UNIVERSITY OF KENTUCKY
LEWIS HONORS COLLEGE

Using Deep Reinforcement Learning to Solve OpenAI's 'CarRacing' Environment

by

Jainam Dhruva

AN UNDERGRADUATE THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DISTINCTION OF UNIVERSITY HONORS

Approved by the following:

Brent Harrison, Ph.D.
Assistant Professor, Department of Computer Science

LEXINGTON, KENTUCKY
May 2023

Abstract

Reinforcement learning is a subset of machine learning which involves an agent learning from interactions with an environment. We maximize the agent's reward and through that learn good behavior for our agent. Reinforcement learning algorithms have proven effective in solving sequential decision-making problems in various fields.OpenAI Gym is a popular platform for testing and benchmarking reinforcement learning algorithms, offering different environments for testing agents. One such environment is the racing gym, which requires the agent (a car) to navigate a racing track, avoid obstacles and complete laps as quickly as possible. In this project, we try to solve OpenAI's racing gym. The text explores two popular reinforcement learning algorithms, Deep Q learning (DQN) and Proximal Policy Optimization (PPO), for solving the racing gym problem. The text compares the performance of the two algorithms, investigates various hyperparameters, and identifies the optimal hyperparameters to train the final models. We discuss some key observations that went into setting up and performing the experiments for hyperparameter tuning. The text can be used as a reference by others to solve similar environments, and save time doing so, using the described reinforcement learning methods in this project.

Contents

Chapter 1

Introduction

1.1 Overview

Reinforcement learning is a sub field of machine learning that involves learning from interactions with an environment to maximize a reward signal. This approach has become popular in recent years due to its effectiveness in solving sequential decision-making problems in a variety of fields, including robotics, game playing, and autonomous driving. Reinforcement learning algorithms have been used to train agents to navigate complex environments, learn new skills, and make decisions based on complex data inputs.

OpenAI Gym is a popular platform for testing and benchmarking reinforcement learning algorithms. It provides a wide range of environments for testing different types of agents, from simple control problems to complex decision-making tasks. One of the most popular environments in OpenAI Gym is the racing gym, a single-agent racing game where the agent, a car, must navigate a racing track while avoiding obstacles and completing laps as quickly as possible.

In this project, we explore two of the most popular reinforcement learning algorithms for solving the racing gym: Deep Q-Network (DQN) and Proximal Policy Optimization (PPO). DQN is a deep learning algorithm that combines Q-learning with neural networks to learn a value function that estimates the expected reward for each action in a given state. PPO is a policy gradient algorithm that optimizes a parameterized policy function directly using gradient ascent.

We experiment with different hyperparameters for both algorithms to observe their effects on the learning curve and select the optimal hyperparameters to train our final model. Hyperparameters play a crucial role in the performance of reinforcement learning algorithms. These parameters control the rate of exploration and exploitation, the size of the neural network, and the discount factor for future rewards, among other things. By carefully selecting the hyperparameters, we can improve the performance of the algorithms and reduce the time required for learning.

We compare the performance of these two algorithms on the racing gym environment and discuss the advantages and disadvantages of each approach. In addition, we highlight some of the personal challenges we faced during the project and outline future directions for research in this field.

Overall, this project provides insight into the effectiveness of different reinforcement learning algorithms for solving complex tasks, the importance of hyperparameter tuning in achieving optimal performance, and demonstrates the potential of these algorithms in solving environments similar to the ‘Car Racing’ environment.

1.2 Problem Statement

OpenAI's Gymnasium provides many environments to test reinforcement learning methods. The "Car Racing" gym is one of the environments provided to benchmark reinforcement learning algorithms. The Car Racing gym is a top-down 2D environment where the goal is to teach the agent - the car - to drive around the track. The state space for the problem is a 96x96 pixel RGB image of the car and the track[1].

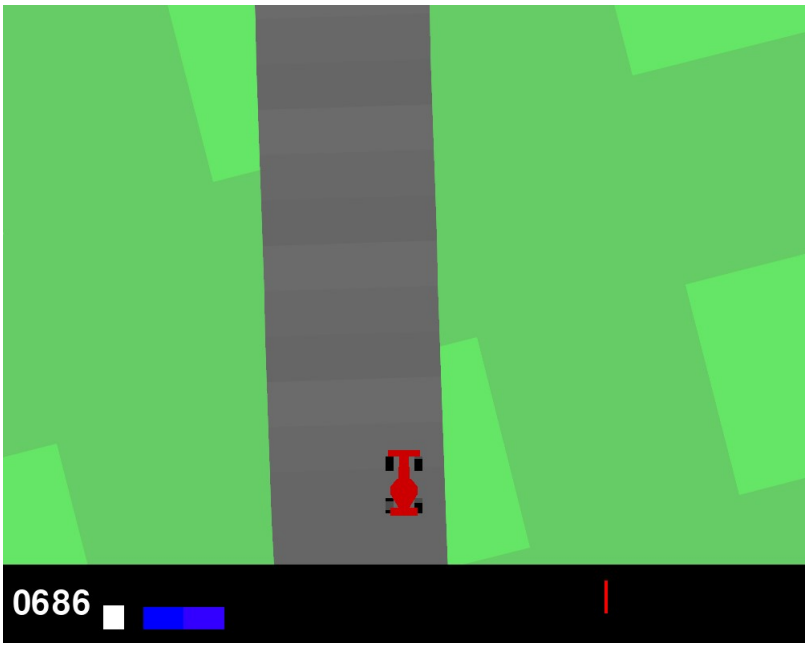


Figure 1.1 The CarRacing Environment

We also need a metric to determine how our agent is performing. This is done using the reward function. The reward function for this environment is:

$$R(t, N) = \frac{1000}{N} - 0.1 * t$$

where, t is the number of frames that have passed so far and N is the total number of tiles our agent - the car - has visited. This reward function encourages the agent to visit as many tiles (N) as possible, and hence finish the track. And simultaneously minimizes the number of frames (t), encouraging to finish the track sooner than later.

In reinforcement learning, an episode is considered all the steps between initial state to final state. In the Car Racing environment, the car is at the center on the race track when the episode begins and the episode terminates when the car visits all the tiles. The episode can also be truncated if the car goes outside of the playfield - that is, far off the track, in which case it will receive a -100 reward and die.

Available Actions	
0	Do nothing
1	Steer Left
2	Steer Right
3	Gas
4	Brake

Figure 1.2 The CarRacing Environment Action Space

In reinforcement learning, the agent must choose an action from a set of possible actions in response to its current state. This set of possible actions is referred to as the action space of the environment. In the Car Racing gym, two options are available for the action space: continuous or discrete. In this project, we opted to work with the discrete action space due to time and scope constraints. The discrete action space in this environment includes five actions: do nothing, steer left, steer right, accelerate, and brake.

Due to the scope of the project, we chose to focus on the discrete action space. This allowed us to simplify the action selection process for the agent, making it easier to implement and train our reinforcement learning models. By limiting the number of available actions, we were able to reduce the complexity of the environment and streamline our implementation. It is worth noting that a discrete action space may not be appropriate for all environments. Although, it is also important to note that the Proximal Policy Optimization (PPO) algorithm, which is one of the methods we discuss in this project, is capable of solving the environment with a continuous action space as well.

Chapter 2

Methods

2.1 Reinforcement Learning

2.1.1 What is Reinforcement Learning?

Reinforcement Learning is a paradigm of machine learning that focuses on teaching an agent(s) to accomplish a goal through interaction with its environment. Reinforcement is often used in solving sequential problems. Sequential problems are problems in which the state of the environment depends on the action the agent took previously.

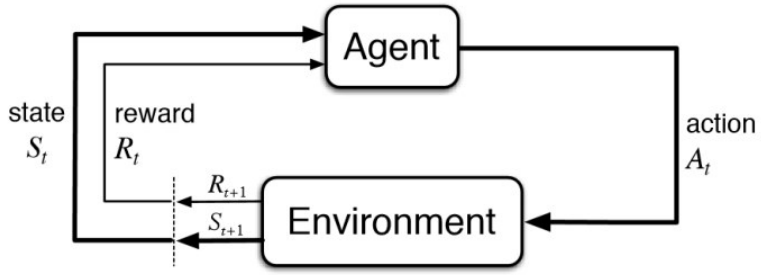


Figure 2.1 The Reinforcement Learning Loop[6]

The agent has (partial) information about its environment, the steps it can take in an environment, and some metric that determines if an action is good or bad, i.e action leads the agent towards accomplishing the goal or not. In reinforcement learning, reaching some value of that metric accomplishes the goal. Formally, the reinforcement learning environment consists of 4 main things: the state space, the agent, the action space,and the reward function. We describe each of these for the Racing Gym in Section 2. Once we have these four components, we can formulate the environment as a Markov Decision Process.

2.1.2 Markov decision process (MDP)

As described in [2], a Markov Decision process is a tuple $< S, A, T, R >$ in which S is a finite set of states, A a finite set of actions, T a transition function defined as $T : S \times A \times S \to [0, 1]$ and R is a reward function $R : S \times A \times S \to \mathbb{R}$.

The above can also include a discount factor, γ that describes how much the reward would be discounted as we move a state to another. "Often MDPs are depicted as a

state transition graph where the nodes correspond to states and (directed) edges denote transitions."[2].

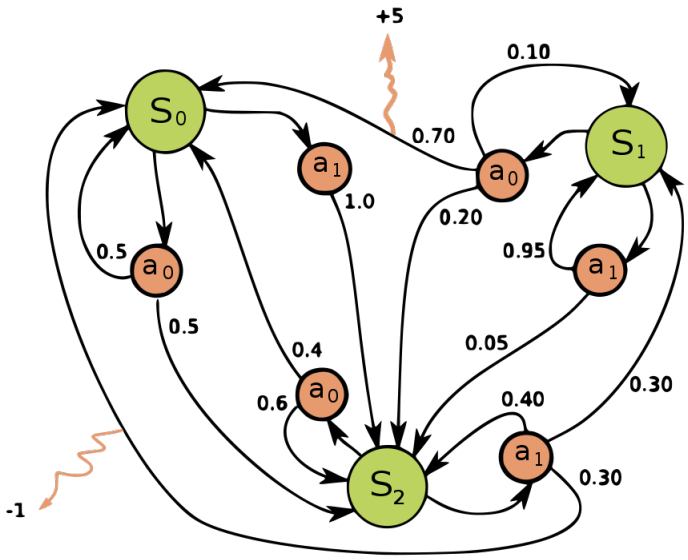


Figure 2.2 Markov Decision Process

2.1.3 Goal of RL: A Policy

The goal of Reinforcement learning is to learn a policy. A policy, given any state in the environment and the actions available to the agent in that state, determines what is the best action the agent can take in that state. So we need to come up with a function:

$$\pi : S \times A \rightarrow [0, 1]$$

where π is the policy. The policy takes the state and the action space as the input and outputs a value between $[0, 1]$ for each action - from which we can choose the best action.

2.2 Deep Reinforcement Learning

2.2.1 Overview

Given the problem, we initially rule out the following methods due to the size of our state space[4]:

1. Maximum likelihood-based methods because due to the large number of state space (256^{3*96^2}), it is likely we have not encountered a state before.

2. Value based reinforcement learning methods such as Vanilla Q-learning and Sarsa - which are table based methods - because they would explode in size due to the large state space.

To handle such a large state space, we make use of Deep Reinforcement Learning. Deep reinforcement learning is the combination of reinforcement learning (RL) and deep learning[5]. Deep Reinforcement Learning utilizes neural networks to approximate the value of the state (or state - action pair) or directly approximates the optimal policy. This is advantageous because neural networks are suited for high dimensional inputs - which is the case given our state space - and can be incrementally trained on experience as it is collected [5], which is what we prefer in solving a sequential problem like this.

2.2.2 Deep Q-Networks (DQN)

"Vanilla" Q-learning is a model-free value based reinforcement learning algorithm. Model-free meaning that it does not require the model of the environment when making the prediction for the values of the function. The Q-learning algorithm works by evaluating "Q-Values" of the state-action pairs in a given state:

$$\underbrace{\text{New}Q(s,a)}_{\text{New Q-Value}} = Q(s,a) + \underbrace{\alpha}_{\text{New Q-Value}}[\underbrace{R(s,a)}_{\text{Reward}} + \underbrace{\gamma}_{\text{Discount rate}} \overbrace{\max Q'(s',a')}^{\text{Maximum predicted reward, given new state and all possible actions}} - Q(s,a)]$$

Once we have the Q-values of all the state-action value pairs in a state, the agent can choose the action with the highest Q-value. Vanilla Q-learning is a tabular method, where all the Q-values of the state-action are stored in a table, and the agent can fetch the Q-value of the state-action pair whenever needed. As we see in the above formula, evaluating the Q-value involves recursive steps, i.e, the Q value of the current state uses the Q-value of the next state. As mentioned in [6], the Q-value spans out to be:

$$Q(s,a) = \gamma Q(s',a) + \gamma^2 Q(s'',a) + ... + \gamma^n Q(s''^{\cdots n}, a)$$

The computation of Q-Value becomes extremely difficult (if not infeasible) with a big state space size, and the table may not be able to fit into limited memory. Hence, we make use of Deep Learning, where we can utilize Neural Networks to predict the Q-value of a state-action pair in any given state[7], rather than storing the Q-values in a table.

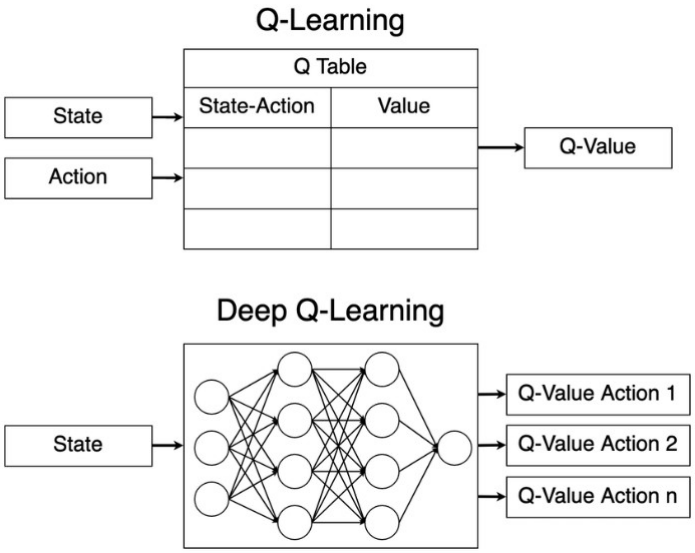


Figure 2.3 Q-Learning vs Deep Q-Network[8]

The brief steps for Deep Q learning involve:

1. Store the experiences in a replay buffer.
2. At each step, determine whether to explore - take random action- or exploit - take the best action as predicted by our Neural Network.
3. Every few steps, update the Neural Network with experiences from the replay buffer.

The key step here is updating/training our Neural Network every few steps. The Loss function for the Neural Network is the Mean Squared Error between the predicted Q-value and Target Q-value. But reflecting on our Bellman equation, we realize that there is no constant target-value. So, we work around that. We introduce one more Neural Network. We now have one prediction Network and one target network. The target network will have the start with the same parameters as the prediction network. The target network will be updated with the parameters of the prediction network every few steps. Introducing a target network makes the target a bit more stationary and hence makes the training a bit more stable[9].

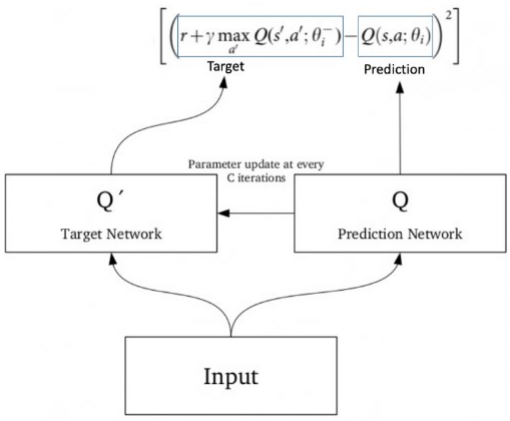


Figure 2.4 Prediction and Target Network in DQN[9]

We use the implementation of DQN provided by Stable Baseline3 [10] to train our agent with different hyperparameters. We explore the effect of tuning different hyperparameters during training on the performance of the DQN algorithm.

[Architecture details of the Neural Network are included in Appendix A]

2.2.3 Proximal Policy Optimization (PPO)

PPO (Proximal Policy Optimization) is a model-free policy-based reinforcement learning method that focuses on learning the policy directly. It uses Policy Gradient to directly learn the policy, rather than learning a value - such as Q-value - that determines the policy. As we know a policy gives us a probability distribution over the actions available in that state. This probability distribution helps us make the decision in that state. Policy based methods, such as PPO, directly learn these probability distributions to determine the policy.

Previous work in policy gradient methods include Trust Region Policy Optimization (TRPO). TRPO tries to make sure that the updates to the policy are safe. PPO core concept is similar - it makes sure that the update to the policy is not too different from the current policy. This makes the updates smoother, and hence makes the learning smoother. In the original PPO paper, it mentions that compared to Trust Region Policy Optimization, PPO is simpler to implement, more general, and has a better sample complexity [11].

As we mentioned, PPO makes the training smoother. It does so by making sure that there are no big leaps when updating the policy. PPO does this by including a "clipping factor (epsilon)". The main objective used in PPO:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[min\left(r_t(\theta)\hat{A}_t, clip\left(r_t(\theta), 1-\epsilon, 1+\epsilon \right)\hat{A}_t \right) \right]$$

where $r_t(\theta)$ is the ratio between the previous policy and the current policy,

$$r_t(\theta) = \frac{\pi_\theta(a_t, s_t)}{\pi_{\theta_{old}}(a_t, s_t)}$$

The L^{CLIP} loss objective asserts that the change between the new and old policy stays between $[1-\epsilon, 1+\epsilon]$. Hence we control how much the policy can chnage in an update, giving us a smoother learning.

We use the implementation of PPO provided by Stable Baseline3 [10] to train our agent with different hyperparameters. We explore the effect of tuning different hyperparameters during training on the performance of the PPO algorithm.

[Architecture details of the Neural Network are included in Appendix A]

Chapter 3

Experiments

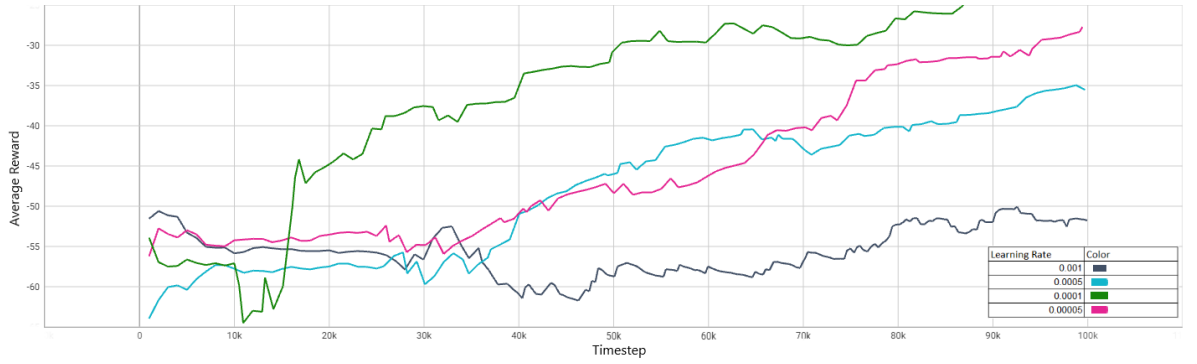
Machine learning algorithms - including Reinforcement Learning algorithms - perform better or worse depending on the choice of hyperparameters. It is known that Reinforcement Learning algorithms are sensitive to their hyperparameters, and they plays a crucial role in solving the environment[12]. The cost of tuning hyperparameters can be quite expensive and also be hard to precisely calculate [13]. So we aim to learn more about how the hyperparameters in our environment affect our learning. Having an insight into hyperparameters could save many hours in testing and solving similar environments.

Below we tune our hyperparameters in both of our algorithms and observe its learning curve. The following learning curves keep track of the average reward the agent obtains per episode at a given timestep. To observe the effect of tuning each hyperparameter in the environment, we compare the different values of that hyperparameter while keeping the values of other hyperparameters unchanged. All models were trained for 100,000 timesteps, except the final two models, which were trained for 1,000,000 timesteps each.

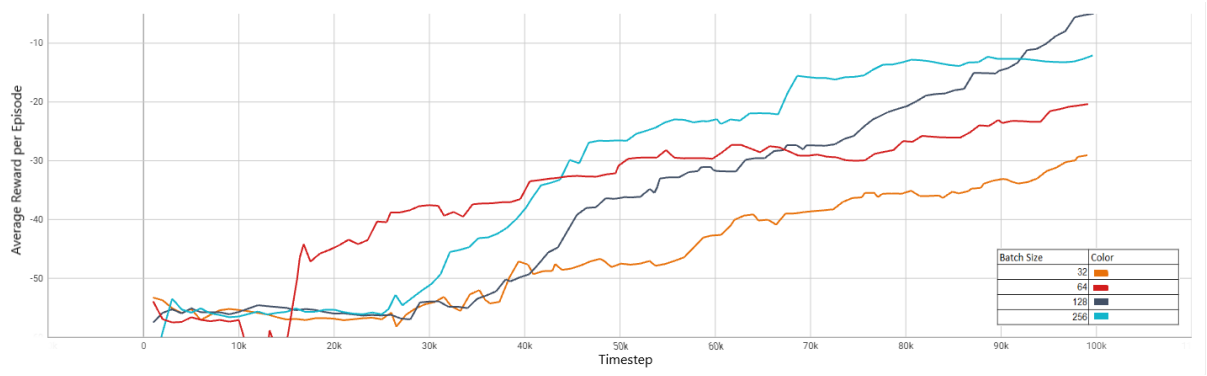
[The training logs are described in Appendix B.]

3.1 DQN Learning Curves

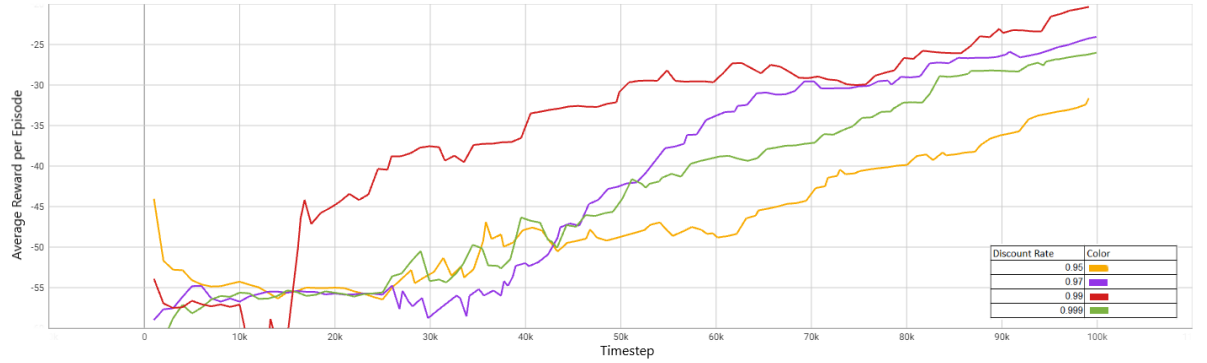
We experiment with learning rate, discount factor, batch size, and prediction model update frequency for our DQN models. Here are the learning curves we observe for each hyperparameter for different values:



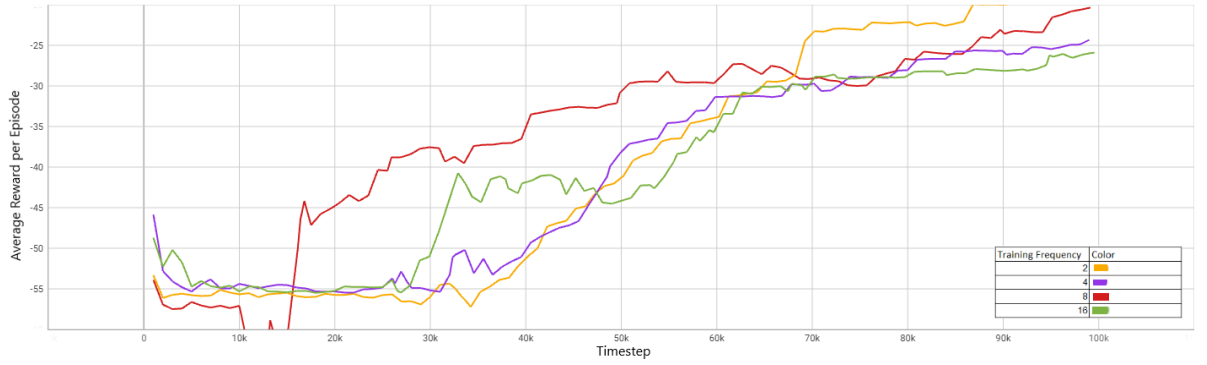
(a) DQN Learning Curves for different learning rates.



(b) DQN Learning Curves for different batch sizes.



(c) DQN Learning Curves for different discount factors.

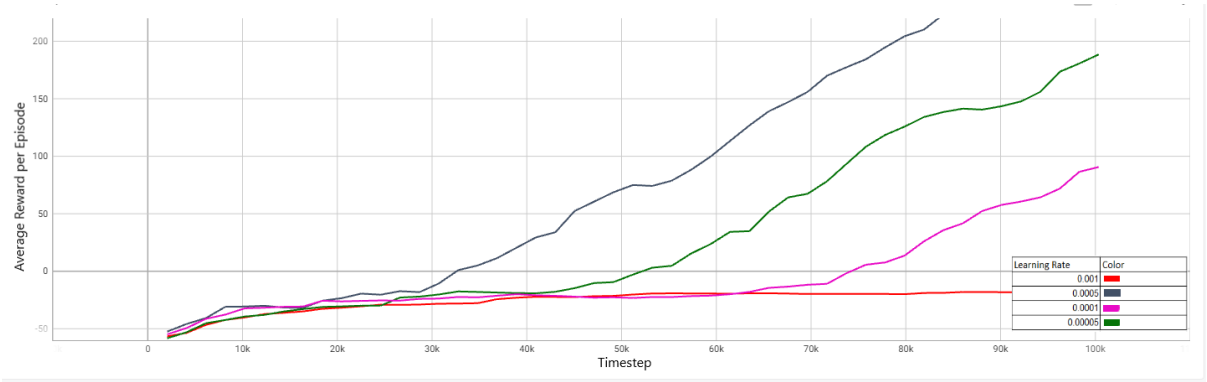


(d) DQN Learning Curves for different prediction model training frequencies.

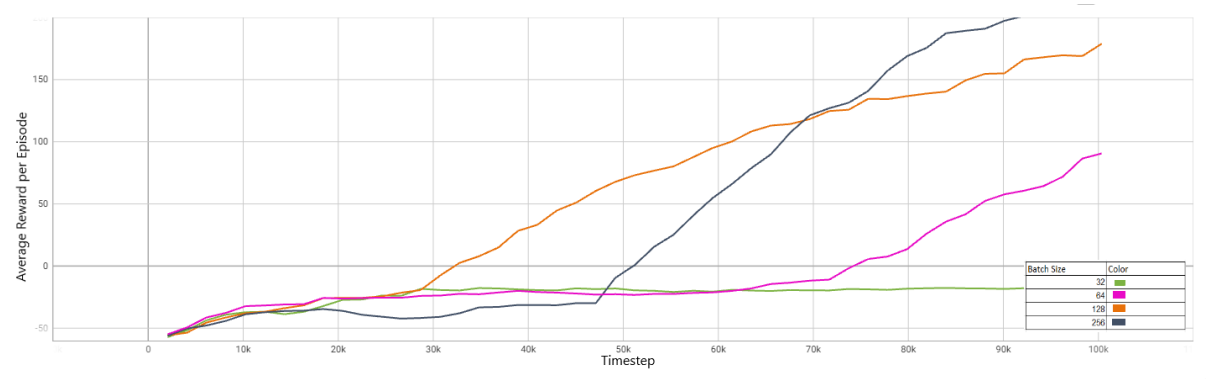
Figure 3.1 DQN Learning Curves

3.2 PPO Learning Curves

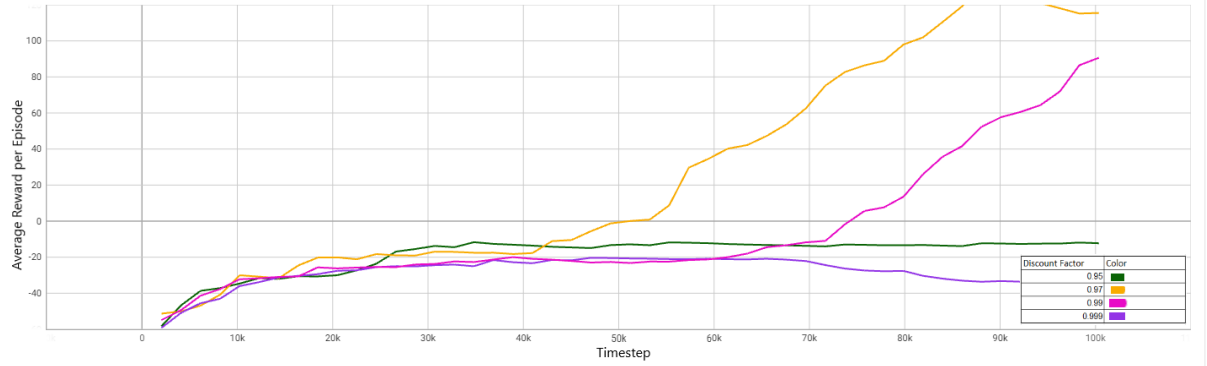
We experiment with learning rate, discount factor, batch size, and clipping range (ϵ) for our PPO models. Here are the learning curves we observe for each hyperparameter for different values:



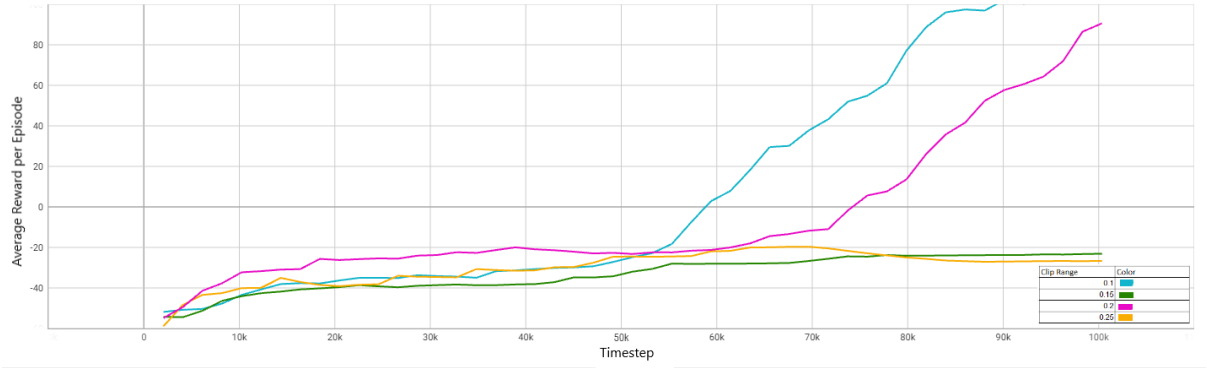
(a) PPO Learning Curves for different learning rates.



(b) PPO Learning Curves for different batch sizes.



(c) PPO Learning Curves for different discount factors.



(d) PPO Learning Curves for different clipping ranges((ϵ).

Figure 3.2 PPO Learning Curves

3.3 Final Learning Curves

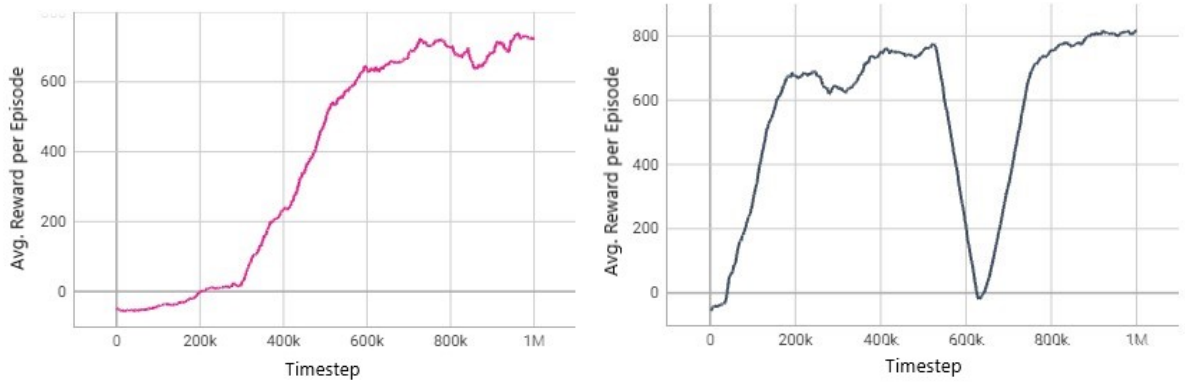
Once we tried and tested different hyperparameters for both the algorithms, we chose the hyperparameters the converged the fastest to a higher reward, i.e., had the steepest overall incline in the learning curve. In case the different values for that hyperparameters showed no significant difference, we arbitraility chose one.

For DQN, the final hyperparameters were:

1. Learning Rate = 0.001
2. Discount Factor = 0.99
3. Batch Size = 64
4. Prediction Model Training Frequency = 8

For PPO, the final hyperparameters were:

1. Learning Rate = 0.005
2. Discount Factor = 0.99
3. Batch Size = 256
4. Clip Range = 0.2



(a) DQN Final Model Learning Curve. **(b)** PPO Final Model Learning Curve.

Figure 3.3 Final Learning Curves.

Chapter 4

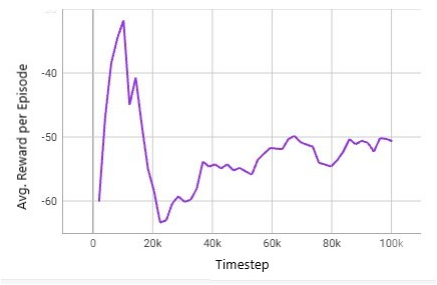
Discussion

Setting up the experiments and performing them provided us with several insights into the algorithms and how the hyperparameters affect the learning. We discuss some of the observations in the following subsections.

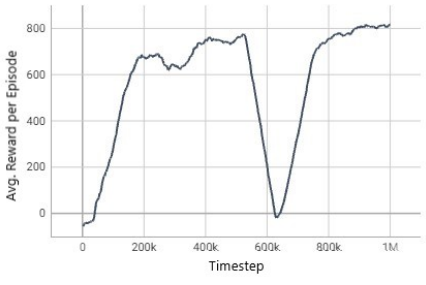
4.1 DQN vs PPO

1. Our findings indicate that the learning curves of Proximal Policy Optimization (PPO) exhibited greater smoothness compared to those of Deep Q Learning (DQN). This observation corroborates our earlier claim that PPO is designed to mitigate the occurrence of abrupt and significant updates to the agent's policy.

2. It is important to acknowledge that if Proximal Policy Optimization (PPO) inadvertently learns negative behaviors during training, it may result in a sharp decline in rewards. We observe "Catastrophic Interference" where the model completely forgets what it had learned. In such instances, the agent may find it exceedingly difficult, if not impossible, to recover. Our Final model observed catastrophic interference, but was fortunately able to recover from that. To avoid catastrophic interference we recommend the use of smaller learning rates (see section 4.2.1) while implementing PPO to prevent the agent from learning negative behaviors. We also recommend saving frequent checkpoints so the model is saved around the peak of convergence in case of catastrophic interference.



(a) PPO not able to recover after Catastrophic Interference given not enough time.



(b) PPO able to recover after Catastrophic Interference given enough time.

Figure 4.1 Catastrophic Interference in PPO.

3. We found that the majority of the learning curves for PPO converged to higher rewards compared to DQN's learning curves with similar hyperparameters over the given 100,000 timesteps, except for those that failed to learn entirely. This observation suggests that PPO may have a better capacity for optimizing the agent's policy in the given environment.

4. As the Deep Q-Network (DQN) algorithm has a longer convergence time, we initially aimed to train our final DQN model for double the timesteps as the Proximal Policy Optimization (PPO) algorithm, amounting to 2,000,000 timesteps. During this extended training period, we observed a decline in the reward values of our DQN model at around 1,000,000 timesteps. Catastrophic interference may occur in DQN when a well-fit model may only have *good* state-action pairs in its replay buffer. Due to that, the model trains on that biased sample, forgets what to do with *bad* state-action pairs and incorrectly assigns high Q-values to *bad* state-action pairs. Consequently, this leads to a degradation in the model's performance. To avoid catastrophic interference, we can roughly project the trajectory of the learning curve and set appropriate stopping criteria. In other words, we need to ensure that our models are not over-trained.

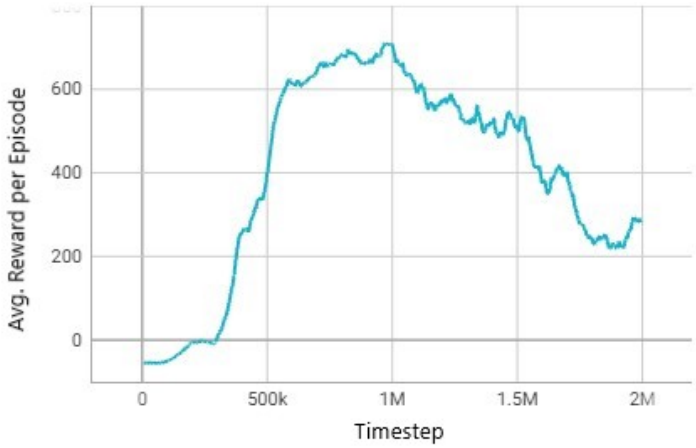


Figure 4.2 Catastrophic Interference in DQN.

4.2 Hyperparameter Insights

1. Learning rates on the scale of $\sim 10^{-3}$ or lower exhibited positive growth in average reward for both Deep Q Learning (DQN) and Proximal Policy Optimization (PPO). However, we observed that learning rates of scale 10^{-2} resulted in no learning for both algorithms. Clearly as we observed in the final learning curve of PPO, even the learning rate of $5 * 10^{-3}$ was a bit high and resulted in a decline after some convergence. Therefore, we recommend setting learning rates to at most $\sim 10^{-3}$ or lower while training the agents in these algorithms.

2. All of the four discount factors tested were suitable for training Deep Q Learning (DQN) models, as they were able to learn with all of them. However, only two of the four discount factors were found to be suitable for training Proximal Policy Optimization (PPO) models, as only those two were able to learn using the tested

discount factors. We did not observe any correlation between the discount factor and the convergence of the learning in our experiments. Nonetheless, we hypothesize that the behavior of our agent - the car - may vary considerably depending on the discount factor, which could affect the overall performance of the reinforcement learning algorithm.

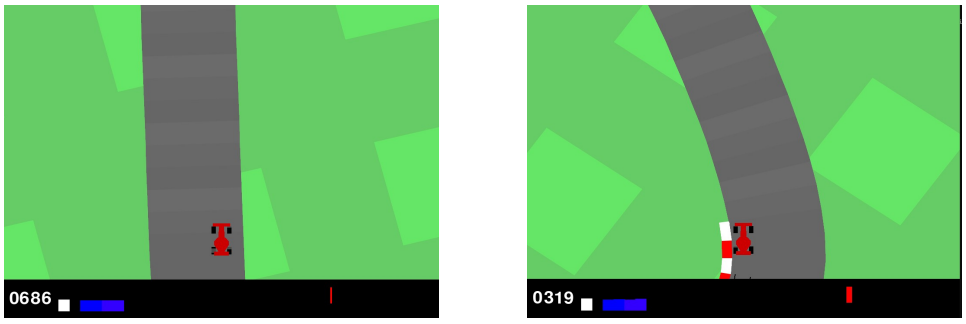
3. Our experiments showed that Deep Q Learning (DQN) models were able to learn using all of the tested batch sizes. On the other hand, Proximal Policy Optimization (PPO) models were only able to learn using all batch sizes except for the batch size of 32. We hypothesize that larger batch sizes may lead to faster convergence of the learning process, although we did not test this hypothesis in this project with our time constraints.

4. The DQN model utilized in our study consisted of two networks: a prediction network and a target network. We performed experiments to determine the effect of a particular hyperparameter, namely, the frequency of steps at which the prediction model is updated. Our results indicate that varying this hyperparameter within a certain range did not result in any significant difference in the learning convergence of the DQN model.

5. Based on the experiments conducted in the environment, two of the four clipping ranges tested in PPO were able to learn, namely values 0.1 and 0.2, while clipping range values 0.15 and 0.25 did not exhibit any learning. However, there was no clear correlation found between clipping range and the learning curve in this particular environment. Further experiments would be required to gain more insight into the effect of clipping range on the learning curve in this environment.

4.3 Agent Behaviour

1. The autonomous agent, represented by a car, was capable of successfully navigating the racing track in both the Proximal Policy Optimization (PPO) and Deep Q-Network (DQN) algorithms, achieving an average reward of 815.7 and 720.00, respectively. Both the models consistently achieved higher reward when evaluated. We also determine the environment to be solved with an eyeball test and make sure the car doesn't have any random behaviour and can go around the racetrack in reasonable time.

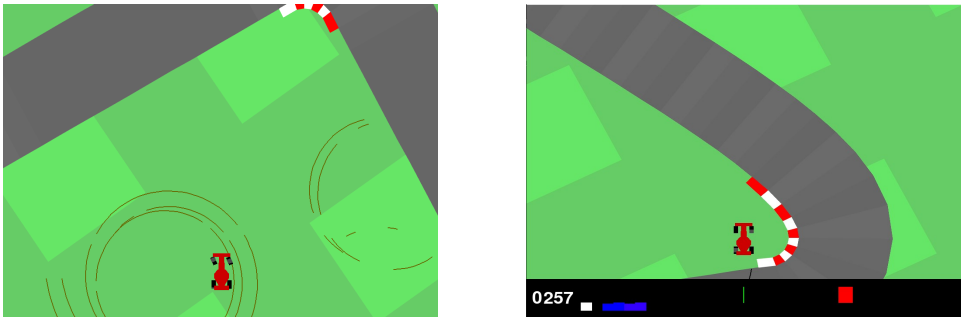


(a) Car driving on a straight path. **(b)** Car finding the racing line.

Figure 4.3 Positive Agent Behaviour

2. An important limitation identified in our approach pertains to the absence of orientation tracking of the car in a given state. Consequently, if the car visits a tile in an opposite orientation, i.e., heading the wrong way, our model's policies may select an action based on the assumption that the car is in the correct orientation.

3. The car's behavior was observed to be erratic when it partially or completely went off track and into the grass. In some instances, the car would continue in loops, while in most cases, it was able to find its way back to the track. However, if it returned to the track facing the wrong direction, the issue of the car being in the wrong orientation mentioned earlier would arise. Occasionally, the car would also come to a stop in this situation.



(a) Car going in circles. **(b)** Car cutting through a turn.

Figure 4.4 Negative Agent Behaviour

4.4 Cost of Hyperparameter Tuning

1. As previously noted, it should be emphasized that the process of hyperparameter tuning can be a resource-intensive task, which can be challenging to accurately estimate [13]. In our case, each trial within each experiment was executed over 100,000 timesteps, with an average runtime of approximately 50 minutes on our current hardware (CPU: Intel i5-4960 @ 3.50 Ghz, GPU: NVIDIA GeForce GTX 1650, RAM: 16.0GB 1600MHz DDR3). Moreover, it should be noted that the final PPO and DQN models required 1,000,000 timesteps respectively to complete the training process.

2. That rounds up to ~ 70 hours of training without accounting for other testing and configuration. This does not include some of the smaller training sessions conducted to test the correctness of the code. The training was mainly done using GPU, and hence also has a significant energy cost and an environmental footprint associated with it.

3. Hence, obtaining knowledge of hyperparameters and model architecture can prove to be highly beneficial when deploying reinforcement learning algorithms in similar environments, such as those included in OpenAI's Gymnasium.

4.5 Challenges Faced

1. The successful training and testing of machine learning models heavily relies on hardware resources, particularly on access to GPUs, due to the computational complexity involved. However, due to personal challenges experienced during the course of the project, access to GPUs was temporarily unavailable for a few weeks, impeding progress and the opportunity for hands-on learning and testing.

2. While undertaking this project, we faced the challenge of having no prior knowledge or experience in reinforcement learning and deep learning, which made it necessary for us to invest significant time and effort to understand and grasp the various concepts involved regarding the theory and its execution.

3. The from-scratch implementation for DQN in this project encountered issues that persisted for a prolonged duration. In hindsight, we realized that the hyperparameters used in our initial implementation were of a completely different scale compared to the ones that have been reported to solve the environment. Therefore, it would have been advantageous to gain some understanding and conduct research on appropriate hyperparameters for this specific environment before proceeding with the implementation of the DQN algorithm from scratch.

Chapter 5

Conclusion

Our experiments show that both DQN and PPO can be used to train an agent to navigate a simple racetrack environment. Our experiments provide some insights into the effect of hyperparameters such as learning rate, discount factor, batch size, and clipping range on the convergence of learning in the environment. We also encountered some limitations and challenges during our implementation, such as the cost of tuning hyperparameters, the hardware dependence of machine learning, and the challenges of understanding new concepts and implementing the algorithms from scratch.

While our experiments provide some valuable insights, we acknowledge that there is still much to be explored and studied in the field of reinforcement learning. Our experiments focused on a simple racetrack environment, and we believe that the results and insights gained can be further extended and applied to more complex environments.

We also acknowledge the potential for further optimization and improvement in the implementation of our models. For example, future work may explore the effects of more advanced techniques such as prioritized experience replay or distributed learning. Future work could also investigate the potential for transfer learning in reinforcement learning, where knowledge gained from training in one environment can be transferred and applied to another environment. This could be particularly useful in cases where the target environment is more complex and requires longer training times.

This project provided a comprehensive analysis of two popular reinforcement learning algorithms, DQN and PPO, in the context of OpenAI's racing gym challenge. We highlight the importance of hyperparameter tuning in developing an effective learning model and demonstrate the potential of reinforcement learning algorithms in addressing intricate decision-making problems in challenging environments. However, we acknowledge that there is still much to be explored and studied in the field of reinforcement learning, and we hope that our work can contribute to further advancements in this exciting field.

References

(1) Klimov, O. Car Racing - Gymnasium Documentation, https://gymnasium.farama.org/environments/box2d/car_racing/.

(2) Van Otterlo, M.; Wiering, M. Reinforcement learning and markov decision processes, https://link.springer.com/chapter/10.1007/978-3-642-27645-3_1, 1970.

(3) Markov decision process, https://en.wikipedia.org/wiki/Markov_decision_process, 2023.

(4) Pablo Aldape, S. S. Reinforcement Learning for a Simple Racing Game, https://web.stanford.edu/class/aa228/reports/2018/final150.pdf, 2018.

(5) Francois-Lavet, V.; Henderson, P.; Islam, R.; Bellemare, M. G.; Pineau, J. An introduction to deep reinforcement learning, https://arxiv.org/abs/1811.12560, 2018.

(6) Sutton, R. S.; Bach, F.; Barto, A. G., *Reinforcement learning: An introduction*; MIT Press Ltd: 2018.

(7) Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; Riedmiller, M. Playing Atari with deep reinforcement learning, https://arxiv.org/abs/1312.5602, 2013.

(8) Sebastianelli, A.; Tipaldi, M.; Ullo, S. L.; Glielmo, L. A deep Q-learning based approach applied to the snake game. https://ieeexplore.ieee.org/document/9480232/.

(9) Choudhary, A. A hands-on introduction to deep Q-learning using openai gym in python, https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/, 2020.

(10) Stable Baseline3 DQN, https://stable-baselines3.readthedocs.io/en/master/modules/dqn.html.

(11) Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. Proximal policy optimization algorithms, https://arxiv.org/abs/1707.06347, 2017.

(12) Eimer, T.; Benjamins, C.; Lindauer, M. Hyperparameters in contextual RL are highly situational, https://arxiv.org/abs/2212.10876#:~:text=Abstract%3A%20Although%20Reinforcement%20Learning%20, 2022.

(13) Obando-Ceron, J. S.; Castro, P. S. Revisiting rainbow: Promoting more insightful and Inclusive Deep Reinforcement Learning Research, https://arxiv.org/abs/2011.14826, 2021.

Appendix A

Network Architectures

A.1 DQN Architecture

```
CnnPolicy(
  (q_net): QNetwork(
    (features_extractor): NatureCNN(
      (cnn): Sequential(
        (0): Conv2d(3, 32, kernel_size=(8, 8), stride=(4, 4))
        (1): ReLU()
        (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
        (3): ReLU()
        (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
        (5): ReLU()
        (6): Flatten(start_dim=1, end_dim=-1)
      )
      (linear): Sequential(
        (0): Linear(in_features=4096, out_features=512, bias=True)
        (1): ReLU()
      )
    )
    (q_net): Sequential(
      (0): Linear(in_features=512, out_features=5, bias=True)
    )
  )
  (q_net_target): QNetwork(
    (features_extractor): NatureCNN(
      (cnn): Sequential(
        (0): Conv2d(3, 32, kernel_size=(8, 8), stride=(4, 4))
        (1): ReLU()
        (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
        (3): ReLU()
        (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
        (5): ReLU()
        (6): Flatten(start_dim=1, end_dim=-1)
      )
      (linear): Sequential(
```

```
        (0): Linear(in_features=4096, out_features=512, bias=True)
        (1): ReLU()
      )
    )
    (q_net): Sequential(
      (0): Linear(in_features=512, out_features=5, bias=True)
    )
  )
)
```

A.2 PPO Architecture

```
ActorCriticCnnPolicy(
  (features_extractor): NatureCNN(
    (cnn): Sequential(
      (0): Conv2d(3, 32, kernel_size=(8, 8), stride=(4, 4))
      (1): ReLU()
      (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
      (3): ReLU()
      (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
      (5): ReLU()
      (6): Flatten(start_dim=1, end_dim=-1)
    )
    (linear): Sequential(
      (0): Linear(in_features=4096, out_features=512, bias=True)
      (1): ReLU()
    )
  )
  (pi_features_extractor): NatureCNN(
    (cnn): Sequential(
      (0): Conv2d(3, 32, kernel_size=(8, 8), stride=(4, 4))
      (1): ReLU()
      (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
      (3): ReLU()
      (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
      (5): ReLU()
      (6): Flatten(start_dim=1, end_dim=-1)
    )
    (linear): Sequential(
      (0): Linear(in_features=4096, out_features=512, bias=True)
      (1): ReLU()
    )
  )
  (vf_features_extractor): NatureCNN(
    (cnn): Sequential(
      (0): Conv2d(3, 32, kernel_size=(8, 8), stride=(4, 4))
      (1): ReLU()
```

```
      (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
      (3): ReLU()
      (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
      (5): ReLU()
      (6): Flatten(start_dim=1, end_dim=-1)
    )
    (linear): Sequential(
      (0): Linear(in_features=4096, out_features=512, bias=True)
      (1): ReLU()
    )
  )
  (mlp_extractor): MlpExtractor(
    (policy_net): Sequential()
    (value_net): Sequential()
  )
  (action_net): Linear(in_features=512, out_features=5, bias=True)
  (value_net): Linear(in_features=512, out_features=1, bias=True)
)
```

Appendix B

Training Logs

B.1 DQN Logs

Constant Hyper Parameters	
Buffer Size	50000
Learning Starts	10000

Learning Rate Trials				
DQN				
Training Number	Learning Rate	Discount Factor (Gamma)	Batch Size	Train Frequency
1	0.001	0.99	64	8
2	0.0005	0.99	64	8
3	0.0001	0.99	64	8
4	0.00005	0.99	64	8

Discount Factor Trials				
DQN				
Training Number	Learning Rate	Discount Factor (Gamma)	Batch Size	Train Frequency
1	0.0001	0.95	64	8
2	0.0001	0.97	64	8
3	0.0001	0.99	64	8
4	0.0001	0.999	64	8

Batch Size Trials				
DQN				
Training Number	Learning Rate	Discount Factor (Gamma)	Batch Size	Train Frequency
1	0.0001	0.99	32	8
2	0.0001	0.99	64	8
3	0.0001	0.99	128	8
4	0.0001	0.99	256	8

Train Frequency Trials				
DQN				
Training Number	Learning Rate	Discount Factor (Gamma)	Batch Size	Train Frequency
1	0.0001	0.99	64	2
2	0.0001	0.99	64	4
3	0.0001	0.99	64	8
4	0.0001	0.99	64	16

B.2 PPO Logs

Learning Rate Trials				
PPO				
Training Number	Learning Rate	Discount Factor (Gamma)	Batch Size	Clip Range
1	0.001	0.99	64	0.2
2	0.0005	0.99	64	0.2
3	0.0001	0.99	64	0.2
4	0.00005	0.99	64	0.2

Discount Factor Trials				
PPO				
Training Number	Learning Rate	Discount Factor (Gamma)	Batch Size	Clip Range
1	0.0001	0.95	64	0.2
2	0.0001	0.97	64	0.2
3	0.0001	0.99	64	0.2
4	0.0001	0.999	64	0.2

Batch Size Trials				
PPO				
Training Number	Learning Rate	Discount Factor (Gamma)	Batch Size	Clip Range
1	0.0001	0.99	32	0.2
2	0.0001	0.99	64	0.2
3	0.0001	0.99	128	0.2
4	0.0001	0.99	256	0.2

Clip Range Trials				
PPO				
Training Number	Learning Rate	Discount Factor (Gamma)	Batch Size	Clip Range
1	0.0001	0.99	64	0.1
2	0.0001	0.99	64	0.15
3	0.0001	0.99	64	0.2
4	0.0001	0.99	64	0.25