UNIVERSITY OF LEEDS

Deep Reinforcement Learning for Autonomous Vehicles

Osman Fadl Ali
SID: 201337691
Date: 01/05/2023

SCHOOL OF MECHANICAL ENGINEERING

UNIVERSITY OF LEEDS

MECH3895 – Individual Engineering Project

PROJECT TITLE: Developing a Deep Reinforcement Learning Model Using Proximal Policy Optimization Techniques for Autonomous Vehicles	
PRESENTED BY	Osman Fadl Ali
SUPERVISED BY	Yanlong Huang

If the project is industrially linked, tick this box and provide details below ☐

COMPANY NAME AND ADDRESS:

Date: 01/05/2023
Signed: [illegible]

Table of Contents

Abstract

The aim of this project is to develop deep reinforcement learning models using proximal policy optimization algorithms for the application of autonomous driving, and investigate the effect of varying the reward function on the model performance. The chosen environment for this project is the OpenAI Box2D Car Racing Environment, which simulates a top-down view of the autonomous car agent navigating through a procedurally generated race track. Several models are trained using a convolutional neural network for different durations to identify the optimal training time. Model performance is examined using visual and numeric evaluation metrics to understand the strengths and weaknesses of each model. Systematic methods including grid-based search techniques are employed to tune the hyperparameters of the model. The tuned model achieves higher reward and exhibits more desirable behavior despite going off the track in some scenarios. To solve this problem, the default reward function is modified by adding a steering penalty that penalizes sudden steering changes. This has a significant impact on the agent's behavior, resulting in smoother driving and better track adherence while considering the trade-off between performance and safety. Thereby, this project provides valuable insights into development of useful reinforcement learning models for autonomous vehicles, and emphasizes the importance of proper training, hyperparameter tuning, and reward function design in shaping agent behavior and achieving desired performance outcomes.

Chapter 1: Introduction

1.1. Introduction to Reinforcement Learning Project

Reinforcement Learning (RL) is an active area of machine learning research that has become one of the leading research fields of Artificial Intelligence. Essentially, RL is a framework in which an agent learns how to interact with an environment from experience, particularly through trial-and-error [1]. Unlike other machine learning methods, RL entails dynamic learning by learning from interaction with the environment, and adjusting actions based on the continuous feedback of a reward signal. Therefore, the most common strategy when training an RL agent is to take actions that would maximize reward. This strategic behavior is referred to as policy, and is often denoted by π.

The foundations of RL can be traced back to the 1950s, when Richard Bellman addressed the optimal control problem with respect to a loss function [2]. His work employed RL techniques to solve the Markov Decision Process (MDP), which subsequently became the most prevalent form for defining RL problems [1]. RL algorithms tackle these problems by guiding the agent in learning to approximate the optimal policy while directly interacting with the environment. In the context of control, this entails the online approximation of solutions to stochastic control problems [3]. As a result, the study of RL is intrinsically connected to not only control, but also decision theory, optimisation, and dynamic programming.

In recent years, the field of RL has been growing rapidly, particularly with advancements in Deep Reinforcement Learning, which combines the dynamic framework of RL with artificial neural networks. This resulted in notable accomplishments, such as the widely publicized victory of the computer program AlphaGo, which garnered significant media attention when it triumphed over world champion Lee Se-dol [4]. This victory piqued the interest of researchers, while simultaneously shedding light on some of the challenges of deep RL. One of these challenges is the curse of dimensionality, which refers to the exponential growth in the number of parameters to be learned as the size of any compact encoding of the system state increases [3]. Another major hurdle is devising an appropriate reward system that enables the agent to explore and exploit the environment, something that will be explored in this project.

Formulating an adequate reward system does not necessarily solve the problem, as RL agents face the challenge of sparse rewards, where the feedback received from the environment is infrequent or delayed. This scarcity of informative signals can hinder the learning process, as the agent can struggle to identify the correlation between its actions and the resulting reward. This issue is exacerbated in complex environments, such as those involving autonomous vehicles, where the agent must learn to make a sequence of decisions to accomplish a long-term goal such as a car agent completing a race track.

Therefore, the goal of this project is to use proximal policy techniques build a deep reinforcement learning model for autonomous vehicles by training the agent in the autonomous driving environment, and investigate the effect of the reward function on the learned policy. By focusing on the reward function, the sparse rewards problem can be tackled by modifying the original reward function and using different reward shaping techniques to analyze their impact on the agent's performance, ultimately aiming to develop a model that can effectively navigate the challenging environment and make optimal decisions. Both visual and numeric evaluation techniques will be used to examine and evaluate the performance of the RL models, which includes analyzing the effect of hyperparameters on model performance and deciding on the appropriate neural network to approximate a viable solution to the problem.

Through this investigation, this project will provide an insight into the interplay between reward functions and policy learning but also contribute to the broader goal of developing more robust and efficient RL models for autonomous vehicles.

1.2. Project Aims

The aim of this project is to develop and train a deep reinforcement learning model using state-of-the-art proximal policy optimization techniques for autonomous vehicles, and investigate the effects of varying the reward function on the model performance.

1.3. Project Objectives

Conducting a literature review on RL and installing the required software and libraries is a prerequisite to the objectives below. The objectives are as follows:

1. Designing and implementing a deep reinforcement learning model by leveraging proximal policy optimization (PPO) algorithms to address the challenges associated with the autonomous driving environment.
2. Conduct extensive model training and evaluation to identify the best model and ensure successful adaptation of the RL model to the continuous environment.
3. Perform systematic hyperparameter tuning to identify the most favorable parameter configuration that will enhance model performance.
4. Use different reward shaping techniques to analyze the influence of varying reward functions on the agent policy behavior.

5. Compile a thorough analysis of experimental results and findings by exploring the effectiveness of the proposed model and its potential application in real-world autonomous vehicles.
6. Derive conclusions and identify potential avenues for future research.

1.4. Report Layout

The next chapter will provide a comprehensive review of the underlying theory of RL, outlining the core elements of an RL framework including environments, states, actions, and rewards. The chapter delves into the taxonomy of RL algorithms, and introduces the key algorithm used in this project. Chapter 3 explores the nature of RL environments with particular focus on how environment characteristics influence the choice of algorithm and exploration strategy. The chapter then introduces the car racing environment with remarks on its action spaces, reward structure, and the type of neural network needed to address the problem. Chapter 4 explains the process of training and evaluating the RL model within the context of the autonomous driving environment. Chapter 5 outlines the various hyperparameters associated with the model and how they were tuned to enhance model performance. Chapter 6 provides insight into the interplay between varying the reward function and policy optimization in the realm of autonomous navigation. The final chapter provides the main takeaways from the project and discusses key findings and future research. These chapters cohesively build on one another to facilitate the aim of developing deep RL models for autonomous vehicles.

Chapter 2: Overview of Reinforcement Learning

2.1. Markov Decision Process (MDP)

MDP is a mathematical framework used to model dynamic decision-making problems in which an agent interacts with the environment in discrete timesteps [5]. MDPs are widely used in RL to represent the underlying structure of the problem the agent is trying to solve. An MDP is defined by a tuple *(S, A, P, R, γ)* where:

1. *S* is a finite set of states, representing the possible configurations of the environment.
2. *A* is a finite set of actions, representing the possible decisions the agent can make in the environment.
3. *P(s' | s, a)* is the state transition probability function, which defines the probability of transitioning from state *s* to *s'* given the agent takes action *a*. This highlights the dynamics of the environment.
4. *R(s, a, s')* is the immediate reward the agent receives for taking action *a* in state *s*, before transitioning into state *s'*.
5. *γ* is the discount factor, which is a real number between 0 and 1 that determines the weight of future rewards with respect to the immediate reward. The choice of *γ* is critical as it dictates how the agent prioritises short-term rewards over long-term rewards. This will be discussed in a later section.

The MDP model provides a structured way of concisely representing the environment's states, actions, dynamics and reward thereby enabling the agent to learn a policy that maximises reward through systematic exploration and exploitation.

2.2. The Agent-Environment Interaction Loop

The agent-environment interaction loop is a diagram that illustrates the process by which the agent interacts with the environment, receives feedback, and updates its knowledge dynamically. The figure below shows the diagram:

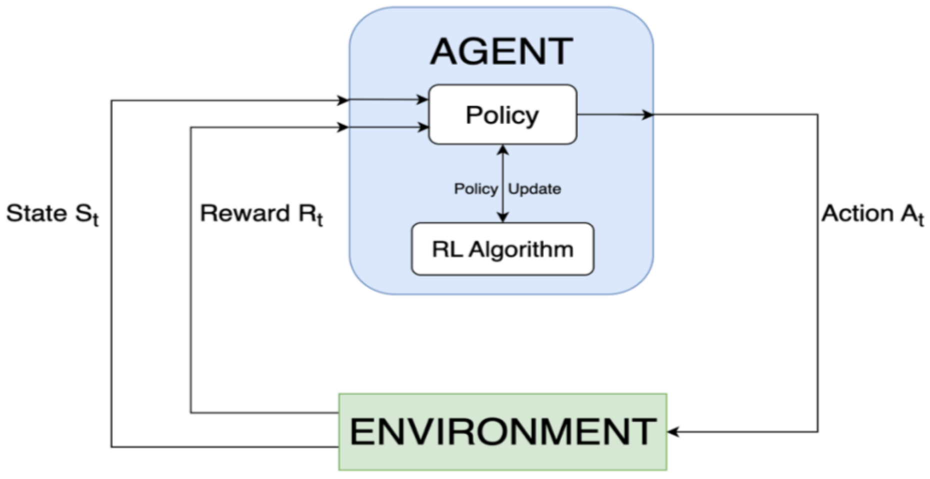


Figure 1: Agent-Environment Interaction Loop

The diagram above illustrates the interaction between an agent and the environment in an RL setting. At each timestep t, the agent in state S_t, takes an action A_t according to its policy. The agent receives a reward R_t before transitioning onto the next state S_{t+1}. The agent uses the feedback signal of the reward and the new state to update its policy and improve its knowledge of the environment. This process is achieved through an RL algorithm. The interaction loop continues until a terminal state is reached, or a predefined stopping criterion has been reached.

2.3. Quality and Value Functions

In order to understand the underlying theory behind RL algorithms, one must be aware of the quality and value functions, and how they underlay the foundation of the most common RL methods and algorithms. The state value function measures the value of an RL agent being in a specific state [6], computing how good any given state is for an agent following policy π. The quality value function is an equation that measures the value of the agent taking an action a_t under policy π, returning the expected reward from the action in that state. Many RL algorithms compute these functions to update their policies iteratively with the goal finding the optimal policy with maximum reward. This is a fundamental concept in RL called dynamic programming [6]. The quality function equation is seen below:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)] \qquad (1)$$

Equation 1: Quality Function [7]

The quality function above shows that the quality at each timestep depends on state *s* and action *a*, and it equates to the current quality estimate, $Q(s,a)$, plus the temporal error. The temporal error consists of two terms: the observed cumulative reward, represented by $[r + \gamma maxa\ Q(s,\ a)]$, which is the actual reward, r, plus the maximum future quality function at the next step, and the second term is the quality function estimate. This is multiplied by the learning rate, α, a hyperparameter that can be tuned to ensure evaluative feedback. Essentially, this update rule uses the value of the quality function to determine the next best action, and it represents the foundations of temporal difference learning. Temporal difference (TD) learning is a primary approach in RL derived from dynamic programming [6], that gives rise to some of the most common algorithms in RL: SARSA and Q-learning.

2.4. RL Methods and Algorithms

RL algorithms can be broadly classified into three main classes: value-based methods, policy-based methods, and actor-critic methods.

Value-based methods focus on learning the quality and value functions, and uses that to estimate the expected cumulative reward from a given state-action pair. By learning the value function, the agent can eventually derive the optimal policy that maximises the expected reward. Common value-based methods include dynamic programming algorithms such as Value Iteration Algorithm [7], which uses knowledge of the complete MDP model to compute the optimal value function and policy. TD learning algorithms, such as Q-learning, learns the value function by bootstrapping from estimates of future values without requiring a complete model of the MDP. Therefore, value-based methods are more suitable in model-free RL settings.

Policy-based methods directly learn the policy, by mapping from states to actions, without explicitly computing the value function. Policy-based algorithms aim to optimise policy by iteratively updating its parameters based on the collected experience with the environment. Key policy-based methods include policy gradient algorithms such as REINFORCE and Natural Policy Gradient, which use a mathematical technique called gradient ascent, to update the policy parameters in the direction that maximises expected reward [7].

Actor-critic algorithms combine both value-based and policy-based approaches. They consist of a critic (value) network that is used to estimate the value function, and an actor (policy) network that optimises the policy according to the estimated value function [8]. The critic is used to evaluate the current policy, while the actor is updated based on the critic's feedback. Examples include DDPG (Deep Deterministic Policy Gradient) and the PPO (Proximal Policy Optimisation) algorithm, which is the main algorithm used in this project. Actor-critic algorithms combine the strengths of value-based and policy-based approaches and help balance the exploration and exploitation of the environment, which is one of the reasons an actor-critic algorithm was chosen for this project. By presenting these classes of methods and algorithms, one can now understand the taxonomy and overview of RL techniques as seen below:

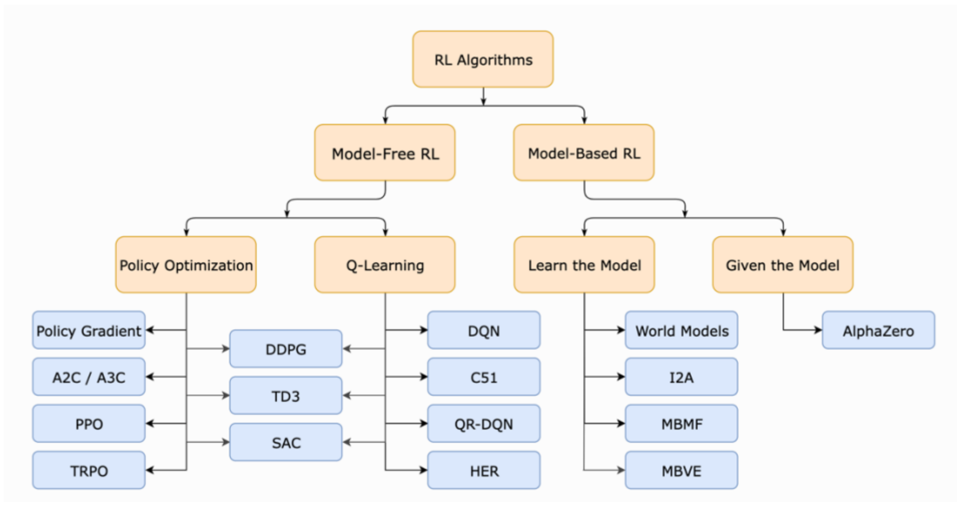


Figure 2: Taxonomy of Reinforcement Learning Algorithms [15]

2.5. Proximal Policy Optimization Algorithms

Proximal Policy Optimization (PPO) is a deep RL algorithm based on the actor-critic architecture. PPO originally belonged to the family of policy gradient methods and was created with the aim of addressing challenges of stability and sample efficiency in training RL agents. PPO was builds upon the concepts of another algorithm, Trust Region Policy optimization (TRPO), which enforces that policy updates stay within a trust region to prevent large updates that could hinder the learning process [8]. TRPO guarantees improvement in policy updates, but has high computational complexity as it is a second-order optimization method. PPO, on the other hand, is a first-order optimization algorithm meaning it relies on the gradient (first-order derivative) of a function to update the model parameters [8]. This function is called the clipped surrogate objective function, and it eliminates the need for complex second-order optimization methods. Therefore, PPO constrains the policy update step, preventing overly large policy updates whilst providing stability at the same time. Below is the clipped surrogate objective function:

$$L(\theta) = E_t[\min\left(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)] \qquad (2)$$

Equation 2: Surrogate Objective Function [9]

Where:

θ represents the policy parameters

$r_t(\theta)$ is the ratio of new and old policy probabilities

A_t is the advantage function, which estimates how much better an action is compared to the average action at a given state

ϵ is a hyperparameter that controls the clipping range

The second term, $clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t$, modifies the surrogate objective by clipping the probability ratio, removing the incentive of moving r_t outside the interval $[1-\epsilon, 1+\epsilon]$

In his paper, Schulman [9] acknowledges the issue of stability for actor critic algorithms, and proposes a generalised version of PPO for continuous domain tasks by modifying the policy gradient implementation. His results show that PPO outperforms a range of actor critic algorithms on almost all continuous control environments. The PPO algorithm not only performs exceptionally well in continuous domains, but can be tuned to become more applicable in a variety of environments including Atari games, humanoid simulation, and autonomous driving, which is the goal of this project.

Chapter 3: Reinforcement Learning Environments

3.1. Introduction

This chapter will discuss RL environments, their characteristics and how that affects the choice of algorithm for RL problems. This chapter will provide a detailed overview of the autonomous driving environment, which serves as a primary focus on this project. Moreover, the chapter will use key aspects of the environment to outline the type of neural network used for this project.

3.2. Environment Characteristics

3.2.1. Discrete vs Continuous Environments

Whether an environment is discrete or continuous makes a huge difference when it comes algorithm selection. This is mainly due to the difference between discrete and continuous actions spaces. A classical chess board, for example, has a discrete number of action spaces whereas a robotic arm has high-dimensional continuous action space, making it more difficult to address, and therefore requires a different algorithm than its discrete counterpart. Therefore, discrete action spaces tend to work well with value-based methods like Q-learning, while continuous action spaces may require policy-based or actor critic algorithms such as PPO.

3.2.2. Fully Observable vs Partially Observable Environments

RL environments can be categorized into fully observable and partially observable environments. In fully observable environments, the agent has complete knowledge about the current state of the environment. This enables the agent to observe all relevant aspects of the environment necessary to make optimal decisions. MDPs are often used to model fully observable environments. A classical chess board is an example of a fully observable environment as the agent can observe the entire board and has complete knowledge of all the pieces in the board.

A partially observable environment is one where the agent has limited information about the current state of the environment, meaning the agent cannot observe all aspects of the environment and must therefore make decisions based on partial information. In the context of autonomous vehicles, environments are typically partially observable as the vehicle relies on sensor data, like cameras and radar, which have limited range and coverage. Therefore, the algorithm chosen must be capable of handling uncertainty and making decisions based on the current state of the environment.

3.2.3. Sparse vs Dense Rewards

In RL, the reward function of an environment plays an important role in facilitating the learning process of an agent. In dense reward environments, the agent receives frequent feedback in the form of rewards or penalties as it interacts with the environment. Having frequent rewards after taking action allows the agent to learn which actions are beneficial and which are not. This can guide the agent's learning process, often leading to faster convergence and more reliable learning.

As mentioned in the introduction section, sparse rewards is a big challenge in RL, and one that occurs when the agent receives feedback infrequently or only upon reaching certain milestones. This can hinder the learning process, as the agent must explore the environment and identify optimal actions without much guidance from the reward signals hence slower learning. In the context of autonomous vehicles, different reward shaping techniques can be used to ensure frequent feedback. For example, dense rewards can be provided to control tasks such as staying in the lanes, or for following traffic rules and maintaining a safe distance from other vehicles.

3.3. Autonomous Driving Environment: Car Racing Environment

3.3.1. Environment Description and Installation

The autonomous driving environment for this project is the Box2D Car Racing environment, developed by OpenAI [10]. Built using the Box2D physics engine, the environment simulates a 2D top-down racing game where the agent is a car that navigates through a procedurally generated racetrack. The agent's objective is to maximize the distance covered in the track within a given time limit while staying on the track. Therefore, this project will investigate the control task of lane keeping for autonomous driving, simulated by the car racing environment. The figure below shows a snippet of the environment. The number on the bottom left shows the current reward:

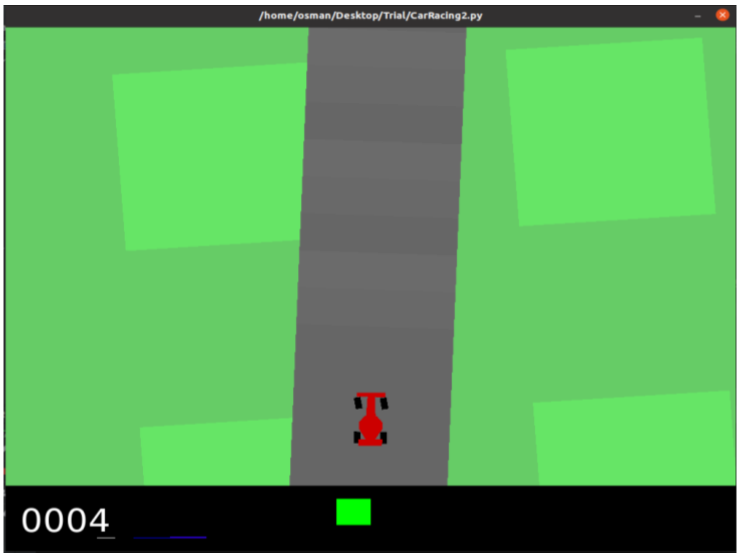


Figure 3: Box2D Car Racing Environment

To install and render the environment, the OpenAI gym library, which contains the car racing environment had to be installed along with the Box2D physics engine. To implement algorithms, Stable-Baselines3 was installed. Stable-Baselines3 is a library that provides a set of high-quality implementations of RL algorithms in Pytorch.

3.3.2. Environment Composition

According to the OpenAI gym documentation [10], the car racing environment consists of the following:

1. **Action Space**: there are 3 actions, steering (-1 is full left, +1 is full right), gas and breaking.
2. **Observation Space**: the state always consists of 96x96 pixel image.
3. **Reward Function**: The reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles visited in the track. For example, if the agent visits 700 frames, the accumulated reward is $1000 - 0.1 \times 700 = 930$.
4. **Initial State**: The car starts at the center of a randomly generated track.
5. **Episode Termination**: One episode finishes when all of the tiles are visited or when the episode reaches a certain time limit. In this project, the episode termination will be via the default time limit (15 seconds).

3.3.3. Neural Networks

Neural networks are crucial when it comes to solving RL problems, particularly when dealing with high-dimensional continuous state and action spaces, such as the Box2D car racing environment. Neural networks serve as function approximators, meaning they can approximate critical functions like the quality function by minimizing the difference between the predicted and true value of the function. This is done by continuously updating their weights through an optimizer. The PPO algorithm uses gradient information to iteratively update model parameters through the Adam (Adaptive Moment Estimation) optimizer. In RL, neural networks learn to approximate the target function through a combination of exploration, experience and feedback from the environment. Therefore, adjusting the reward function of the environment will influence how the neural network is trained.

Neural networks can learn to generalize across similar states and actions, enabling the agent to make reasonable decisions even in unseen situations. This ability to generalize based on unseen data is particularly important in the car racing environment, as the racetrack is randomly generated every episode. Thereby, choosing the right neural network is a big step in solving this RL problem. For this project, convolutional neural networks (CNNs) are well-suited for the car racing environment because they can effectively process and learn from image-based state representations, which is key given that the input is a 96x96 pixel image. Just like any artificial neural network, a CNN has an input layer, a hidden layer, and an output layer. However, CNNs differ as they have convolutional layers within the hidden layers. The figure below illustrates how a CNN can process the input image through a series of convolution and pooling layers:

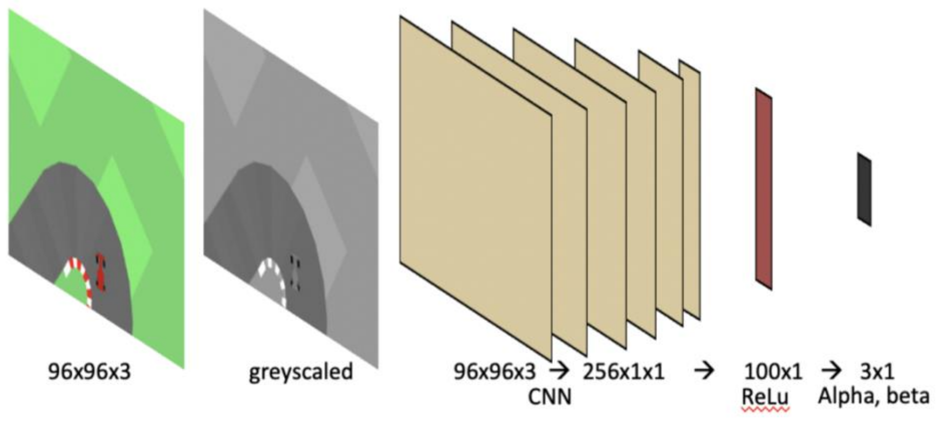


Figure 4: State Processing from Pixel Image to Linear Vector via CNN [14]

CNNs can learn hierarchical features from the raw input image, starting with low level features such as the edges and corners of the image in the initial layers, and progressively learning higher level features such as the track segments and car orientation in deeper layers. As a result, the CNN can efficiently process the RGB input image by learning the relevant features, allowing the agent to make better decisions based on visual input.

Chapter 4: Model Training and Evaluation

4.1. Introduction

This chapter presents the process of training and evaluating RL models for autonomous vehicles in the Box2D car racing environment. An overview of the model training process will be provided, highlighting challenges the RL agent incurs when navigating procedurally generated tracks. Additionally, the chapter will delve into the evaluation methodologies employed to assess and compare the performance of all the trained models.

4.2. Model Training and Analysis

To train the model, the necessary libraries had to be imported including Gym for creating the environment, and Stable-Baselines3 for the PPO algorithm. The environment was set up by wrapping the car racing environment in a dummy vector environment to ensure compatibility with Stable-Baselines3. A python script (available in the appendices) was then written to define the model, its parameters, the CNN, and save the model, and its data for later use. The first model was trained for 20,000 timesteps as this is the amount taken to train models in the Cart Pole environment, a simpler environment cited in the Gyms documentation [10]. The figure below illustrates the first training verdict:

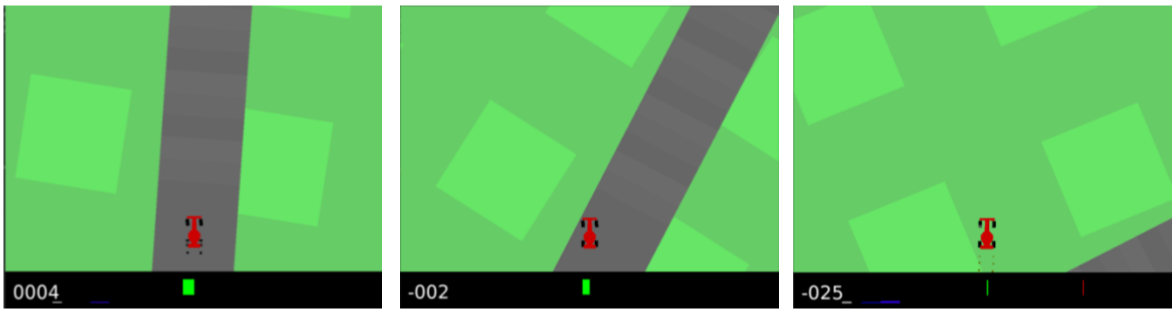


Figure 5: Different Frames for the First Trained Model

It was clear that the complex task of navigating a racetrack was going to require more training and tuning before the car can complete the track. The accumulated reward, as seen in the bottom left of the figure, decreases as the car goes off the track before reaching a final reward value of -182. It should be noted that in the car racing environment, the maximum attainable reward is around 900 and anything above 700 is considered desirable according to the environment documentation.

At this point, it was unclear whether the agent is learning or not, or whether there was any issue with the training set up. To examine this, the next model was trained for 100,000 timesteps and instead of relying solely on visual analysis, a callback function was written to store the episodic reward and the reward standard deviation every 10,000 timesteps during training. The graph below highlights the training improvement over time, with error bars representing the standard deviation:

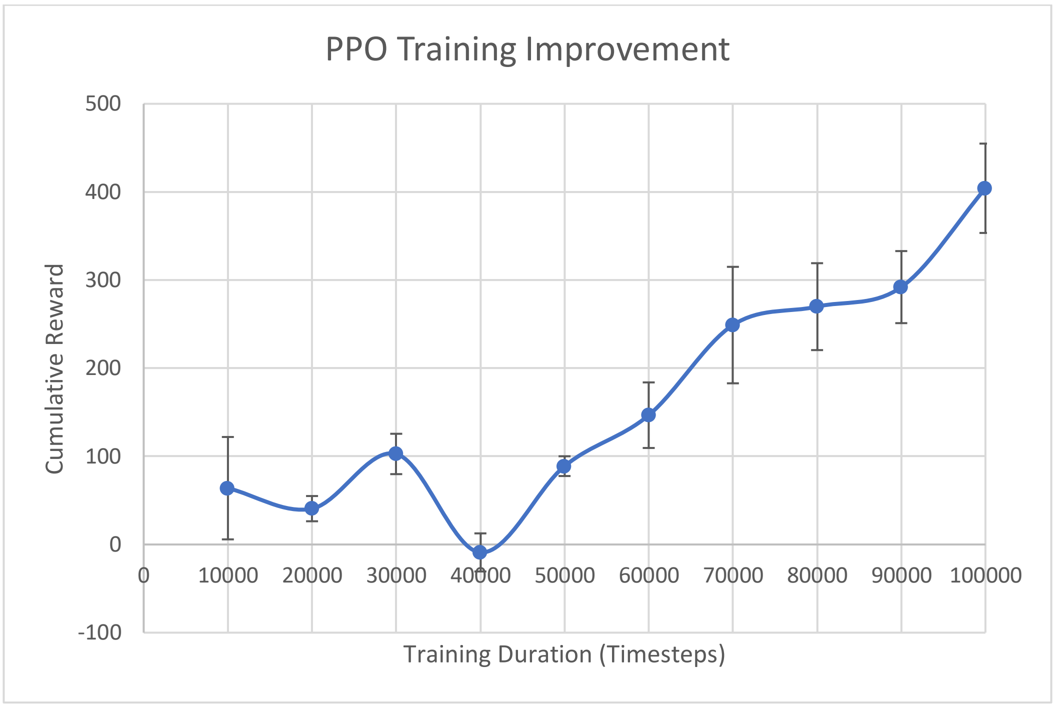
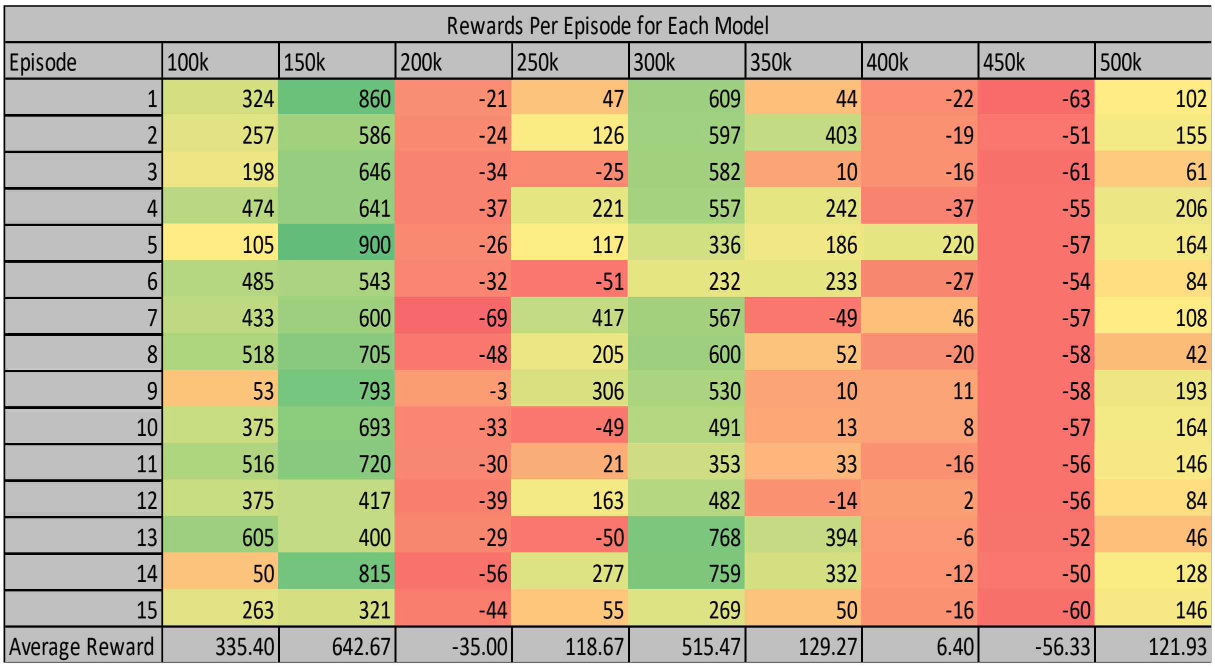


Figure 6: Average Reward Graph for 100,000 Timestep PPO Model

In the early stages of training (timesteps 0-30,000), the average episode reward fluctuates, indicating that the agent is still exploring the environment. The standard deviation also fluctuates from high to low, showing a high level of variability in the rewards received in that duration. Between timesteps 30,000-70,000, the average reward starts to increase despite the dip in 40,000 which is likely due to the agent exploring new parts of the state-action space. This is likely to be a common challenge given a new track is generated every episode. From timestep 70,000 onwards, the average reward shows a consistent upwards trend, indicating that the agent has learned a more effective policy than that of previous timesteps. Moreover, the standard deviation starts to decrease, suggesting that the agent's performance is becoming more stable and consistent across episodes. Overall, the results demonstrate that the PPO agent is capable of learning a successful policy for the car racing environment. The variability of the reward standard deviation, however, suggests that there is still room for improvement when it comes to the consistency of the model.

To further explore this approach and gain a comprehensive understanding of the agent's performance, several models were trained with varying timesteps and tested across 15 episodes, and the rewards per episode were logged. Testing each model across 15 episodes reduces performance variability and provides a measure for the consistency of each model hence effective evaluation. Based on the previous results, a reasonable hypothesis to make is that the performance of the agent, measured by the average reward, improves as the number of timesteps during training increases. The reward per episode data, however, show that this is not the case. Conditional formatting was applied to the data for easy comparison, with high rewards labelled with the color green, intermediate rewards with yellow, and low rewards with red:

Table 1: Conditional Formatting of the Reward Per Episode Data



Rewards Per Episode for Each Model									
Episode	100k	150k	200k	250k	300k	350k	400k	450k	500k
1	324	860	-21	47	609	44	-22	-63	102
2	257	586	-24	126	597	403	-19	-51	155
3	198	646	-34	-25	582	10	-16	-61	61
4	474	641	-37	221	557	242	-37	-55	206
5	105	900	-26	117	336	186	220	-57	164
6	485	543	-32	-51	232	233	-27	-54	84
7	433	600	-69	417	567	-49	46	-57	108
8	518	705	-48	205	600	52	-20	-58	42
9	53	793	-3	306	530	10	11	-58	193
10	375	693	-33	-49	491	13	8	-57	164
11	516	720	-30	21	353	33	-16	-56	146
12	375	417	-39	163	482	-14	2	-56	84
13	605	400	-29	-50	768	394	-6	-52	46
14	50	815	-56	277	759	332	-12	-50	128
15	263	321	-44	55	269	50	-16	-60	146
Average Reward	335.40	642.67	-35.00	118.67	515.47	129.27	6.40	-56.33	121.93

It is important to recall that any reward over 700 in the car racing environment is considered desirable due to the complexity of the task. From the data, it is apparent that the 150k and the 300k models generally achieve higher rewards compared to the other models. The 150k model, in particular, performs well across most episodes, with an average of 642, and instances where the reward is above 800. Therefore, the 150k model can be classed as the best model in this case. The 200k and 450k models exhibit very poor performance, with negative reward values for all episodes. This indicates that these models were not able to learn an effective policy needed to exploit the car racing environment. The performance of the 250k, 350k, and the 500k varies across episodes, with some episodes showing relatively high rewards and other episodes show low rewards. This suggests that these models have learned some successful strategies that work in some tracks, but do not work in other tracks. This high variability is seen with other models as well, which leads to the belief that the PPO models need hyperparameter tuning, something that will be explored in the next chapter.

While it was expected that more training would lead to better performance, the results above show that this is not the case. One likely reason for this is overfitting – as the number of training timesteps increases, there is a risk that the model starts to overfit the training data. This means that the model learns specific variations of the training environment, but fails to generalize to new situations, specifically new racetracks in this case. This leads to poorer performance across different episodes.

Another probable reason for sufficiently trained models to not perform well is instability in the learning process, which stems from the agent not being able to learn the optimal policy due to extended training times. One can assess the stability of the learning process for an RL agent through a policy gradient loss graph. A policy loss graph plots the aforementioned surrogate objective function (loss function) against the number of training timesteps [11], providing insights into the agent's learning process. In this experiment, the policy loss data of the best model (150k) and the worst model (450k) were logged using tensorboard, and their graphs are seen below:

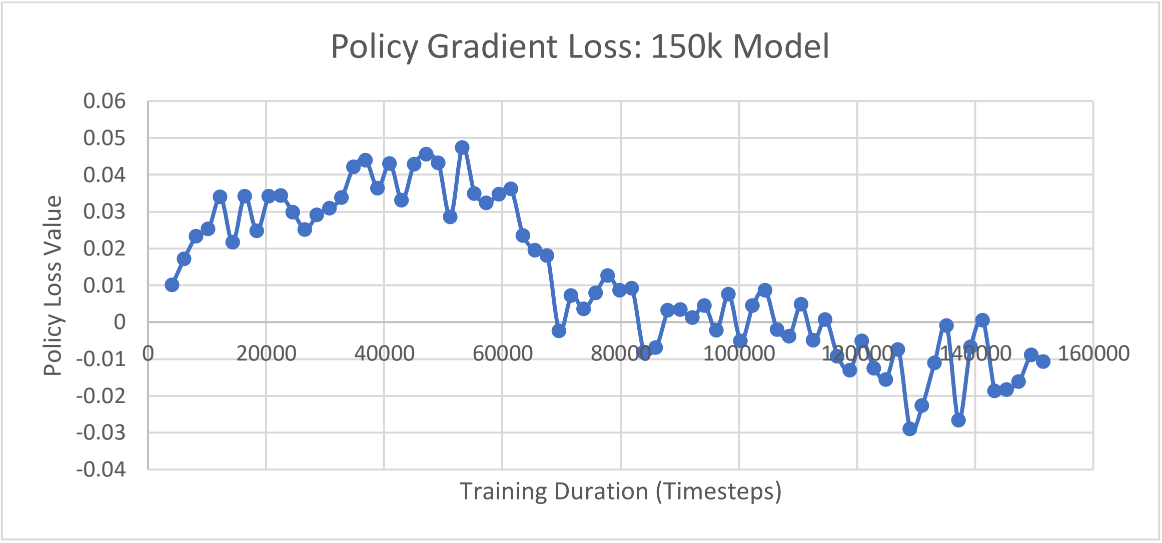


Figure 7: Policy Gradient Loss Graph of 150k PPO Model

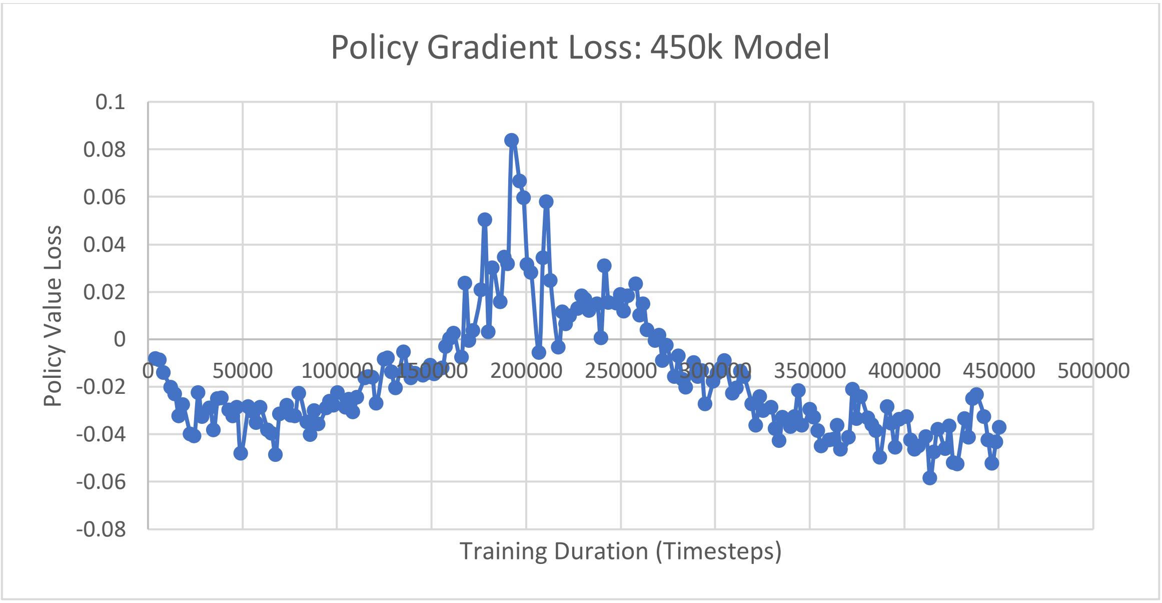


Figure 8: Policy Gradient Loss Graph of 450k PPO Model

The policy gradient graph for the 150k model displays a downward trend in loss values as time passes. Lower loss values indicate that the model's predictions closely match the actual outcomes from the environment, which is a sign of good performance. This suggests that the model is able to learn and improve its policy as the training progresses. There are a few spikes in the loss values but overall, the loss values are decreasing gradually indicating a stable learning process hence the model averaging high rewards. On the other hand, the 450k model, which had the lowest average reward, exhibits erratic behavior with fluctuations and spikes that show no clear trend. Loss values initially decrease to around -0.6 then rise to 0.8 before falling down again to -0.6. This suggests that the model is struggling to learn an effective policy for the car racing environment. The lack of a clear trend in loss values means the model is not improving as training advances, which is also a sign that the model is overfitting to the training data, and is not able to generalize to new situations such as new tracks or new turns in the environment.

To conclude this section of the chapter, the last few experiments show that there is not a clear relationship between the amount of training an RL agent receives and the model's performance. It is therefore essential to consider factors such as overfitting, learning stability and loss function values when interpreting model results in RL.

4.3. Further Evaluation

Using different evaluation metrics is advantageous as it may help capture various aspects of the agent's performance. In this section of the chapter, trained models will be evaluated using different evaluation techniques in order to gain a more comprehensive understanding of the strengths and weaknesses of each model.

4.3.1. Visual Analysis

After training all the previous models, it became clear that some models perform better on some tracks than other models. Therefore, one track was chosen at random to test and compare the best three models. The figure below shows the track:

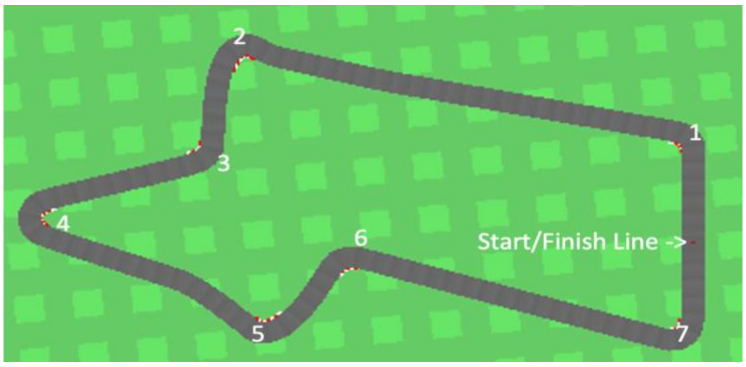


Figure 9: Race Track with Labelled Checkpoints

To carefully analyze the performance of each model, multiple episodes were simulated on the track above. Here is a summary of the performance of each model:

150k Model: The 150k model averaged the highest reward (over 500) between the three, moving very quick on the straights and takes relatively quick sharp turns until it reaches turn 4 where it often goes off the track and comes back again, losing some reward in the process. The car agent typically reaches the straight after checkpoint 6 on the track before the episode terminates due to the set time limit.

300k Model: The 300k Model averages the second highest reward of around 400-500, with similar speed to the previous model. However, there are more instances in which car agent goes off the track, particularly in turns 2 and 4. In some cases, the agent goes off track and is not able to recover, reducing the average reward to below 400. The agent is usually between checkpoints 5 and 6 before the episode terminates.

100k Model: The 100k model averaged rewards between 300-400. Unlike the other two models, the 100k model moves at a relatively steady speed in the straights and takes much longer times to turn which reduces the reward as the car is not able to cover most of the track within the specified time limit. However, the 100k model does not go off track as much, which is an advantage it has over the other models.

Visual analysis has proved to be very valuable when evaluating RL models because one can identify distinct characteristics for each model which are not evident through quantitative metrics such as episodic rewards. For example, if the penalty for going off-track was higher, then the average reward of the 100k model would be higher than that of other models. It was also seen that all models struggled with U-turns so that is certainly an area for further improvement.

4.3.2. Model Consistency Using Evaluate Policy

The evaluate policy method is a function in the Stable-baselines3 library that assesses the performance of the trained models by running a specified number of episodes and returning the average reward and standard deviation without the need to manually record any data. Therefore, one can use the evaluate policy method to examine the consistency of each model by analyzing the reward standard deviation across the specified number of episodes. This is because the standard deviation provides insight into the variability of the rewards for each model, thus a low standard deviation would indicate a more consistent model. To ensure unambiguous results, the number of evaluation episodes for this experiment is 20. The python script for this experiment is available in the appendices. The results are shown in the bar chart below:

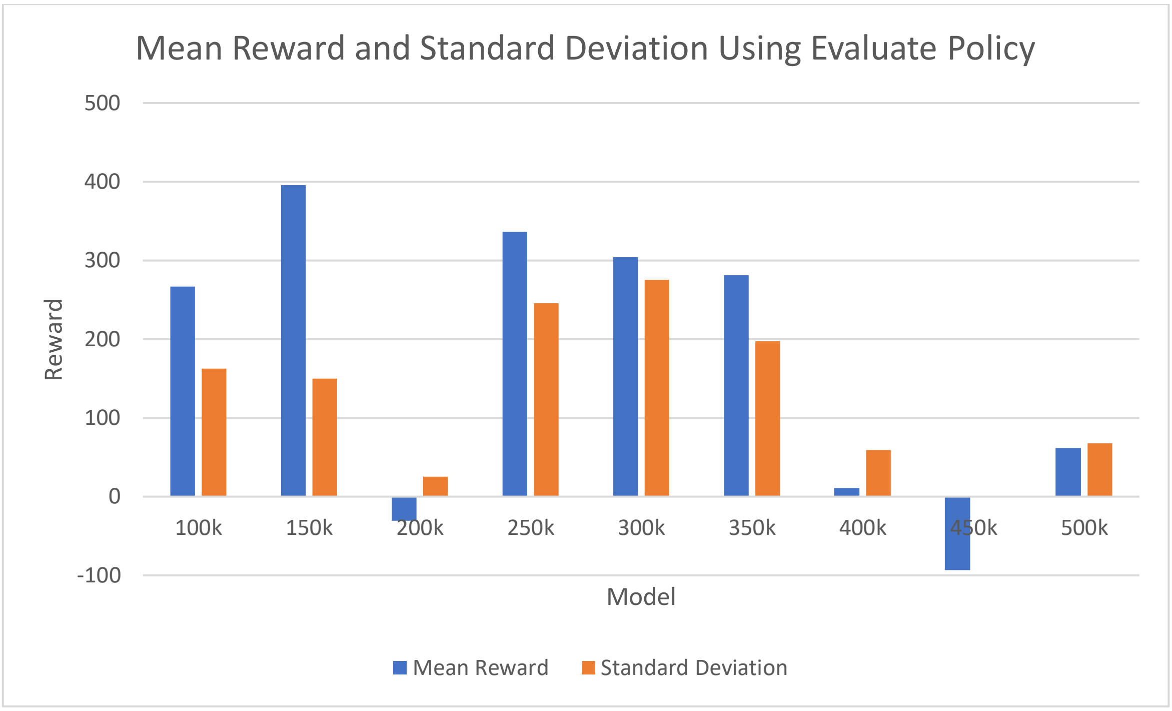


Figure 10: Bar Chart of Mean Reward and Standard Deviation Across 20 Evaluation Episodes

The bar chart reveals a correlation between the magnitude of the reward and the corresponding standard deviation for each model. It is observed that models with greater rewards tend to exhibit higher standard deviations and are thus less consistent. This relationship is substantiated by the relatively low standard deviations associated with the poor performing models – the 400k, 450k, and 500k models. Once again, the 150k model proves to be the best model in this experiment with the highest reward of around 400, and a relatively low standard deviation when compared to the superior models. This means that the 150k model is more consistent and thus 150,000 timesteps is the optimal training duration. Evaluating models based on mean reward and standard deviation provides a more thorough overview of the model performance.

When comparing these results to the previous experiment, there were some inconsistencies with the performance of some models. For example, the 250k model shows notable improvement in performance from the first experiment, with the second highest mean reward of 336. It is also seen that the rewards for most models were slightly lower using the evaluate policy method. These differences in reward are likely due to the fact that the evaluate policy method uses different initial states depending on the random seed written in the Stable-Baselines3 documentation. This means that the car agent might not begin from the start line in figure 9, which can result in different rewards for each model. Despite these differences, both experiments validate that model performance does not increase with additional training.

Chapter 5: Hyperparameter Optimization

5.1. Introduction

This chapter will delve into the importance of hyperparameter optimization on machine learning models, with a particular focus on the parameters of the PPO algorithm. The chapter will explore grid-based approaches to fine-tune the model and discuss their advantages and disadvantages. Thereby, this chapter aims to provide a comprehensive overview of hyperparameter tuning in the context of the PPO algorithm for the car racing environment.

5.2. Proximal Policy Optimization Hyperparameters

In machine learning, hyperparameters are model parameters that have a direct influence on the performance, speed, and convergence of the model. In the context of RL, the complexity and stochastic nature of RL environments can make it challenging to find a single set of hyperparameters that are optimal for all stages of learning. Therefore, it is crucial to understand the key hyperparameters of the PPO algorithm to realize their potential impact on the learning process. Below is a description of the key hyperparameters of the PPO algorithm:

1. **Learning Rate:** (Default 0.0003)

The learning rate, also known as the Adam Stepsize [9], is a critical hyperparameter that controls the step size taken during the optimization process when updating the model's parameters. In this case, the learning rate is updating the weights of the CNN. Therefore, the learning rate affects how quickly the agent's policy is updated. A small learning rate means the policy will be updated conservatively, which can lead to stable learning, but may slow down the convergence – the time needed to reach an optimal solution. A large learning rate can introduce instability in training, causing the PPO algorithm to overshoot the optimal solution.

2. **Number of Steps:** (Default 2048)

The number of steps, also known as the Horizon [9], is the number of timesteps collected between each policy update. In other words, it is the number of steps taken in the car racing environment before the car agent updates their policy. A large value of the number of steps means more timesteps taken before each policy update which can improve the accuracy of the model estimations but can also increase computational complexity.

3. **Number of Epochs and Batch Size:** (Default 10 and 64)

An epoch represents the process of passing the entire dataset in the neural network once, both forward and backward. The dataset is typically divided into smaller parts called batches. This gives rise to another hyperparameter of the PPO algorithm called Batch Size, and it is the number of training data samples in a single batch. For example, if the training dataset consists of 2000 training examples and was divided into batches of 500, it will take 4 training iterations to complete 1 epoch. A high number of epochs means that the agent will have more opportunities to learn and refine their understanding of the environment. This can lead to better performance and thus the number of epochs is one of the most critical parameters of the PPO algorithm. The disadvantage of having a higher number of epochs is the extended training times. Similarly, although a high batch size could lead to more accurate updates, there is a cost of increased computational complexity.

4. **Gamma the Discount Factor (γ):** (Default 0.99)

As mentioned previously, γ is a real number between 0 and 1 called the discount factor that determines the weight of future rewards with respect to immediate rewards. When gamma is closer to 1, the agent places higher emphasis on future rewards over short-term rewards. This is beneficial as it encourages the agent to plan for the long term and make decisions that maximize the cumulative reward over time.

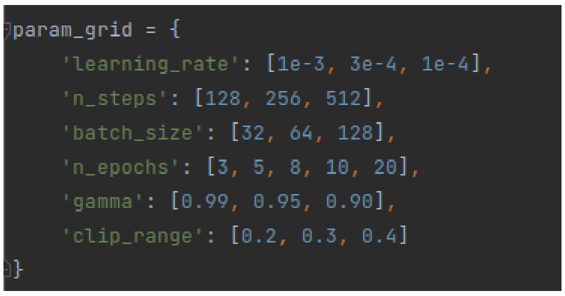
5. **Clip Range:** (Default 0.20)

The Clip Range, often donated with epsilon ϵ, is a parameter that controls the magnitude of policy updates during training [12]. The purpose of clipping is to ensure that the new policy does not deviate far from the old policy, which helps avoid excessively large updates that can hinder the learning process. To handle optimization, the PPO algorithm computes the ratio of probabilities for the new policy and the old policy. This ratio is clipped within the range $[1-\epsilon, 1+\epsilon]$. By doing so, the PPO algorithm sets a limit on how much the policy can change in a single update step [12]. Therefore, choosing an appropriate value for the clip range is essential for balancing the exploration-exploitation trade-off.

5.3. Tuning the Model

The goal of this section of the chapter is to conduct the process of finding the optimal configuration of hyperparameters that will achieve the best performance of the model. To tune the model, one can approach the problem in various ways. Manual search is when the user manually adjusts hyperparameters based on intuition and experience, which be very time consuming. Another popular method is a random search, which involves sampling random combinations of hyperparameter values within a predefined range. This method can be more effective when the environment has a continuous action space (i.e., car racing environment) because it can help identify promising regions in the search space more quickly, without relying on intuition or experience.

In this project, a more systematic approach to hyperparameter tuning was employed by utilizing a grid-based search method (GridSearchCV [13]). This involves exhaustively trying all possible combinations of the hyperparameter values within a predefined search space, and identifying the combination of hyperparameters that yield the best result. The main issue with this approach is that it can be computationally intensive, which was the case as model training was performed in parallel to reduce training time. The search space is defined below, and values chosen for each parameter were approximations from relevant literature [9] [12]:



```
param_grid = {
    'learning_rate': [1e-3, 3e-4, 1e-4],
    'n_steps': [128, 256, 512],
    'batch_size': [32, 64, 128],
    'n_epochs': [3, 5, 8, 10, 20],
    'gamma': [0.99, 0.95, 0.90],
    'clip_range': [0.2, 0.3, 0.4]
}
```

Figure 11: PPO Hyperparameter Grid

To minimize the training duration, all parameters were constrained to three values, except for the number of epochs, given its significant impact on model performance. As determined in the preceding chapter, the optimal training period for the car racing environment is 150,000 timesteps. Consequently, the best hyperparameter combinations were trained for that period across 15 episodes. The best tuned model averaged a reward of 644 with the following hyperparameters:

Table 2: Hyperparameter Optimization Results

Hyperparameters	Tuned Values
Learning Rate	1e-4
Number of Steps	512
Batch Size	128
Number of Epochs	20
Gamma Discount Factor	0.99
Clip Range	0.2

The tuned gamma and clip range values matches the PPO defaults, suggesting that some PPO parameters require minimal tuning. The tuned number of epochs was the highest value from the search space. To experiment, all the tuned parameters were kept the same and the number of epochs was increased to 50, 100, 150, 200. Despite the prolonged training times, the model with 100 epochs received the highest average reward of around 750, which is considered desirable in the car racing environment. This proved that in order to carefully tune an RL model, a systematic approach of grid-based methods followed by further experimentation can help identify the best hyperparameter combinations that result in better performance.

5.4. 150k Model vs Optimized Model

A test script was written to evaluate the tuned model with 100 number of epochs against the best 150k model from the previous chapter. The results are as follows:

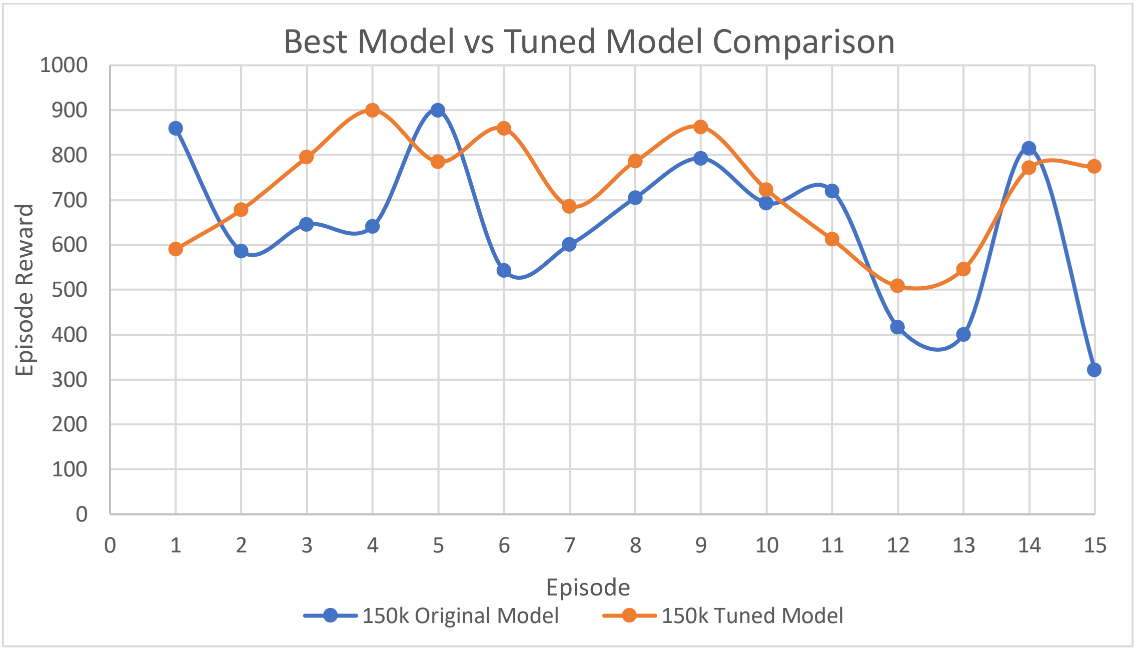


Figure 12: Comparison between Original and Tuned Model Across 15 Episodes

In 11 out of the 15 episodes, the tuned model yields higher reward than the original 150k model, indicating the benefits of hyperparameter tuning. The tuning process has allowed the optimized model to better adapt to the car racing environment, leading to improved performance in most episodes. The difference between the average reward of both models was 82, which is quite significant. Episodes 12 and 13 show a dip in performance but that is attributed to the tracks having more than one U-turn, making it harder for the agent to score higher rewards. Visually, both models perform the similarly on the straights of the track but the tuned model takes sharper turns as seen below. There are few instances when the tuned car agent goes off track but manages to recover every time, which was not the case with the original model. See frame below:

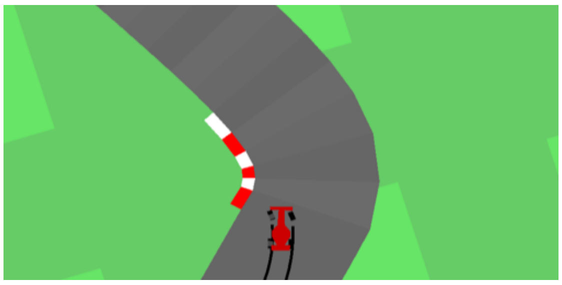


Figure 13: Tuned Car Agent Taking Sharp Turns

Chapter 6: Reward Modification

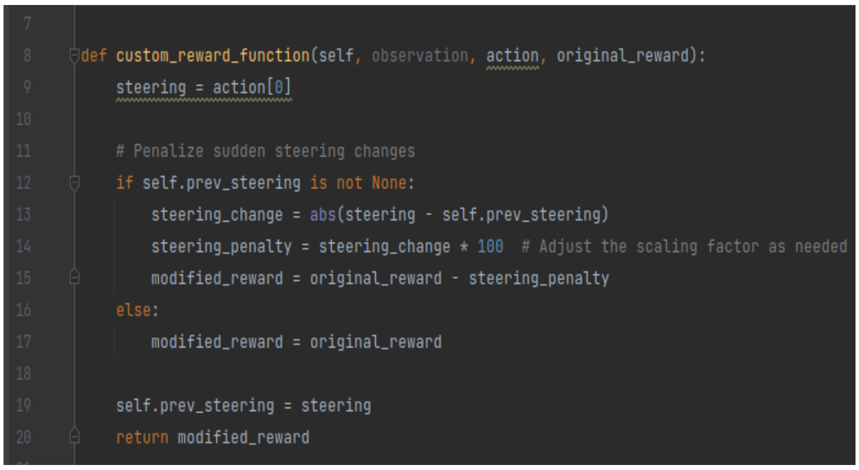
6.1. Introduction

The primary focus of this chapter is to understand the effects of changing the reward function on the agent's policy and the model performance. The chapter will delve into identifying and implementing new reward function to the car racing environment, followed by an evaluation of each model. By systematically investigating the effects of changing the reward function, one could be able to optimize models more effectively for application of autonomous driving.

6.2. Reward Function Modification

It is important to recall the default reward function of the car racing environment. As stated in chapter 3, the default reward function is: -0.1 for every frame and +1000/N for every track tile visited, where N is the total number of track tiles visited. For example, if the agent visits 700 frames, the accumulated reward is $1000 - 0.1 \times 700 = 930$. To adjust the reward function, it is essential to first identify a specific aspect of the agent's behaviour that necessitates improvement. The next step is formulating a modified reward structure that enhances that aspect. The implementation of the new reward function entails writing a function featuring the revised reward structure, which overrides the environment's step function. The step function is responsible for executing the agent's actions and updating the state of the environment. By overriding it, one can modify how the agent is rewarded for its actions and investigate any change in agent behaviour.

A recurring issue observed across all trained models thus far is the tendency of the car agent to deviate from the track during turns. Therefore, a custom reward function was created to promote smoother driving and keep the agent on the track by penalizing sudden steering changes. A snippet of the reward function is below:



```python
def custom_reward_function(self, observation, action, original_reward):
    steering = action[0]

    # Penalize sudden steering changes
    if self.prev_steering is not None:
        steering_change = abs(steering - self.prev_steering)
        steering_penalty = steering_change * 100  # Adjust the scaling factor as needed
        modified_reward = original_reward - steering_penalty
    else:
        modified_reward = original_reward

    self.prev_steering = steering
    return modified_reward
```

Figure 14: Custom Reward Function to Prevent Sudden Steering Changes

In this function, the steering value is extracted from the agent. Line 12 checks to see if there is a previous steering value. If there is, the steering change variable stores the absolute difference between current steering value and the previous one. This is multiplied with a scaling factor of 100 to obtain the steering penalty which is reduced from the original reward to compute the modified reward. The modified reward function therefore encourages the agent to learn a policy that avoids abrupt steering and promotes better adherence to the track. The model with the modified reward function was trained with the tuned hyperparameters for the optimal training duration of 150,000 timesteps.

6.3. Model Evaluation

The trained model incorporating the modified reward function was evaluated across 15 episodes, yielding unexpected results. Despite undergoing parameter tuning and training for 150,000 timesteps, the model attained a relatively low average cumulative reward of 201. However, upon simulating the model, it was observed that unlike previous models, this model consistently remained on track. The following figure shows multiple frames in which the agent completes a U-turn smoothly, something that previous models struggled with:

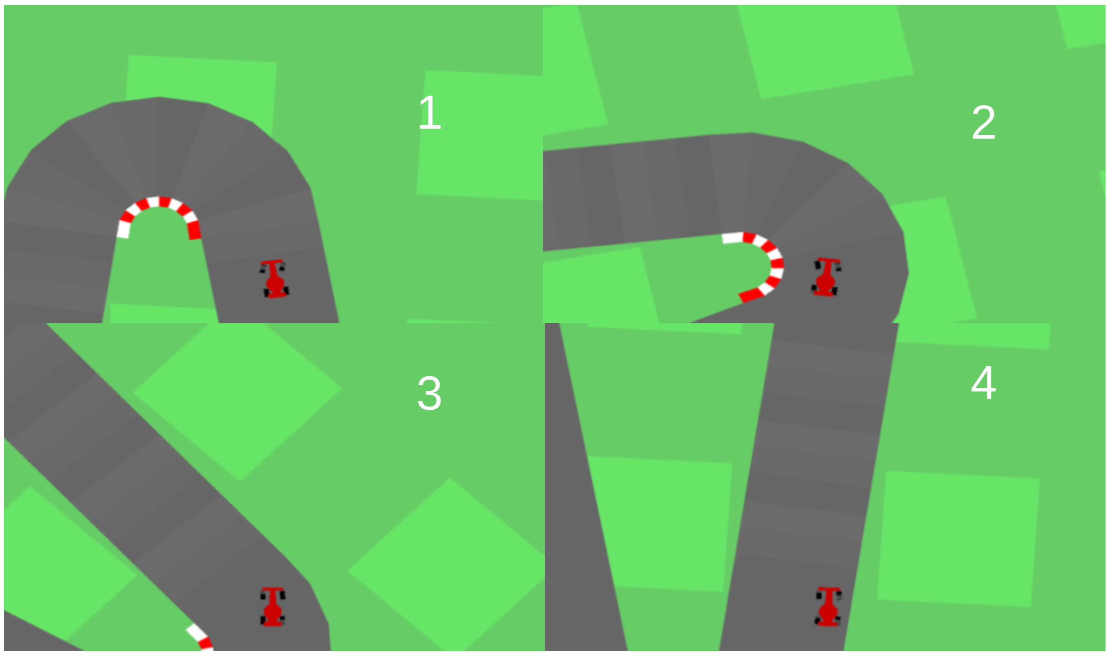


Figure 15: Car Agent with Modified Reward Function Completing U-turn

The car agent manages to complete every U-turn smoothly and drives slowly without going off track. It is therefore safe to say that the modification of the reward function had a direct influence in the agent's policy and effectively promoted adherence to the track. The reason the model averages a low reward is because the car agent does not move very quick at the straights of the track and thus does not visit as many tiles of the track in one episode. To counter this, the scaling factor was reduced to 50 and the model was retrained with the same pre-existing conditions. The difference in results was significant enough to warrant a comparison with the tuned model from the previous chapter. The results are as follows:

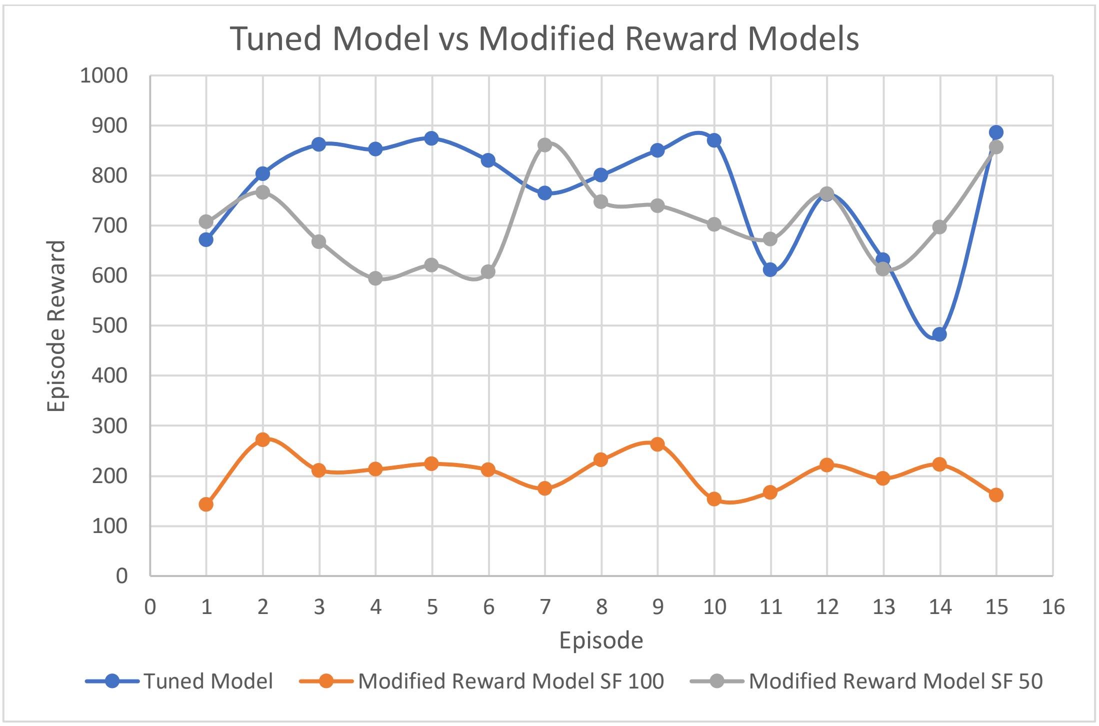


Figure 16: Comparison of New Reward Models with Different Scaling Factors Against Tuned Model

It is apparent that the reduction in the scaling factor to 50 had a huge impact on the model performance, with an episodic reward average of around 700. This makes it comparable to the best tuned model from the previous chapter. From a visual observation, a noticeable distinction between the two models emerges: the model employing a scaling factor of 50 demonstrates tendency to maintain its position on the track more effectively, while the tuned model exhibits greater speed, subsequently traversing a larger number of track tiles and obtaining slightly higher reward. Although greater speeds help achieve higher rewards, it is essential to consider the trade-off between performance and safety when evaluating both models, especially when considering real-world applications such as autonomous vehicles.

This experiment affirms that the choice of reward function can significantly influence the agent’s policy, ultimately affecting the overall model performance and behaviour of the car agent. Consequently, creating a proper reward function is a critical aspect of developing more effective deep reinforcement learning models for autonomous vehicles.

Chapter 7: Discussions and Conclusions

7.1. Project Achievements

This project has resulted in several notable achievements which demonstrate the potential of deep reinforcement learning in the context of autonomous driving environments. The achievements are below:

1. Leveraged proximal policy optimization algorithms to design and implement a deep RL model, specifically tailored to the autonomous driving environment.
2. Conducted extensive model training and evaluation to identify the best model and the optimal training time.
3. Performed systematic hyperparameter tuning using grid-based methods to identify the most favourable parameter configuration that enhances performance.
4. Successfully modified the reward function to influence agent policy behaviour and maintain admirable performance.
5. Compiled a comprehensive analysis of experimental results, showcasing the potential of the trained models in real-world autonomous driving applications.
6. Derived conclusions from the project results, emphasizing the importance of carefully designing reward functions and tuning hyperparameters to implement useful models.

7.2. Discussion

The project began with the design of deep reinforcement learning models by utilizing the PPO algorithm and a convolutional neural network. Several models were trained with varying timesteps, and their performance was assessed using different evaluation metrics. The 150k model performed the best in all experiments, with the highest reward and a relatively low standard deviation, thus better consistency. This indicated that 150,000 timesteps is the optimal training time for PPO models in the autonomous driving environment. The combination of visual and numeric evaluation techniques provided valuable insights into the strengths and weaknesses of each model, and revealed that there is no clear relationship between training duration and performance of autonomous driving models.

The next chapter introduced the key hyperparameters of the PPO algorithm and explored various tuning methods aimed at enhancing performance. A systematic approach using grid-based search was employed to identify the best parameter configuration with 100 epochs. The final tuned model was trained for 150,000 timesteps, averaging a high reward of 750 per episode. Visual analysis displayed the tuned car agent taking sharper turns and recovering more quickly when going off track.

Therefore, the tuning process allowed the model to better adapt to the autonomous driving environment.

Modifying the reward function was the most challenging part of the project. The default reward function was modified to address the car agent's tendency to deviate from the track by penalizing sudden steering changes. The model was trained for the optimal training time with the tuned parameter configuration. The steering penalty proved to effective in keeping the agent on the track but hindered model performance. Reducing the steering penalty by changing the scaling factor improved the results significantly, ultimately revealing that the environment's reward structure has a direct impact on policy behaviour. The trade-off between safety and performance was met, indicating that one can optimize RL models for autonomous driving applications through reward manipulation.

7.3. Conclusion

The aim of this project was to develop deep reinforcement learning models using proximal policy optimization approaches for autonomous vehicles, and to investigate the effect of varying the reward function on model performance. This aim was achieved by conducting a thorough investigation on the key aspects of reinforcement learning, including model training, hyperparameter tuning, and reward function modification.

Throughout the course of the project, several key findings emerged, and the conclusion from every chapter helped improve the model for the next chapter. This approach helped contribute to the broader goal of developing optimized reinforcement learning models for autonomous vehicles. The main project limitation is that this project focused on the car racing environment, primarily tackling the problem of lane-keeping. According to relevant literature, however, Autonomous driving offers a range of perception level tasks that need to be solved as well. These include motion planning, vehicle localization, automated parking, traffic sign detection and so on. These are avenues of future research that can be explored further.

Bibliography

[1] Y. S. S. Z. Le Lyu, "The Advance of Reinforcement Learning and Deep Reinforcement Learning," *2022 IEEE International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA),* vol. 22, no. 1, pp. 644-648, 2022.

[2] R. Bellman, "Dynamic programming and Lagrange multipliers," *Proceedings of the National Academy of Sciences,* vol. 42, no. 10, pp. 767-769, 1956.

[3] A. G. Barto, "Recent Advances in Hierarchical Reinforcement Learning," *Klewer Academic Publishers,* vol. 13, no. 1, pp. 41-77, 2003.

[4] A. H. B. D. J. P. J. Scott R. Granter, "AlphaGo, Deep Learning, and the Future of the Human Microscopist," *Archives of Pathology & Laboratory Medicine,* vol. 141, no. 5, pp. 619-621, 2017.

[5] A. G. B. Richard S. Sutton, in *Reinforcement Learning: An Introduction*, Cambridge, Massachusetts, The MIT Press, 2018, pp. 47-54.

[6] Y. Li, "DEEP REINFORCEMENT LEARNING: AN OVERVIEW," *arXiv,* vol. 6, pp. 7-8, 2018.

[7] B. R. Kiran, "Deep Reinforcement Learning for Autonomous Driving: A survey," *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS,* vol. 23, no. 6, pp. 4909-4925, 2022.

[8] Y. Gu, "Proximal Policy Optimization With Policy Feedback," *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS: SYSTEMS,* vol. 52, no. 7, pp. 4600-4610, 2022.

[9] F. W. P. D. A. R. O. K. John Schulman, "Proximal Policy Optimization Algorithms," *arXiv,* vol. 2, 2017.

[10] OpenAI, "Gym Documentation: Car Racing," 2022. [Online]. Available: https://www.gymlibrary.dev/environments/box2d/car_racing/. [Accessed 8 April 2023].

[11] J. J. Garau-Luis, "Evolving Generalizable Actor-Critic Algorithms," *arXiv,* vol. 2, pp. 4-8, 2022.

[12] D. L. S. Mo´nika Farsang, "Decaying Clipping Range in Proximal Policy Optimization," *arXiv,* vol. 3, pp. 1-5, 2021.

[13] F. Pedregosa, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research,* vol. 12, pp. 2825-2830, 2011.

[14] J. S. &. V. Valverde, Artist, *Comparing Reinforcement Learning Techniques on carRacing-v0 Environment.* [Art]. Stanford University, 2019.

[15] *A Taxonomy of RL Algorithms.* [Art]. OpenAI Inc., 2018.

Appendices

Appendix 1: First Python Script

```
import gym
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv
import os

env = gym.make("CarRacing-v0")

#  Training our Model
env = DummyVecEnv([lambda: env])
log_path = os.path.join('Training', 'Car Logs')
model = PPO('CnnPolicy', env, verbose=1, tensorboard_log=log_path)
model.learn(total_timesteps=20000)

#  Saving our model
ppo_path = os.path.join('Training', 'Car Saved Models', 'PPO1_First_Model_20k')
model.save(ppo_path)

# Testing our model
episodes = 15

for episodes in range(1, episodes+1):
    obs = env.reset()
    done = False
    score = 0
    while not done:
        env.render()
        action, _ = model.predict(obs)
        obs, reward, done, info = env.step(action)
        score += reward  # accumulating our reward
    print("Episode:{} Score: {}".format(episodes, score))
env.close()  # close down the render frame
```

Appendix 2: Evaluate Policy Python Script and Data

```
from stable_baselines3 import PPO
from stable_baselines3.common.evaluation import evaluate_policy
import gym
import os

ppo_path_100k = os.path.join('Training', 'Car Saved Models', 'PPO_100k_Driving_Model')
ppo_path_150k = os.path.join('Training', 'Car Saved Models', 'PPO_150k_Driving_Model')
ppo_path_200k = os.path.join('Training', 'Car Saved Models', 'PPO_200k_Driving_Model')
ppo_path_250k = os.path.join('Training', 'Car Saved Models', 'PPO_250k_Driving_Model')
ppo_path_300k = os.path.join('Training', 'Car Saved Models', 'PPO_300k_Driving_Model')
ppo_path_350k = os.path.join('Training', 'Car Saved Models', 'PPO_350k_Driving_Model')
ppo_path_400k = os.path.join('Training', 'Car Saved Models', 'PPO_400k_Driving_Model')
ppo_path_450k = os.path.join('Training', 'Car Saved Models', 'PPO_450k_Driving_Model')
ppo_path_500k = os.path.join('Training', 'Car Saved Models', 'PPO_500k_Driving_Model')

model_100k = PPO.load(ppo_path_100k)
model_150k = PPO.load(ppo_path_150k)
model_200k = PPO.load(ppo_path_200k)
model_250k = PPO.load(ppo_path_250k)
model_300k = PPO.load(ppo_path_300k)
model_350k = PPO.load(ppo_path_350k)
model_400k = PPO.load(ppo_path_400k)
model_450k = PPO.load(ppo_path_450k)
model_500k = PPO.load(ppo_path_500k)

env = gym.make("CarRacing-v0")
n_eval_episodes = 20  # Number of episodes to run for evaluation

mean_reward_100k, std_reward_100k = evaluate_policy(model_100k, env,
n_eval_episodes=n_eval_episodes)
mean_reward_150k, std_reward_150k = evaluate_policy(model_150k, env,
n_eval_episodes=n_eval_episodes)
mean_reward_200k, std_reward_200k = evaluate_policy(model_200k, env,
n_eval_episodes=n_eval_episodes)
mean_reward_250k, std_reward_250k = evaluate_policy(model_250k, env,
n_eval_episodes=n_eval_episodes)
mean_reward_300k, std_reward_300k = evaluate_policy(model_300k, env,
n_eval_episodes=n_eval_episodes)
mean_reward_350k, std_reward_350k = evaluate_policy(model_350k, env,
n_eval_episodes=n_eval_episodes)
mean_reward_400k, std_reward_400k = evaluate_policy(model_400k, env,
n_eval_episodes=n_eval_episodes)
mean_reward_450k, std_reward_450k = evaluate_policy(model_450k, env,
n_eval_episodes=n_eval_episodes)
mean_reward_500k, std_reward_500k = evaluate_policy(model_500k, env,
n_eval_episodes=n_eval_episodes)

print(f"100k Model: Mean Reward = {mean_reward_100k}, Std. Dev. = {std_reward_100k}")
print(f"150k Model: Mean Reward = {mean_reward_150k}, Std. Dev. = {std_reward_150k}")
print(f"200k Model: Mean Reward = {mean_reward_200k}, Std. Dev. = {std_reward_200k}")
print(f"250k Model: Mean Reward = {mean_reward_250k}, Std. Dev. = {std_reward_250k}")
print(f"300k Model: Mean Reward = {mean_reward_300k}, Std. Dev. = {std_reward_300k}")
print(f"350k Model: Mean Reward = {mean_reward_350k}, Std. Dev. = {std_reward_350k}")
print(f"400k Model: Mean Reward = {mean_reward_400k}, Std. Dev. = {std_reward_400k}")
print(f"450k Model: Mean Reward = {mean_reward_450k}, Std. Dev. = {std_reward_450k}")
print(f"500k Model: Mean Reward = {mean_reward_500k}, Std. Dev. = {std_reward_500k}")
```

Reward and Standard Deviation Data Using Evaluate Policy:

Model	Mean Reward	Standard Deviation
100k	266.669495	162.2576428
150k	395.4978159	149.9644531
200k	-30.79121351	25.27853374
250k	336.1186772	246.008562
300k	304.5762573	275.0696133
350k	281.3502626	197.4764479
400k	10.99048075	59.27304585
450k	-93.0001572	0.384808963
500k	61.77331032	68.09338764

Appendix 3: Top Ten Models Using Grid-search Tuning

Model	Learning Rate	Number of Steps	Batch Size	Number of Epochs	Gamma	Clip Range	Average Reward
1	1.00E-04	512	128	20	0.99	0.2	644.32
2	3.00E-04	256	128	20	0.99	0.2	640.12
3	1.00E-04	512	64	20	0.99	0.2	609.89
4	3.00E-04	256	128	20	0.99	0.2	582.85
5	3.00E-04	512	128	20	0.99	0.2	580.90
6	3.00E-04	512	64	20	0.99	0.2	567.76
7	1.00E-03	256	32	20	0.99	0.2	559.24
8	1.00E-04	512	32	20	0.99	0.2	550.89
9	1.00E-04	512	32	20	0.99	0.2	549.11
10	1.00E-04	256	128	20	0.99	0.2	544.02