducting a sufficiently large number of episodes, the potential influence of fluctuations can be mitigated and obtain a more general evaluation of the algorithms' abilities to solve the given task.

4.5 Rewards and Performance Metrics

The rewards in the Acrobot environment are calculated by starting at 0 and -1 for each step taken. This means that a total reward will be taken for each episode and will then be calculated into a mean and a standard deviation for analysis. The mean provides an average performance measure, while standard deviation shows the consistency the algorithm. Analysing these

metrics allows for assessing the algorithms average success and stability in its performance.

Entropy loss is a good performance metric for analysis because it provides insights into the level of exploration or exploitation of the policy. Higher entropy indicates a more exploratory policy, which can be beneficial for discovering new and potentially better actions. Monitoring entropy loss helps ensure a balance between exploration and exploitation during training.

Explained variance is a valuable performance metric for analysis as it indicates how well the learned value function predicts the true values. Higher explained variance signifies that the value function captures a larger portion of the underlying dynamics of the environment, leading to more accurate predictions and better decision-making by the agent.

Policy gradient loss serves as a meaningful performance metric for analysis because it reflects the discrepancy between the predicted policy and the desired policy. By minimizing the policy gradient loss, the algorithm aims to improve the policy's alignment with the optimal behaviour, enhancing the agent's ability to select actions that yield higher expected rewards.

Policy loss is a crucial performance metric for analysis as it measures the quality of policy updates during training. By combining different loss components, such as policy gradient loss and entropy loss, policy loss provides a comprehensive assessment of the policy's performance and guides the optimization process to enhance the agent's decision-making capabilities.

Value loss is a significant performance metric for analysis because it quantifies the accuracy of the value function's predictions. A lower value loss indicates a better approximation of the true values, enabling the agent

to make more informed decisions based on reliable estimates of the expected returns. Monitoring value loss helps ensure the value function's effectiveness in capturing the value of different states or state-action pairs.

In TRPO the policy objective and KL divergence loss are commonly used as performance metrics because they are directly related to the key optimization objective and constraint of the TRPO algorithm, and as such are only accessible from TRPO.

The policy objective serves as a comprehensive performance metric for analysis in TRPO as it captures the trade-off between maximizing the expected return and maintaining policy stability. By maximizing the policy objective, TRPO aims to improve the agent's performance by increasing the expected return while adhering to a trust region defined by the KL divergence penalty. The policy objective provides a unified criterion that balances exploration and exploitation, allowing TRPO to optimize the policy in a principled manner. Monitoring the policy objective provides valuable insights into the overall performance and convergence of the TRPO algorithm.

KL divergence loss is a valuable performance metric for analysis in TRPO because it quantifies the dissimilarity between the current policy and the updated policy. By minimizing the KL divergence loss, TRPO ensures that the policy update remains within a predefined trust region, preventing abrupt and unstable changes in the policy. The constraint on KL divergence plays a crucial role in maintaining the stability and convergence of the TRPO algorithm by regulating the magnitude of policy updates and promoting smooth transitions between policy iterations. Monitoring KL divergence loss provides insights into the policy update process and ensures that the algorithm operates within a controlled range of policy modifications.

While policy objective and KL divergence loss are emphasized in TRPO, it's important to note that other performance metrics such as entropy loss, value loss, and explained variance are still relevant and valuable in assessing the agent's performance and the quality of learned policies. However, TRPO's primary focus is on the policy objective and maintaining a trust region through the constraint on KL divergence.

4.6 Risk Analysis

One issue that may arise is if the physics simulation software that is obtained is not able to produce what I is need or is not supported anymore, meaning there will not be enough online resources or documentation needed to understand the program. This is quite likely because many publicly available programs are no longer supported or are outdated, and most are only used for specific reasons and may not support the test which is needed. This will only cause minor time loss as it will not be difficult to obtain another program as there are plenty of free physics software online. I will make sure this does not happen by researching each program before using them to see if they are still supported and are able to do run the pole-balancing test. I will also make sure I have two to three other programs I can use in case the one that is currently in use is no longer suitable.

Another risk is that the physics simulation which I use may not be compatible with reinforcement learning and may not allow outside software to control it. This is not too likely as most physics simulators allows for outside software to control it and is usually made purposely for it. This will not cause a significant impact as I will only need to find a new physics simulator or different RL algorithms, all of which are plentiful online. I can mitigate this from happening by using a physics simulator that is purpose built for RL and encourages RL algorithms to be used on it. This will also help with usability as the documentation of the program will show how to use RL with it.

Another issue I may face is if the Acrobot environment does not produce satisfactory results with the RL algorithms and it is possible none of them will be able to learn how to complete it. This is semi-likely as the Acrobot environment is a fairly complex one, but all the algorithms chosen are also suited for these types of simulation. If this does occur, then it may cause a minor time loss as I will have to find a new environment to use which may take time to research and implement. To mitigate this I will have other environments in mind and test them with the algorithms as well.

Chapter 5

Implementation

5.1 Libraries

Library	Version
torch	2.1.0.dev20230307+cu117
tensorboard	2.12.0
stable-baselines3	2.0.0a5
sb3-contrib	2.0.0a4
pygame	2.1.3.dev8
Gym	0.26.2

Table 2: Table showing libraries used and their version.

Torch is an open-source machine learning framework primarily used for deep learning tasks. It is used in the project through the stable-baselines3 library to implement reinforcement learning algorithms. Torch is an excellent choice for RL tasks because it has efficient implementations of various RL algorithms, making it well-suited for complex tasks and environments.

Tensorboard is a tool for visualizing data and graphs related to machine learning. It has been used to log and visualize the training process of the reinforcement learning algorithms. Tensorboard is a popular choice for

machine learning visualization because it offers an easy-to-use interface and integrates seamlessly with other machine learning libraries.

Stable-baselines3 is a popular library for implementing and testing reinforcement learning algorithms. It provides a wide range of pre-implemented algorithms, including PPO, A2C, TRPO. I have used stable-baselines3 to implement PPO, A2C algorithms. Stable-baselines3 was a good choice because it is a well-maintained library with a large community, making it easy to find help if you encounter any issues.

Sb3-contrib is an extension of the stable-baselines3 library that provides additional reinforcement learning algorithms, including TRPO.

Pygame is a Python library used for game development and visualisation and it was used to render the environment so I could see how the algorithms had learnt to solve the task myself.

Gym is an open-source toolkit for environments which RL algorithms can be tested on, and I used it to get the Acrobot environment. Gym provides a wide range of environments, making it easy to experiment with different reinforcement learning algorithms and find one which suited this project well.

In summary, the libraries used in this project were all chosen for their specific strengths and suitability for the task at hand. The combination of these libraries allowed me to easily implement and test a range of reinforcement learning algorithms and effectively visualize and log the training process.

5.2 Random Agent Implementation

```python
import gym
env_name = "Acrobot-v1"                          # specify the environment to use
env = gym.make(env_name, render_mode='human')    # create an instance of the environment
episodes = 100                                   # Number of episodes the agent runs for

for episode in range(1, episodes + 1):           # loop over each episode
    state = env.reset()                          # reset the environment to the initial state

    terminated = False                           # initialize episode-specific variables
    score = 0                                    #

    while not terminated:                        # loop until the episode is terminated
        env.render()                             # render the environment

        action = env.action_space.sample()       # choose a random action

        obs, reward, terminated, truncated, info = env.step(action) # take a step in the environment

        score += reward                          # update the episode score

    print('Episode:{} Score: {}'.format(episode, score))           # print the episode score

env.close()                                      # Close the env
```

Figure 2: Code snippet showing a random agent being tested.

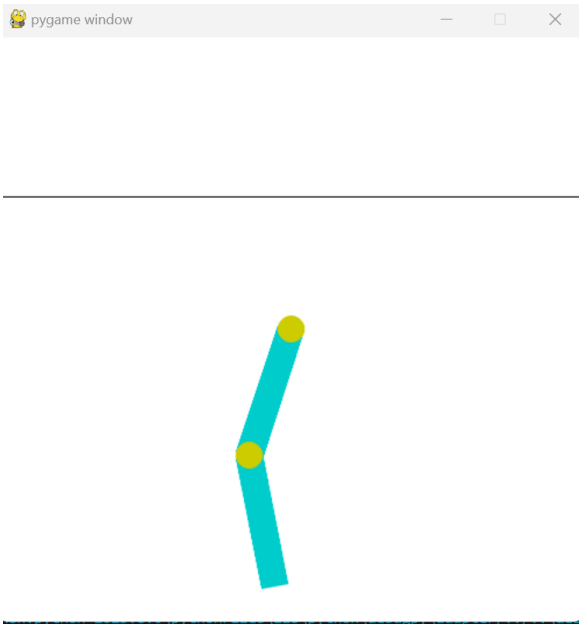


Figure 3: Pygame window showing random agent in environment.

First, I loaded the Acrobot environment with a random agent and ran it for a set number of episodes to ensure that the environment was properly configured and could be interacted with by the agent. This preliminary step is a common practice in RL research and helps to identify any issues or errors that may arise during training.

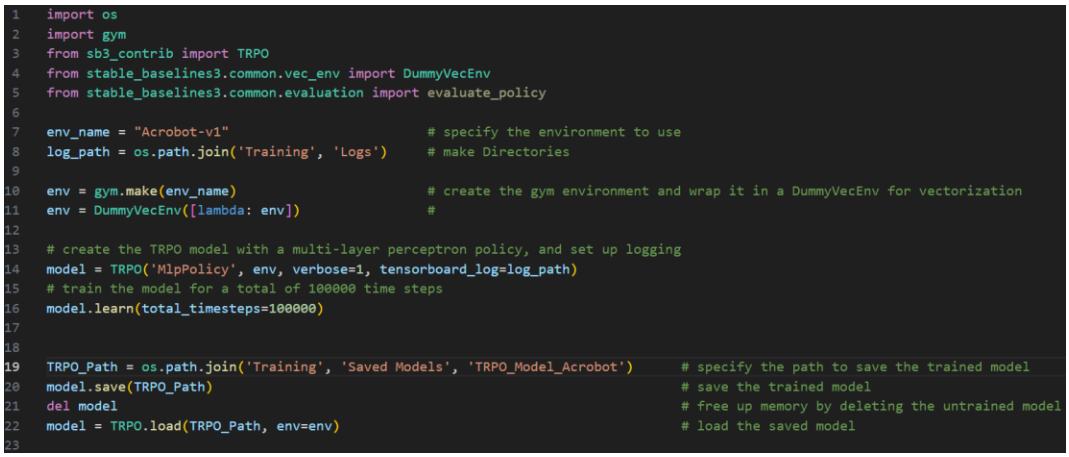
5.3 Default Algorithm Training

```python
import os
import gym
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.common.evaluation import evaluate_policy

env_name = "Acrobot-v1"                         # specify the environment to use
log_path = os.path.join('Training', 'Logs')     # make Directories

env = gym.make(env_name)                        # create the gym environment and wrap it in a DummyVecEnv for vectorization
env = DummyVecEnv([lambda: env])                #

# create the PPO model with a multi-layer perceptron policy, and set up logging
model = PPO('MlpPolicy', env, verbose=1, tensorboard_log=log_path)
# train the model for a total of 100000 time steps
model.learn(total_timesteps=100000)


PPO_Path = os.path.join('Training', 'Saved Models', 'PPO_Model_Acrobot')   # specify the path to save the trained model
model.save(PPO_Path)                                                       # save the trained model
del model                                                                  # free up memory by deleting the untrained model
model = PPO.load(PPO_Path, env=env)                                        # load the saved model

```

Figure 4: Code snippet showing default PPO being trained.

```python
import os
import gym
from stable_baselines3 import A2C
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.common.evaluation import evaluate_policy

env_name = "Acrobot-v1"                         # specify the environment to use
log_path = os.path.join('Training', 'Logs')     # make Directories

env = gym.make(env_name)                        # create the gym environment and wrap it in a DummyVecEnv for vectorization
env = DummyVecEnv([lambda: env])                #

# create the A2C model with a multi-layer perceptron policy, and set up logging
model = A2C('MlpPolicy', env, verbose=1, tensorboard_log=log_path)
# train the model for a total of 100000 time steps
model.learn(total_timesteps=100000)


A2C_Path = os.path.join('Training', 'Saved Models', 'A2C_Model_Acrobot')    # specify the path to save the trained model
model.save(A2C_Path)                                                       # save the trained model
del model                                                                  # free up memory by deleting the untrained model
model = A2C.load(A2C_Path, env=env)                                        # load the saved model
```

Figure 5: Code snippet showing default A2C being trained.



```python
import os
import gym
from sb3_contrib import TRPO
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.common.evaluation import evaluate_policy

env_name = "Acrobot-v1"                         # specify the environment to use
log_path = os.path.join('Training', 'Logs')     # make Directories

env = gym.make(env_name)                        # create the gym environment and wrap it in a DummyVecEnv for vectorization
env = DummyVecEnv([lambda: env])                #

# create the TRPO model with a multi-layer perceptron policy, and set up logging
model = TRPO('MlpPolicy', env, verbose=1, tensorboard_log=log_path)
# train the model for a total of 100000 time steps
model.learn(total_timesteps=100000)


TRPO_Path = os.path.join('Training', 'Saved Models', 'TRPO_Model_Acrobot')      # specify the path to save the trained model
model.save(TRPO_Path)                                                          # save the trained model
del model                                                                      # free up memory by deleting the untrained model
model = TRPO.load(TRPO_Path, env=env)                                          # load the saved model

```

Figure 6: Code snippet showing default TRPO being trained.

In the initial phase of the experiment, the PPO, A2C, and TRPO algorithms were applied to the Acrobot environment with their default hyperparameters to evaluate their performance and see their ability to learn the problem. This step was conducted to ensure that the algorithms were functioning correctly and capable of learning the problem without any modifications to the hyperparameters. The purpose of this step was to establish a baseline performance level for each algorithm, which could be compared against their respective performances after hyperparameter tuning.

`DummyVecEnv` is used in the code is which converts the environment into a vectorized one. Vectorized environments allow multiple parallel instances of the environment to be run simultaneously, which can improve training efficiency.

`model.learn()` is a method provided by Stable Baselines3 that trains the RL model using the specified algorithm. The `total_timesteps` parameter determines the total number of timesteps for which the model will be trained.

The code also creates a log directory using `log_path = os.path.join('Training', 'Logs')` so that all logs of training are saved as well as the final model itself.

Overall, the provided code sets up the environment, defines the hyperparameters for the PPO algorithm, trains the model for a specified number of timesteps, saves the trained model, and allows for later loading and usage of the trained model. Logging is also set up to track the training progress.

5.4 Hyperparameter Values

Hyperparameter:	**My Value**	**Default Value (PPO)**	**Default Value (A2C)**	**Default Value (TRPO)**
Learning Rate	0.001	0.0003	0.0001	0.0003
Gamma	0.98	0.99	0.99	0.99
Batch Size	96	64	NA	64
Number of Epochs	8	4	NA	NA
Clip Range	0.3	0.2	NA	NA

Table 2: Table showing hyperparameters for default and adjusted algorithms.

A higher learning rate can lead to faster initial learning progress by taking larger steps in the parameter space. This can be particularly useful when the initial exploration is slow. This can also be advantageous in situations where the agent needs to explore a wide range of actions and policies to discover optimal or near-optimal strategies. A higher learning rate also helps the algorithm escape local optima by allowing larger updates to the policy parameters. This can prevent the algorithm from getting stuck in suboptimal solutions and encourage exploration of potentially better policies.

However, it's important to note that changing the learning rate is not always beneficial. A higher learning rate can also introduce instability and cause the learning process to become erratic or even diverge. It's crucial to carefully tune the learning rate and monitor the learning process to ensure stable and effective training.

Changing the gamma hyperparameter from 0.99 to 0.98 can have several effects on the reinforcement learning algorithm. Gamma represents the discount factor used to weigh future rewards in the learning process. Increasing gamma can give more weight to future rewards, which can be beneficial for algorithms that require longer-term planning. On the other hand, decreasing gamma can give less weight to future rewards, which can be beneficial for algorithms that require shorter-term planning or when the reward structure changes rapidly. Lowering gamma could lead to faster convergence of the algorithm as the impact of future rewards on the learning process is reduced. This can lead to faster learning and more efficient use of computational resources. Lower gamma may also lead to improved stability and lower variance.

With a larger batch size, the algorithm can estimate the gradients more accurately. This can lead to more stable updates and faster convergence.

The increased batch size provides a larger sample of experiences, reducing the variance in gradient estimates and resulting in more reliable updates to the policy or value function.

Increasing the batch size may also help strike a better balance between exploration and exploitation. With a larger batch, the algorithm can explore a wider range of states and actions, potentially discovering more diverse and optimal strategies. This can be particularly useful in complex environments where exploration is crucial for discovering the best policies.

However, it's important to note that increasing the batch size also comes with trade-offs. Larger batch sizes require more memory and computational resources, which can limit scalability in certain settings. Additionally, larger batches might result in slower updates and potentially miss out on the most recent experiences.

Increasing the number of epochs allows the algorithm to perform more updates using the available data. This can lead to better utilization of the collected experiences and a more efficient learning process. With more epochs, the algorithm has multiple passes over the dataset, refining the policy or value function estimates and potentially improving the convergence rate.

Policy gradient algorithms (such as PPO), rely on optimizing the policy through gradient-based updates, and increasing the number of epochs can provide more opportunities for the algorithm to fine-tune the policy parameters and find better policies. It allows for a more thorough exploration of the parameter space, potentially leading to improved performance and more optimal policies.

It is important to note that increasing the number of epochs also has some trade-offs. Larger numbers of epochs require additional computational

resources and time, as each epoch involves processing and updating the policy based on the collected experiences. Additionally, increasing the number of epochs beyond a certain point may lead to overfitting, where the policy becomes too specific to the training data and performs poorly on unseen data.

The clip range plays a crucial role in balancing exploration and exploitation in policy optimization. A larger clip range allows for more exploration, as it permits larger updates to the policy parameters. By increasing the clip range from 0.2 to 0.3, the algorithm has the potential to explore a wider range of actions and policies, allowing for the discovery of novel and potentially better solutions.

In certain scenarios, the default clip range of 0.2 may be too conservative, resulting in underfitting of the policy. Underfitting occurs when the policy changes too little, preventing it from fully exploring and exploiting the environment. By increasing the clip range to 0.3, the algorithm can be more aggressive in updating the policy parameters, potentially mitigating underfitting and enabling faster and more effective learning.

Unfortunately, a larger clip range increases the potential for large policy updates, which may lead to instability or divergence if set too high.

TRPO and A2C don’t have access to all of these hyperparameters due to the fact that they can’t use them or don’t need them for learning. For example A2C doesn’t have access to batch size because A2C (Advantage Actor-Critic) algorithm is an on-policy reinforcement learning algorithm that updates its policy using data collected during a single interaction with the environment. Unlike off-policy algorithms such as PPO, A2C does not use a replay buffer or experience replay, which means it does not explicitly use a batch size parameter.

5.5 Trained Algorithm Testing

```python
import os
import gym
from stable_baselines3 import PPO
from stable_baselines3 import A2C
from sb3_contrib import TRPO
from stable_baselines3.common.evaluation import evaluate_policy

env_name = "Acrobot-v1"                                                    # specify the environment to use
env = gym.make(env_name, render_mode='human')                              # create the gym environment
model_Path = os.path.join('Training', 'Saved Models', 'TRPO_Model_Acrobot_Params')  # retrieve the trained model
model = TRPO.load(model_Path, env=env)                                     # and load it
episodes = 100                                                             # number of episodes the agent runs for
for episode in range(1, episodes+1):                                       # run algorithm in environment for each episode
    obs, _ = env.reset()                                                   # set the observation as the start of the environment
    terminated = False                                                     #initialize episode-specific variables
    score = 0                                                              #
    while not terminated:                                                  # loop while not terminated
        env.render()                                                       # render the environment
        action, _ = model.predict(obs)                                     # action is predicted by algorithm
        obs, reward, terminated, truncated, info = env.step(action)        # take the action
        score += reward                                                    # calculate reward
    print('Episode:{} Score: {}'.format(episode, score))                   # output score for each episode

env.close()                                                                # close the environment
```

Figure 7: Code snippet showing testing of TRPO algorithm with adjusted hyperparameters.

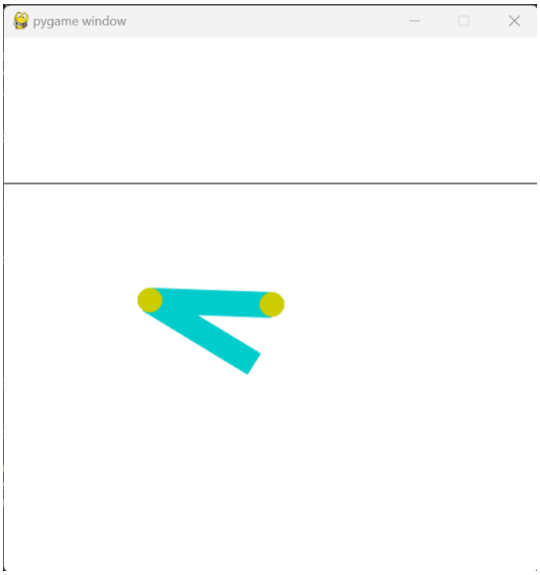


Figure 8: Pygame window showing TRPO agent in Acrobot environment.

When training an agent in a reinforcement learning task, it is important to measure its performance over multiple episodes. This is because an agent's performance in any given episode can be highly variable due to factors such as random initialization and chaotic variables in the environment. By running the agent for multiple episodes and averaging the results, a more reliable estimate of its true performance is achieved.

100 episodes were used for testing because it provides a good balance between computational efficiency and statistical reliability. With 100 episodes, a reasonable estimate of the algorithm's performance can be found while keeping the testing phase relatively quick. Additionally, 100 episodes is a commonly used number in literature and allows for easy comparison to other studies.

As for the code itself, actions are used, and they represent the decisions made by the RL agent to interact with the environment. The line `action, _ = model.predict(obs)` predicts the action to be taken by the RL agent based on the current observation (`obs`). The `model.predict()` method uses the trained RL model (`model`) to predict the action. Also in the code, the line `obs, reward, terminated, truncated, info = env.step(action)` executes the action in the environment and provides relevant information about the step and returns several variables:

`obs`: The new observation/state after the action is taken.

`reward`: The reward obtained from the environment based on the action.

`terminated`: A boolean flag indicating whether the episode is terminated (True) or not (False).

`truncated`: A flag specific to certain environments, indicating if the episode was truncated (True) or not (False).

`info`: Additional information provided by the environment (e.g., diagnostic information, debug data).

The code repeats the steps of predicting an action, executing the action in the environment, and receiving observations and rewards until the episode is terminated. By iteratively predicting actions and taking steps, the RL agent explores the environment, learns from the rewards it receives, and refines its policy to maximize cumulative rewards over time. The process continues for the specified number of episodes in the provided code, and the episode scores are printed as output.

Chapter 6

Results & Discussion

6.1 Episode Scores

Algorithm:	Mean Episode Score	Standard Deviation
PPO (Default Hyperparameters)	-86.04	25.32
PPO (Changed Hyperparameters)	-86.89	23.26
A2C (Default Hyperparameters)	-152.96	22.82
A2C (Changed Hyperparameters)	-83.37	20.02
TRPO (Default Hyperparameters)	-89.02	29.35
TRPO (Changed Hyperparameters)	-83.37	23.79

Table 3: Table showing mean episode score and SD for each algorithm.

The PPO algorithm with changed hyperparameters slightly outperformed the default version, as indicated by a slightly lower mean and standard deviation. However, the difference in performance between the two configurations is marginal.

In the case of A2C, adjusting the hyperparameters led to a significant improvement in performance. The algorithm with changed hyperparameters demonstrated a substantially lower mean score and standard deviation compared to the default configuration. This suggests that the selected hyperparameter changes positively impacted the algorithm's learning and stability.

Similar to PPO, the TRPO algorithm with changed hyperparameters exhibited a slightly better performance compared to the default configuration, with a lower mean and standard deviation. However, the improvements were not as pronounced A2C with changed hyperparameters.

One possible reason for the observed differences in performance between the modified and default hyperparameter configurations is the sensitivity of the algorithms to specific hyperparameter. While PPO showed only a slight improvement with changed hyperparameters, it suggests that the algorithm may be relatively robust to variations in its configuration. On the other hand, the significant improvement in A2C's performance with adjusted hyperparameters implies that the algorithm is more sensitive to the chosen settings, and small changes can have a notable impact. As for TRPO, the modest performance gains with modified hyperparameters indicate that the algorithm may have a moderate sensitivity to hyperparameter adjustments. These results highlight the importance of carefully tuning hyperparameters to maximize the performance of

reinforcement learning algorithms and the potential for fine-tuning specific configurations to achieve better results.

The algorithm with the best average score was A2C with adjusted hyperparameters, having a mean of -83.37. The algorithm with the lowest standard deviation is A2C with changed hyperparameters, with a value of 20.02.

These results suggest that A2C with adjusted hyperparameters not only achieved the best overall average performance but also demonstrated the most consistent and stable results, as indicated by the lowest standard deviation. This shows that the selected hyperparameter changes for A2C had a more significant positive impact compared to the other algorithms.

In summary, adjusting the hyperparameters yielded varying degrees of improvement for each algorithm. A2C with changed hyperparameters emerged as the top performer in terms of both average score and stability, while PPO and TRPO showed relatively smaller improvements. These findings highlight the importance of hyperparameter tuning in optimizing the performance of reinforcement learning algorithms.

The larger difference in results from default A2C to the hyperparameters I used may be due to the larger change in learning rate (going from 0.0001 to 0.001). This will create a larger difference in results because the learning rate determines the step size at which the algorithm updates the model's parameters based on the observed rewards and gradients. A higher learning rate can result in more significant updates, potentially leading to faster convergence or overshooting the optimal policy. Therefore, the larger change in learning rate from the default value in A2C may have influenced the algorithm's learning dynamics, resulting in the larger differences in performance.

6.2 Performance Metric Results

*All Graphs show smoothed line of best fit in dark blue and the unsmoothed original results in transparent blue.

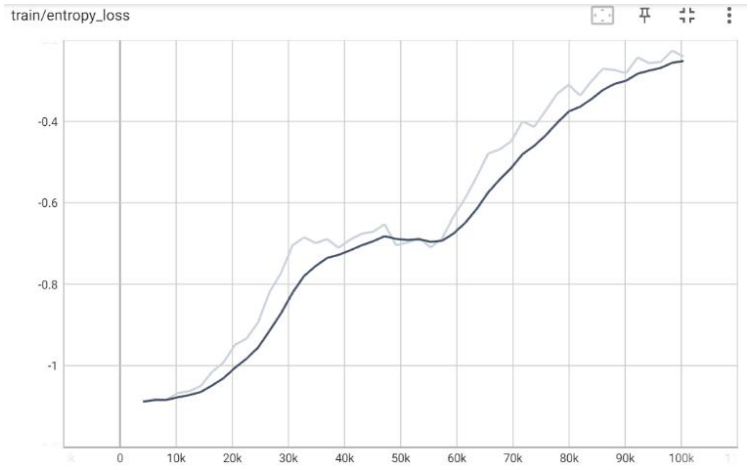


Figure 9: Tensorboard graph showing entropy loss for PPO.

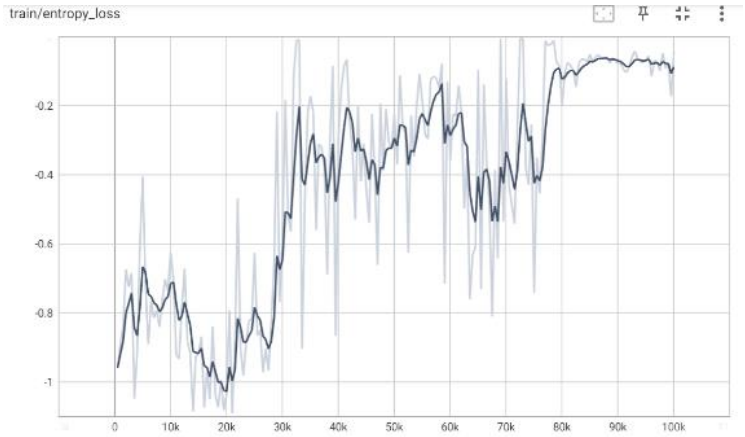


Figure 10: Tensorboard graph showing entropy loss for A2C.

The entropy loss graphs for PPO and A2C show the behaviour of entropy loss over time during the training process. Entropy loss is a measure of the policy's uncertainty or randomness. It indicates how diverse the policy's

actions are and can be used as a regularization term to encourage exploration.

In the case of PPO, the entropy loss graph demonstrates a steady incline which suggests that initially, the policy has a relatively high level of randomness or uncertainty in its actions, but the training progresses, the entropy loss decreases, indicating that the policy becomes more deterministic and less random. This reduction in entropy loss can be attributed to the optimization process that maximizes the expected return while balancing exploration and exploitation. The policy tends to become more focused on exploiting the actions that yield higher rewards rather than exploring random actions.

On the other hand, the entropy loss graph for A2C exhibits more fluctuations and variability compared to PPO. It fluctuates throughout the training process and eventually converging and stabilising. The fluctuations in the entropy loss suggest that the A2C algorithm explores a wider range of actions during training. It may periodically introduce more randomness into the policy to encourage exploration and prevent premature convergence to suboptimal solutions. The overall decreasing trend in entropy loss indicates that, over time, the policy becomes more deterministic and less exploratory.

From comparing the two graphs, it is evident that PPO's entropy loss has a steadier incline, suggesting a more gradual transition from a random policy to a more deterministic one. In contrast, A2C's entropy loss fluctuates more but still converges to a lower value. This difference in behaviour reflects the inherent characteristics and trade-offs of the two algorithms. PPO places a stronger emphasis on stability and avoids large policy updates, resulting in a smoother entropy loss curve. A2C, on the other hand, may

exhibit more exploration and variation due to its on-policy nature and the potential for more frequent policy updates.

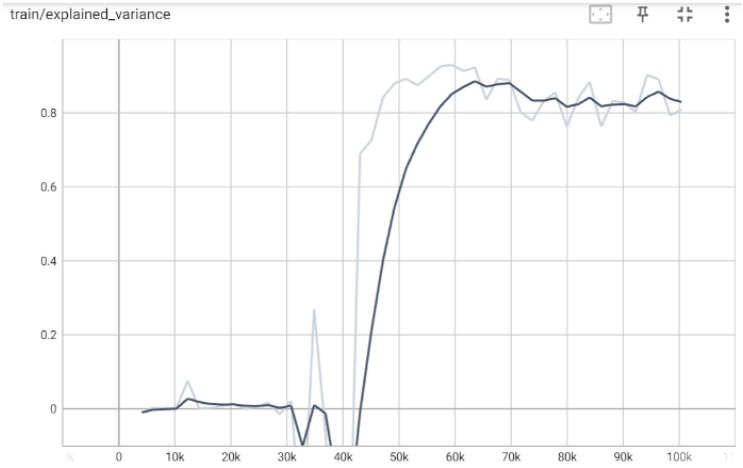


Figure 11: Tensorboard graph showing explained variance for PPO.

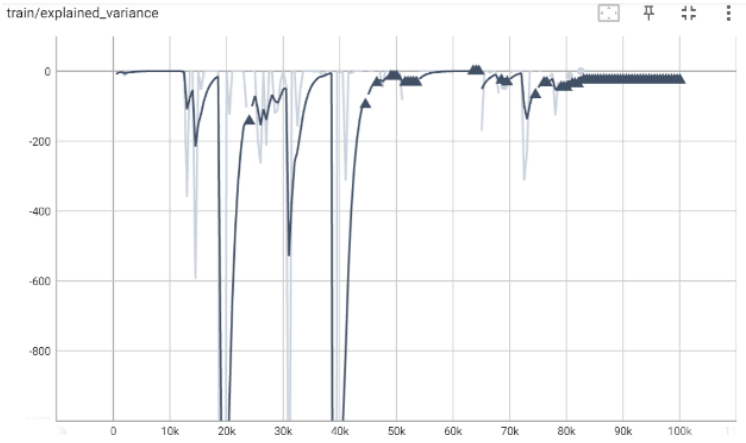


Figure 12: Tensorboard graph showing explained variance for A2C.

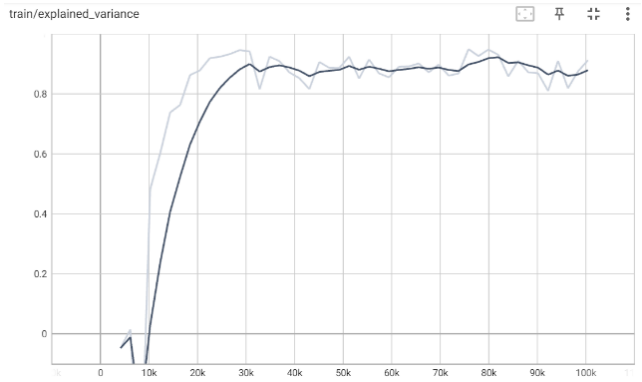


Figure 13: Tensorboard graph showing explained variance for TRPO.

The explained variance graphs for PPO, A2C, and TRPO provide insights into how well the value function approximations capture the variance in the observed returns or rewards. Explained variance measures the proportion of variance in the returns that can be explained by the value function.

Starting with PPO, its explained variance graph indicates that initially the explained variance remains close to 0. This suggests that the value function fails to capture much of the variance in the returns, indicating suboptimal performance. However, there is a subsequent dip followed by a rapid increase, indicating a significant improvement in the value function's ability to explain the observed returns. The value function stabilizes at this level, suggesting that it captures a substantial portion of the variance in the returns. This improvement demonstrates the algorithm's learning capability and the effectiveness of the training process in refining the value function.

In the case of A2C, the explained variance graph shows a tendency to stay near 0 for most of the training period. However, there are noticeable downward spikes throughout the graph, indicating instances where the value function fails to explain a significant portion of the variance in the returns. Additionally, the presence of NA values suggests certain episodes or timesteps where the value function encounters difficulties in providing accurate estimations. These large downward spikes and NA values indicate

challenges or inconsistencies in the learning process, potentially impacting the algorithm's overall performance.

For TRPO, the explained variance graph starts near 0 and quickly shoots. This rapid increase suggests that the value function captures a substantial proportion of the variance in the returns early in the training process. The graph stabilizes at a high level, indicating that the value function consistently explains a large portion of the observed returns. This early convergence of explained variance highlights TRPO's effectiveness in learning and approximating the value function quickly.

Comparing the three graphs, distinct patterns in the behaviour are observed for explained variance for each algorithm. PPO shows a slower initial improvement before reaching a stable level, indicating a gradual refinement of the value function. A2C exhibits frequent downward spikes and NA values, indicating potential challenges in capturing the variance in returns consistently. TRPO, on the other hand, demonstrates an early rapid increase in explained variance, suggesting a quick and effective learning process.

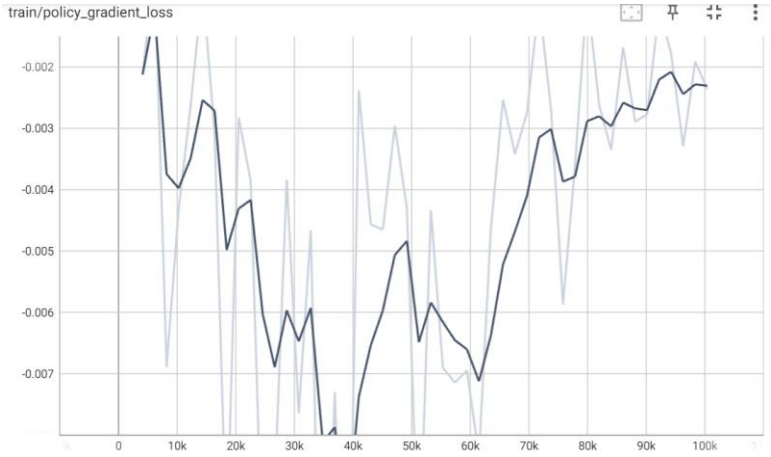


Figure 14: Tensorboard graph showing policy gradient loss for PPO.

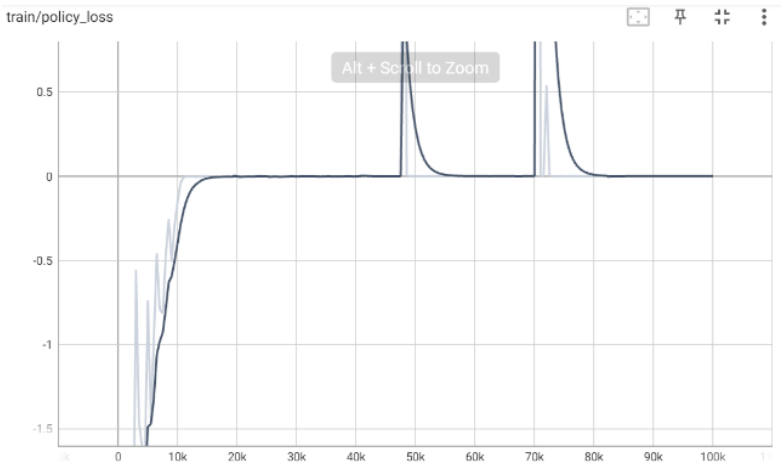


Figure 15: Tensorboard graph showing policy loss for A2C.

The policy gradient loss and policy loss graphs provide insights into the optimization of the policy during the training process for PPO and A2C algorithms.

Starting with the policy gradient loss graph for PPO, it begins near 0 and exhibits fluctuations over the first 35,000 timesteps. The gradual decline indicates a reduction in the policy gradient loss, which suggests an improvement in the policy optimization process. The fluctuations in the loss value indicate potential adjustments and fine-tuning of the policy based on the observed gradients. Overall, this graph demonstrates that the policy is being iteratively updated to optimize its performance throughout the training process.

For the policy loss graph for A2C, it starts significantly below 0 and quickly reaches a value of 0, where it stabilizes. This suggests that the policy loss is effectively minimized and converges to a satisfactory level early in the training process. The few sharp upward spikes observed in the graph may indicate occasional deviations or perturbations in the policy loss, but the policy quickly adjusts and returns to the optimal state. This stable policy loss graph suggests that the policy optimization process for A2C is efficient and reaches a desirable solution relatively quickly.

When comparing the two graphs, some differences in their patterns are observed. The policy gradient loss graph for PPO shows a gradual decline with fluctuations, indicating a more iterative and continuous refinement of the policy over time. In contrast, the policy loss graph for A2C quickly reaches a stable state with a value of 0, indicating that the policy optimization process achieves convergence early on.

These differences may stem from the variations in the algorithms' optimization objectives and mechanisms. PPO aims to strike a balance between exploration and exploitation, using a surrogate objective function to update the policy iteratively. On the other hand, A2C employs a more direct approach, optimizing the policy directly using the advantage function and minimizing the policy loss.

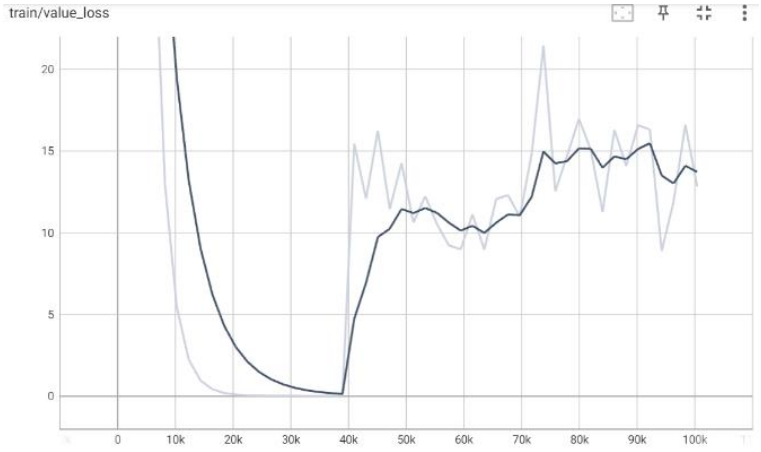


Figure 16: Tensorboard graph showing value loss for PPO.

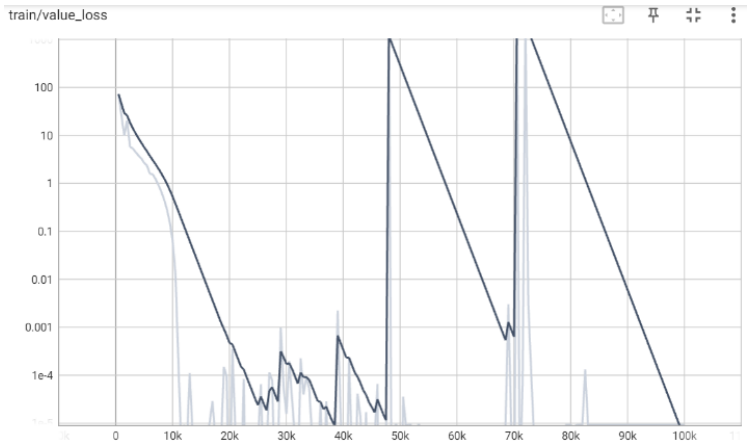


Figure 17: Tensorboard graph showing value loss for A2C.

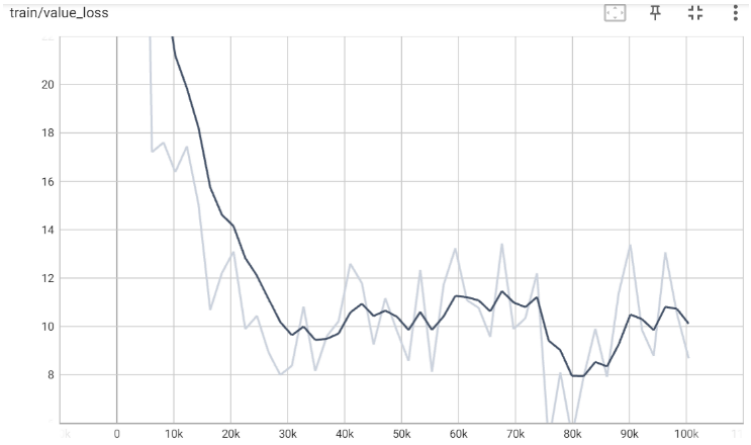


Figure 18: Tensorboard graph showing value loss for TRPO.

The value loss graph for PPO begins relatively high above 0 and gradually decreases over the first 40,000 timesteps, reaching a value near 0. This initial decline indicates that the value estimates produced by the algorithm are getting closer to the true values of the states or state-action pairs. However, the subsequent increase to around 10 suggests a deviation or suboptimal estimation in the value function. The stabilization at this value and the subsequent increase to about 14 indicate that the algorithm has reached a relatively stable value estimate for the given task. This stabilization suggests that the value function has reached a reasonable approximation of the true values.

For value loss graph for A2C, it starts near 100 and steadily falls to near 0 over 25,000 timesteps. This decline indicates an improvement in the accuracy of the value function estimates, as they are getting closer to the true values. The relatively stable region near 0 suggests that the algorithm has achieved a good approximation of the value function. However, the presence of sharp upward spikes indicates occasional perturbations or deviations in the value loss, which may be due to certain states or transitions that are more challenging to estimate accurately.

Examining the value loss graph for TRPO, it starts well above 0 and gradually decreases over the training period, reaching a value of about 10. This decline suggests an improvement in the value function estimates, as they approach the true values. The subsequent stabilization at this value indicates that the algorithm has reached a relatively stable estimation of the value function. The small dip at around 80,000 timesteps followed by an increase back to 10 suggests a minor fluctuation but does not significantly affect the overall convergence and stability of the value function.

When comparing the three graphs, differences in their patterns and convergence behaviours are observed. PPO's value loss graph shows an initial decrease, followed by an increase and stabilization, indicating a convergence to a reasonably accurate value function estimate. A2C's graph demonstrates a steady decline to near 0, with occasional spikes, indicating a relatively stable estimation with intermittent challenges. TRPO's graph exhibits a gradual decrease and stabilization, with a minor fluctuation but still maintaining a stable value estimate.

These differences may arise due to variations in the algorithms' value function approximation methods, optimization objectives, or exploration strategies. Each algorithm may prioritize different aspects of value

estimation and convergence, resulting in different patterns in their value loss graphs.

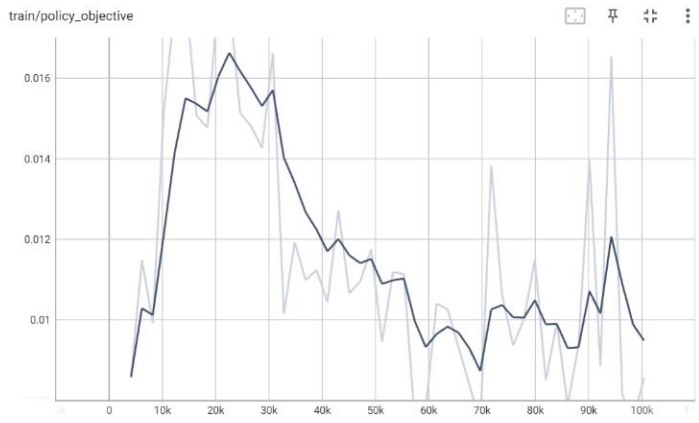


Figure 19: Tensorboard graph showing policy objective for TRPO.

The policy objective graph for TRPO demonstrates the optimization progress and convergence of the policy during training. It begins near 0, indicating that the initial policy objective is relatively close to the desired objective. The quick increase suggest an improvement in the policy's objective value, indicating that the policy is becoming more aligned with the desired behaviour.

As the graph continues, the policy objective steadily decreases, indicating further improvement in the alignment between the policy's actions and the desired behaviour. This decline suggests that the policy is converging towards a more optimal policy that maximizes the expected return or achieves the task objectives.

However, it's important to note that the subsequent increase does not necessarily indicate a degradation in the policy's performance. This slight increase could be due to fluctuations or fine-tuning of the policy during the training process. It's possible that the algorithm is exploring alternative actions or adjusting optimize the policy further.

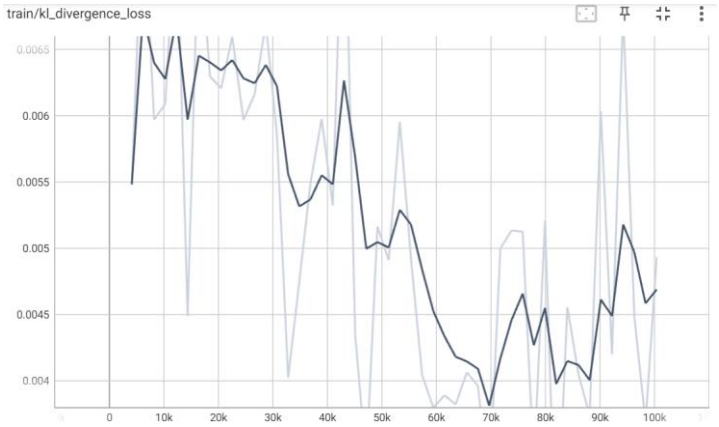


Figure 20: Tensorboard graph showing KL divergence loss for TRPO.

The KL divergence loss graph for the TRPO algorithm provides insights into the alignment between the new policy and the previous policy during the training process. The initial value indicates a relatively low divergence between the new policy and the previous policy, suggesting that the updates to the policy are conservative and do not deviate significantly from the existing policy.

As the training progresses, the KL divergence loss increases which indicates that the updates to the policy are becoming more substantial, and the new policy is starting to diverge further from the previous policy. This increase in divergence suggests exploration and adaptation in the policy, as the algorithm searches for potentially better policy configurations.

However, the subsequent decline in the KL divergence loss signifies a reduction in the divergence between the new and previous policies. This indicates that the updates to the policy have been adjusted to align more closely with the previous policy. The decrease in divergence suggests that the algorithm is finding a balance between exploration and exploitation, refining the policy to achieve better performance while still leveraging the knowledge gained from the previous policy.

The final value suggests that the policy has reached a state where the updates are relatively conservative, and the divergence from the previous policy has been minimized. This convergence indicates that the TRPO algorithm has found a policy that optimizes the expected return or achieves the task objectives while maintaining a certain level of similarity to the previous policy.

Chapter 7

Conclusion

In this study, the performance of Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C), and Trust Region Policy Optimization (TRPO) algorithms were investigated. By adjusting hyperparameters and analysing various metrics, valuable insights were gained into their behaviour and performance in the Acrobot environment.

One limitation of the study, however, is the focus on a specific task, which may restrict the generalizability of the findings to other domains. Future work should involve evaluating these algorithms on a diverse set of tasks, including complex and real-world environments, to assess their robustness and applicability across different problem domains.

Despite this limitation, the study yielded several successes. Observations for adjusting hyperparameters showed to have a positive impact on the algorithms' performance, with all adjusted algorithms outperforming their default counterparts. This demonstrates the importance of hyperparameter tuning in maximizing algorithm effectiveness. Additionally, the A2C algorithm stood out as the most improved algorithm, both in terms of average score and standard deviation. This success highlights the potential

for further exploration and refinement of hyperparameters to enhance algorithm performance.

Moreover, the analysis of various metrics provided valuable insights into the learning dynamics of these algorithms. Distinct patterns were observed in entropy loss, explained variance, policy gradient loss, value loss, and policy objective graphs for each algorithm. These patterns shed light on the exploration-exploitation trade-offs, policy refinement processes, and value estimation accuracy. Such insights can guide future algorithm development and shed light on their inner workings.

While the study offers important contributions, there are several opportunities for future work. Firstly, exploring additional hyperparameter configurations beyond the ones considered in this study can provide a deeper understanding of their effects on algorithm performance. Furthermore, investigating the algorithms' performance on a wider range of tasks can reveal their strengths, limitations, and potential for transfer learning.

Additionally, future research could focus on improving and developing methods to extract actionable insights from these results and find further use in real-world applications.

Lastly, evaluating the scalability and efficiency of these algorithms on large-scale and distributed computing systems can enable their application to more complex and data-intensive tasks.

In conclusion, this study has provided valuable insights into the behaviour and performance of PPO, A2C, and TRPO algorithms. It has highlighted their successes, such as the impact of hyperparameter tuning and the improved performance of the A2C algorithm. Moreover, limitations have been discussed, including the task-specific nature of our study, and

identified opportunities for future research. By addressing these limitations and exploring the suggested avenues, advancements in the field of reinforcement learning can be made to unlock the full potential of these algorithms in solving complex real-world problems.

References

Kober, J., Bagnell, J.A. and Peters, J., 2013. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*

G. E. Hinton, ìConnectionist learning procedures,îArticial Intelli-gence, vol. 40, no. 1ñ3, pp. 185ñ234, September 1989.

Larsen, T.N., Teigen, H.Ø., Laache, T., Varagnolo, D. and Rasheed, A., 2021. Comparing deep reinforcement learning algorithms' ability to safely navigate chal-lenging waters. Frontiers in Robotics and AI, 8, p.738113.

Atkeson CG and Santamaria JC (n.d.) A comparison of direct and model-based re-inforcement learning. In: .

Yogeswaran M and Ponnambalam SG (2012) Reinforcement learning: exploration–exploitation dilemma in multi-agent foraging task. OPSEARCH, OPSEARCH 49(3): 223–236.

Nagendra S, Podila N, Ugarakhod R, et al. (2017) Comparison of reinforcement learning algorithms applied to the cart-pole problem. In: .

Reinforcement Learning: An Introduction Richard S. Sutton and Andrew G. Barto (chapter 6.4)

Wiering M and Schmidhuber J (1998) None. Machine Learning, Machine Learning 33(1): 105–115.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D. and Meger, D., 2018, April. Deep reinforcement learning that matters. In Proceedings of the AAAI conference on artificial intelligence (Vol. 32, No. 1).

Wu, Y., Mansimov, E., Grosse, R.B., Liao, S. and Ba, J., 2017. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. Advances in neural information processing systems, 30.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.