ROBUST TRANSFER LEARNING IN DEEP REINFORCEMENT LEARNING

A PROJECT REPORT

Submitted By

SAI PRAKASH 185001132

MAHESH BHARADWAJ K 185001089

PRITHAM IMMANUEL ILANGO 185001117

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

Department of Computer Science and Engineering

Sri Sivasubramaniya Nadar College of Engineering

(An Autonomous Institution, Affiliated to Anna University)

Rajiv Gandhi Salai (OMR), Kalavakkam - 603110

June 2022

Sri Sivasubramaniya Nadar College of Engineering

(An Autonomous Institution, Affiliated to Anna University)

BONAFIDE CERTIFICATE

Certified that this project report titled **"ROBUST TRANSFER LEARNING IN DEEP REINFORCEMENT LEARNING"** is the *bonafide* work of "**MAHESH BHARADWAJ K (185001089)**, **PRITHAM IMMANUEL ILANGO. G (185001117)** and **SAI PRAKASH (185001132)**, " who carried out the project work under my supervision.

Certified further that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

DR. T.T. MIRNALINEE
HEAD OF THE DEPARTMENT
Professor,
Department of CSE,
SSN College of Engineering,
Kalavakkam - 603 110

DR. R.S. MILTON
SUPERVISOR
Professor,
Department of CSE,
SSN College of Engineering,
Kalavakkam - 603 110

Place:
Date:

Submitted for the examination held on. . . . . .

Internal Examiner **External Examiner**

ACKNOWLEDGEMENTS

We thank GOD, the almighty for giving me strength and knowledge to do this project.

We would like to thank and deep sense of gratitude to our guide **DR. R.S. MILTON**, Professor, Department of Computer Science and Engineering, for his valuable advice and suggestions as well as his continued guidance, patience and support that helped us to shape and refine our work.

Our sincere thanks to **Dr. T.T. MIRNALINEE**, Professor and Head of the Department of Computer Science and Engineering, for her words of advice and encouragement and we would like to thank our project Coordinator **Dr.B. BHARATHI**, Associate Professor, Department of Computer Science and Engineering for her valuable suggestions throughout this project.

We express our deep respect to the founder **Dr. SHIV NADAR**, Chairman, SSN Institutions. We also express our appreciation to our **Dr. V. E. ANNAMALAI**, Principal, for all the help he has rendered during this course of study.

We would like to extend our sincere thanks to all the teaching and non-teaching staffs of our department who have contributed directly and indirectly during the course of our project work. Finally, We would like to thank our parents and friends for their patience, cooperation and moral support throughout my life.

MAHESH **PRITHAM IMMANUEL** **SAI PRAKASH**
BHARADWAJ K **ILANGO**

ABSTRACT

Reinforcement Learning is the branch of artificial intelligence where the agent learns by an iterative process of taking actions in its environment and receiving rewards from the environment based on the effect of the actions taken. Reinforcement learning is particularly useful for simulating use cases which in the real world cannot be experimented with due to the hazardous nature of the experiment such as autonomous driving vehicles. Transfer Learning is popular in deep learning as it increases efficiency and reduces training costs. However, transfer learning in reinforcement learning is significantly harder because of the variation in state space of different tasks. In this paper, we aim to study the effects of adversarial training when implementing transfer learning for reinforcement learning. We hypothesize that better representation learning from adversarial training will enhance transfer learning and make it more viable for reinforcement learning tasks. We also develop an easy-to-use, customizable environments in Unity that use the MLagents framework, to train different models. We have designed 5 distinct tracks of increasing complexity on which we have trained 'cars' to complete a lap successfully using the Proximal Policy Optimization algorithm to perform our experiments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Reinforcement Learning (RL) is a branch of artificial intelligence where the agent learns by a constant process of taking actions in the environment and receiving rewards from the environment based on the correctness of the actions taken. RL is particularly useful for simulating use cases which in the real world cannot be experimented with due to the hazardous nature of the experiment such as autonomous driving vehicles.

However, RL becomes difficult to use when we need to train a large number of agents. RL algorithms generally take a lot of time and compute when training from scratch, which poses a problem for tasks that are only marginally different from each other. In such scenarios where tasks are sufficiently similar to each other, Transfer Learning (TL) provides a way to save time and compute by reusing the knowledge already learnt so far.

TL is uniquely challenging in RL when compared to its contemporaries in Deep Learning because the samples are being generated from a dynamic environment and not from a static dataset. The agent takes certain actions and the rewards gained from those actions are fed back into the agent for further training. There is a lot of inherent variability in the kinds of values that can be generated. Therefore, TL in RL can be quite unpredictable even between marginally different environments. The models get perturbed easily and no transfer of knowledge takes place. In order for TL to perform well, much more robust representations of the state are required.

Adversarial Training (AT) was initially used in Deep Learning to help protect models from adversarial attacks. Adversarial Attacks involved adding random bits of noise to the inputs that the model receives in order to confuse the model. Only a small amount of noise is required to disturb the models. AT involves the addition of noise to inputs during the training process itself, in order to better prepare the agents against small perturbations in the inputs. This training helps the model build more robust representations of the state space, which also improves the ability of the model to transfer to new environments. We hypothesize that these kinds of robust representations will result in a more improved form of TL for RL tasks.

In our work, we aim to quantify the effectiveness of Transfer Learning with Adversarial Training when it comes to Reinforcement Learning environments. We perform Transfer Learning with Adversarial Training in two different methods and compare its effectiveness to vanilla Transfer Learning in RL environments of varying levels of similarities.

CHAPTER 2

BACKGROUND AND MOTIVATION

2.1 REINFORCEMENT LEARNING

Reinforcement Learning is a sub-domain of machine learning, in which a learner or agent must find an optimal method to perform a specific task when interacting with a dynamic environment. The agent is made to learn from the experience it gains from said interaction. The actions of the agent influence the environment, and thus also influences the agent's future actions. Therefore the goal of an agent is to find the optimal actions for the state it's in, with an objective of maximizing net reward.

An agent in the environment, in the state s_i, performs an action a_i using a policy π, to reach the state s_{i+1}, upon which the agent collects a reward r_i. The policy π is used to take an action a_i based on the value function at s_{i+1}, so as to maximize its future rewards. The reward returned to the model, based on its action, is then used as feedback to quantify the agent's performance.

Most reinforcement learning problems can be formally defined as a Markov Decision Process. That is, the problems can be defined following the Markov Property: the transition probability and the reward depend only on the current state s_i and action a_i, and are not influenced by the past states and actions.

A key component of Reinforcement learning is the exploration-exploitation problem. Exploration is the virtue by which an agent is able to experience new

states while interacting with the environment, and exploitation is when the agent is able to use its knowledge gained in prior iterations to choose the optimal decision. During training it is recommended the agent explores a multitude of states, to learn the alternative paths to the end goal, in order to determine the optimal path. However it is also important to monitor the exploration, to ensure that the agent does miss a potentially optimal path.

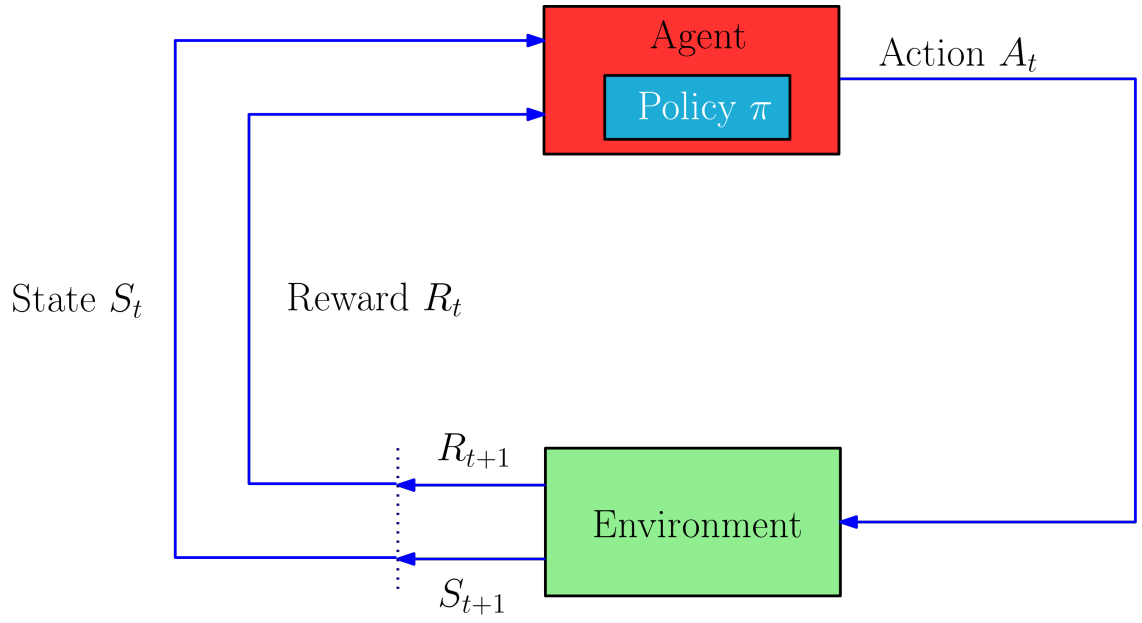

FIGURE 2.1: Reinforcement Learning

2.1.1 DEEP REINFORCEMENT LEARNING

We have already seen that in reinforcement learning, the agent tries to learn a policy π in order to maximize its rewards. The policy π is usually given by $\pi(s \leftarrow a)$, a function mapping states to actions. In deep reinforcement learning we use neural networks to approximate the policy π, and other learnable functions such as the value function.

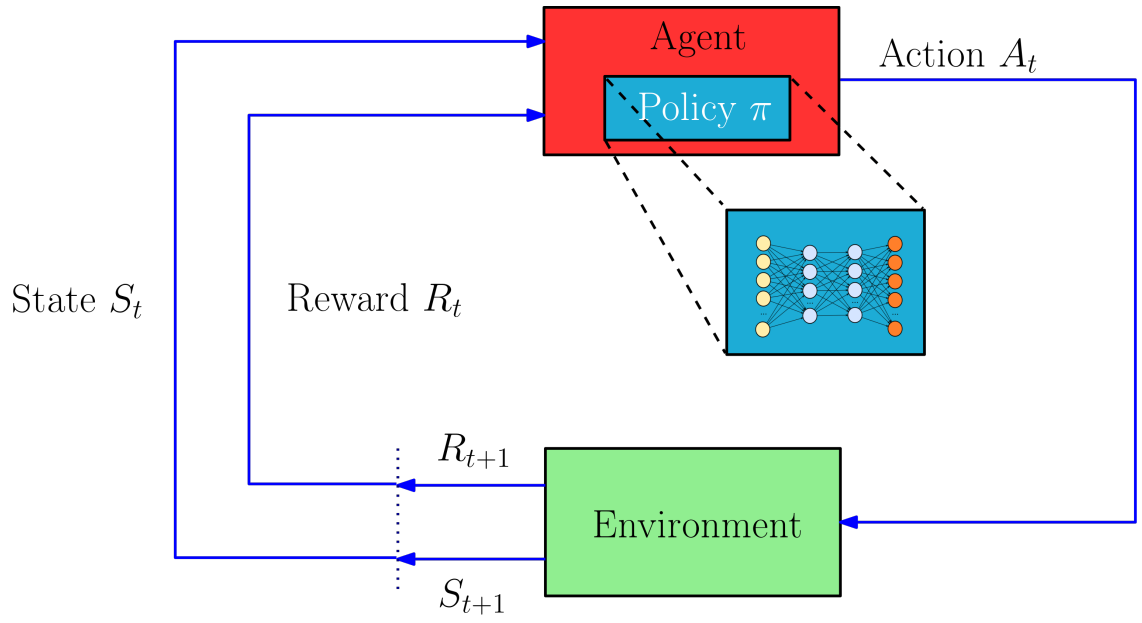

FIGURE 2.2: Deep Reinforcement Learning

2.2 TRANSFER LEARNING

Transfer learning is a method that improves the performance on the target task by leveraging the knowledge gained by performing the same experiment on a similar source task. This can be often seen in supervised learning problems such as image labeling, where the knowledge from the source task is used to enhance the performance and efficiency of learning in the target task. However, this can be quite challenging in reinforcement learning, as each problem may have a different state space. We are able to utilize transfer learning to learn tasks that are rather similar, i.e, having a similar observation space and states.

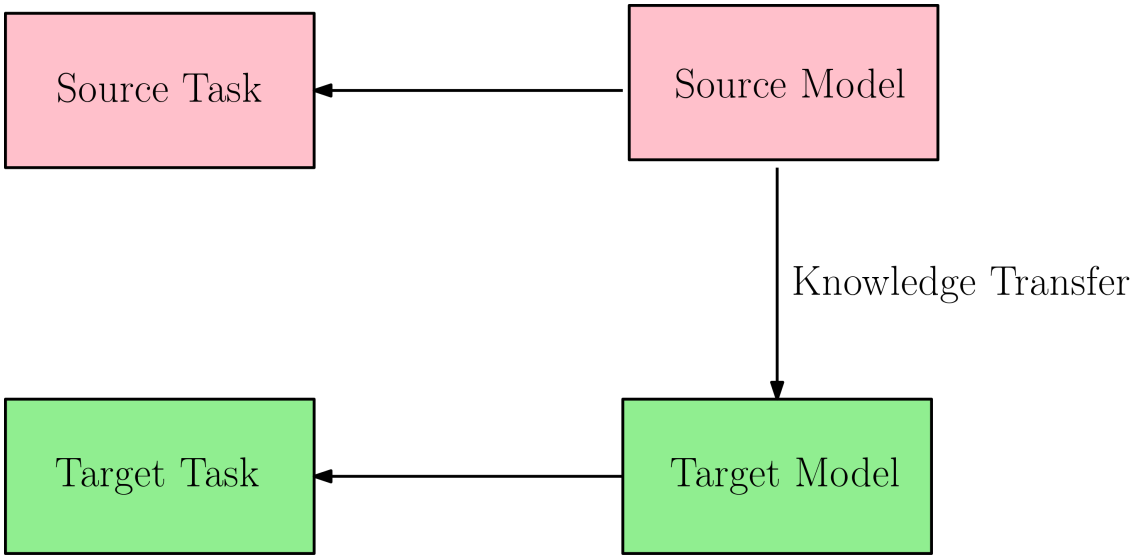

FIGURE 2.3: Transfer Learning

2.3 ADVERSARIAL TRAINING

Many machine learning and deep learning models are often trained and tested on data from the same statistical distribution, and are hence often overtuned to perform well only on said distribution. This however causes issues when deployed with real world data and may be prone to adversaries that may take advantage of this fact and compromise the result of the model. Thus we use adversarial training to mitigate this concern.

Adversarial training in reinforcement learning is made possible by using an adversary to attack the observations or the actions of the model. Knowing that adversarial training improves representational learning in supervised models, we can intuitively say that using adversarial training along with transfer learning will improve the performance of an agent using reinforcement learning.

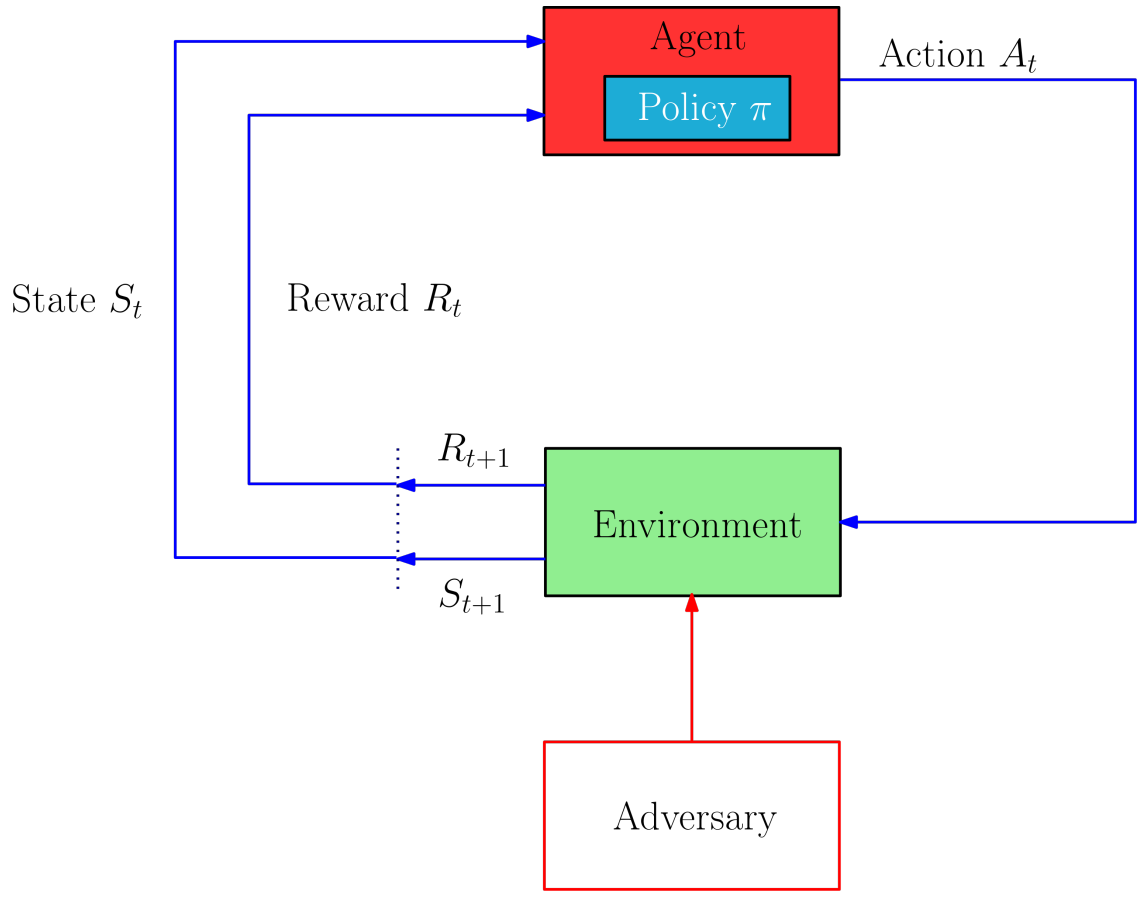

FIGURE 2.4: Adversarial Training in Reinforcement Learning

2.4 UNITY

Unity is the game engine we used to create our environments. Unity allows us to create a 3D environment on which we can run our reinforcement learning algorithms. It uses a scripting API written in C#. The engine also supports Object Oriented Programming paradigm, to create various game objects components that enable us to create an interactive environment for our agent.

Unity provides the user with a console, in which we can visualize the various components of the environments. The properties and functionalities of said

components are defined in their respective script files. This gives us great control over the environment. We can set the goal state of the agent and define the various events on which we give the agent a reward, this is done by calling the API provided by unity. This flexibility in defining the problem makes Unity all the more alluring and our primary choice for creating an environment.

Unity provides a package known as MLagents, which provides a wrapper API and some baseline implementations of well known Deep Reinforcement learning algorithms. It also acts as the connector between the model and the environment enabling us to configure different models and train them in our environment.

CHAPTER 3

LITERATURE SURVEY

This chapter covers the literature pertaining to Reinforcment Learning (**??**), Deep Reinforcement Learning (**??**), Transfer Learning (**??**), Adversarial Training (**??**) and Unity ML Agents (**??**).

3.1 REINFORCEMENT LEARNING

Reinforcement Learning refers to a trial-and-error interaction based approach by which an agent learns to behave in its environment. This is inspired by animal psychology study and has its roots in statistics, psychology and computer science [**?**]. Some early real world applications of RL involved the usage of computer simulation enhanced common sense based rules to create a fuzzy logic system to design nuclear power plant control systems [**?**]. The usage of multi-layer perceptron as a function approximator was first studied by [**?**] for the cart pole problem [**?**].

3.2 DEEP REINFORCEMENT LEARNING

Deep Reinforcement learning (DRL) has been able to solve a wide range of complex decision-making tasks that were previously out of reach for a machine [**?**]. Q Learning [**?**] is a primitive RL algorithm wherein the the value of the state is

updated by performing actions at the state and observing the rewards. Variations of Q-learning allow us to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning [**?**]. Actor critic DRL algorithims such as DDPG [**?**] and PPO [**?**], can be used effectively in continuous state space environments.

3.3 TRANSFER LEARNING

Transfer learning is inspired by humans applying knowledge learned previously to solve new tasks in a shorter amount of time and in an improved manner in comparison to approaching the same task from scratch [**?**]. In the computer science domain, Transfer Learning is the process of using knowledge learnt in source domains in order to augment training and model performance in a similar target domain [**?**].

TL in Deep Learning(DL) has been successfully applied to obtain state of the art performance on a smaller target dataset by transferring the knowledge from a large source dataset [**?**]. TL in DL has been found to be resilient to the extent that even if the source domains and target domains are quite distant, the performance obtained by TL models outperforms the approach which uses random weight initialization [**?**].

TL in RL is more rigid in contrast to the DL equivalents; in RL based algorithms, the agent's representaion of the world(its sensors and actuators) need to remain the same with flexibility in the state and action space of the agent. In cases where the semantics of the actions vary across tasks, there is a requirement for a mapping function between the two target tasks [**?**]. Approaches for transfer learning in RL

are of five categories, namely, Reward Shaping, Learning from Demonstrations, Policy Transfer, Inter-task Mapping, and Representational Transfer [**?**]. Recent research has shown that performance of TL models can be improved by decoupling the processes of learning the domain and the reward function [**?**].

3.4 ADVERSARIAL ATTACK & TRAINING

Adversarial attacks were first investigated in [**?**] wherein the impact of adding noise to image data to misclassify the images was studied. Initial methods to make models resilient to these attacks by adding Gaussian Blur and AutoEncoders to negate some impact of adversarial noise was studied by [**?**]. Other methods for Adversarial defence were studied by performing image transformations [**?**].

Adversarial Training [**?**] also serves as a good form for defence and is resilient against multiple forms of attacks. Reinforcement Learning models, like Deep Neural Networks are impacted by adversarial attacks [**?**]. The RL agents can be manipulated by using Fast Gradient Sign Method(FGSM) [**?**] to perturb observation frames in a Deep Reinforcement Learning setting or dealing with an optimal adversarial agent, trained to drive the system into suboptimal states [**?**]. Defending RL models from Adversarial Attacks is best done by Adversarial Training methods like [**?**] which makes use of Stochastic Gradient Langevin Dynamics(SGLD).

3.5 UNITY ML AGENTS

The ML agents toolkit is an open source package in unity that provides an intuitive and efficient way to create environments in unity game engine. The ML agents toolkit in unity is used to create 2D and 3D environments that enable reinforcement learning. Previous work with the ML agents toolkit involves creating a single lane track for object avoidance[**?**], and training actor critic algorithms on a third person shooter gaming environment[**?**].

CHAPTER 4

SYSTEM ARCHITECTURE

This chapter describes the overall architecture of the project. The first section(**??**) covers the custom environment that we have developed using Unity game Engine for conducting our experiment. We then describe our the agent(**??**) wherein the agent, the methods by which the agent perceives the environment (**??**), obtains the rewards and takes decisions using the PPO algorithm are discussed . The penultimate section(**??**) introduces the Transfer Learning process by which the knowledge learnt by the agent in one track is transferred to another track in our environment. Finally, we cover adversarial attacks(**??**) on the action space and the observation space of the agent and performing Transfer Learning using Adversarial Training as a precursor.

4.1 ENVIRONMENT

To perform Deep Reinforcement learning we have designed and created our own custom environment in Unity game engine. We have used a package called MLagents developed specifically for Unity that allows us to run our reinforcement learning algorithms in the environment.

We have designed 5 tracks of varying complexity to measure model performance in environments having the same action space. We shall further explore the different tracks in the upcoming chapters.

Building the environment from scratch, allows us to have greater control over the different events that may occur and influence the agent. All the environments have similar events that are used to return a scalar reward to the agent.

The tracks are built using 4 major components : the sidewall(3), the checkpoints(2), the agent(1) and the ray perception sensors(4) as seen in figure **??**

There are numerous checkpoints along the tracks that are placed approximately equidistant to one another. These checkpoints are used to encourage the agent to go in the right direction, and to measure how far the agent has progressed in a track. These invisible checkpoints have an event associated with them that enables the agent to detect them when a collision occurs. When this event is fired, we check if the current checkpoint is the correct checkpoint or not and appropriately provide a reward.

The sidewalls are solid game objects that are used to make sure the agent/car doesn't veer off track. When the agent collides with the sidewall, a function is triggered and a negative reward is sent to the agent. The ray-perception sensors are sensory beams originating from the car/agent, that allow the agent to detect the other game objects and components in the environment. The interactions between these 4 core components enable us to set our scalar rewards, that guide the agent and provide the essential logic required for training.

4.1.1 REWARD SCHEME

The reward scheme is crucial to enable the agent to train for the task. Rewards need to be provided at a constant interval when the agent is performing as

expected in order to train faster and similarly negative rewards need to be provided immediately when performance is unexpected in order to train the agent faster. By experimenting with various permutations, we have arrived at the following reward scheme for the agent:

1. **At each step**: A *small negative* reward is provided to incentivize the agent to travel quicker around the track.

2. **OnWallCollision:** A *moderate negative* reward is provided to discourage agent from hitting the walls on the track.

3. **OnWallCollisionStay:** The difference between collision stay and collision is that once an object has collided in Unity, if it is still in contact with the object post collision, a separate event called OnCollisionStay is triggered and hence a *small negative* reward is provided to prohibit agent from using the walls to perform its turning actions.

4. **CorrectCheckpoint:** The car has managed to make some progress in the track and hence to reward its progress, a *positive* reward is provided.

5. **IncorrectCheckpoint:** The car is moving in the wrong direction on the track and hence to prevent it from continuing to do so, we provide a *negative* reward when it passes through an incorrect checkpoint.

6. **LapCompleted:** The car has completed a lap successfully around the track and hence a *large positive* reward is provided to the agent.

4.2 AGENT

4.2.1 Actor-Critic Methods

Actor Critic Methods are a subclass of Reinforcement Learning algorithms. They consist of a "Critic" and an "Actor" entities during the training process. The "Critic" estimates the value function, ie, the value or predicted reward of a given state. This value function could be a state or action value function. The "Actor" in turn, makes use of the Critic's output in order to update the policy distribution, and takes action based on the current policy. The samples generated by the Actor in turn, are used to update the value estimates of the Critic. Both the Actor and Critic are function approximators, generally with neural networks.

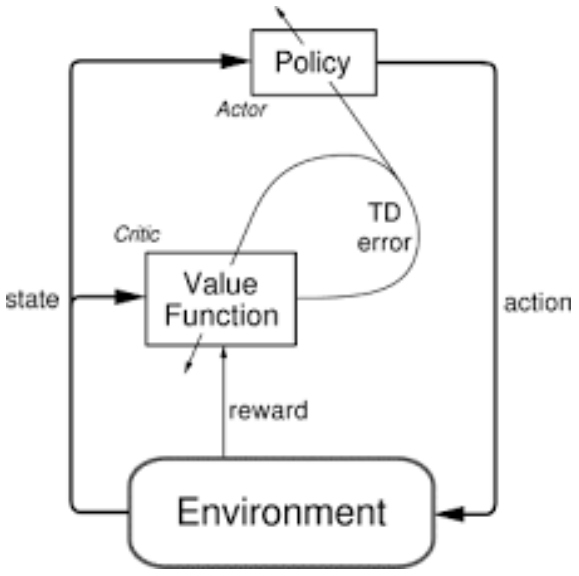

FIGURE 4.1: Actor Critic Methods

4.2.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is an RL algorithm that falls under the actor-critic class of methods. It is a policy gradient method, ie, it uses a function approximator to represent the policy and performs gradient ascent on it to obtain the maximum reward. The main advantage of PPO over vanilla policy gradient or actor-critic algorithms is the fact that PPO clips the updates made to its value function to a certain extent. By clipping its updates, PPO ensures that the policy doesn't go too far in the wrong direction and ruin its chances of finding the optimal solution.

In order to understand how PPO performs this clipping, we need to understand a few key terms first. The first term is $r(\theta)$. $r(\theta)$ is defined as the probability ratio of an action under the current policy and the action under the previous policy. If a particular action becomes more likely to be taken by the current policy, the value of $r(\theta)$ will be greater than 1. If a particular action becomes less likely to be taken by the current policy, the value of $r(\theta)$ will be between 0 and 1.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}, so\ r(\theta_{old}) = 1 \tag{4.1}$$

The second term is the advantage $\hat{A}_t$. The advantage $\hat{A}_t$ is an estimate of the relative value of the selected action in the current state. This advantage is used to estimate how good a particular action is compared to the average action in that state. The advantage is used to reduce variance in the updates made to the value function. If the advantage function for an action is positive, it means that action is good and will generate better rewards in the future. Therefore, we should increase the

probability of picking that particular action in the future. On the other hand, if the advantage function for an action is negative, the probability of picking that action should be reduced.

$$A(s,a) = Q(s,a) - V(s) \tag{4.2}$$

The general method in which policy-gradient methods work is by optimizing a policy loss function. Optimizing this function for reward can theoretically lead to an optimized function, but it is often quite difficult due to the nature of the samples. During the training process, the neural network representing our value function is constantly being updated, therefore our values are quite noisy and may not be very accurate.

PPO aims to solve this issue by making use of both of these terms, ie, $r(\theta)$ and $\hat{A}_t$, to generate a unique loss function called the Clipped Surrogate Objective. Clipped Surrogate Objective computes the minimum of two terms: the normal PG(Policy gradient) objective and clipped PG objective. The key component in PPO is the 2nd term, which clips the normal PG objective between $1+\varepsilon$ and $1-\varepsilon$. The effects of this clipping can be captured in the below graphs.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)] \tag{4.3}$$

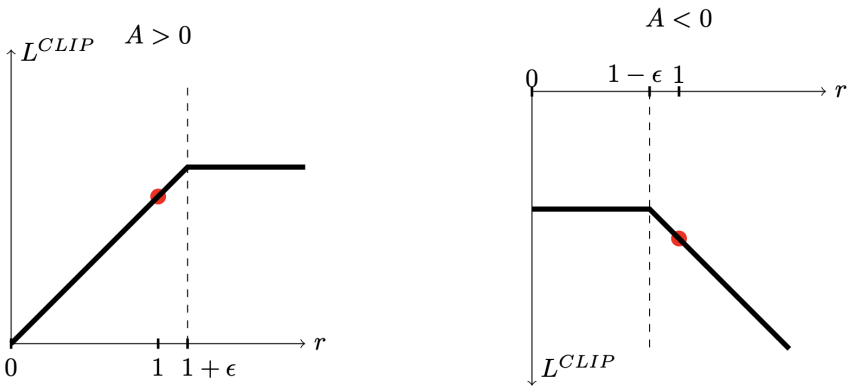

FIGURE 4.2: Clipped Loss Graph

On the left side, we see the scenario where the advantage is positive, i.e., the action taken by the agent had a better-than-expected return. With increasing values of r, the value of the L^{CLIP} also increases, but only to a certain point. This prevents the algorithm from taking an update too far in the wrong direction. A similar scenario takes place on the right side as well. The algorithm is prevented from over-correcting its mistakes.

PPO combines this loss function with two other terms, namely L^{VF} and S. L^{VF} is the mean squared error of the value function that is in charge of updating the network. S is a measure of entropy, which helps the agent act spontaneously in the initial stages of training, which can help the agent experiment and find optimal actions.

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{4.4}$$

4.2.3 ALGORITHM: PPO

Algorithm 1 Proximal Policy Optimization (PPO)

for iteration = 1,2,... **do**

for actor = 1,2,...,N **do**

Run policy $\pi_{\theta_{old}}$ in the environment for T timesteps

Compute advantage estimates $\hat{A}_1,\ldots,\hat{A}_T$

end for

Optimize surroage L wrt θ, with K epochs and minibatch size $M \leq NT$

$\theta_{old} \leftarrow \theta$

end for

4.3 AGENT INPUTS

The agent in our environment is the Car(1) itself. The car makes decisions regarding acceleration deceleration and steering angle based on the inputs that it receives. The two major types of inputs that the agent perceives from the environment are the current speed and the Ray Perception Senor data(4) from the sensors placed on the car. The Ray Perception inputs inform the car of the distance of the object, what kind of object it is and also the angle from the car at which the object is located. For example, the object could be a Checkpoint(2) or a SideWall(3). The car makes use of this information to decide when to accelerate, brake, or turn.

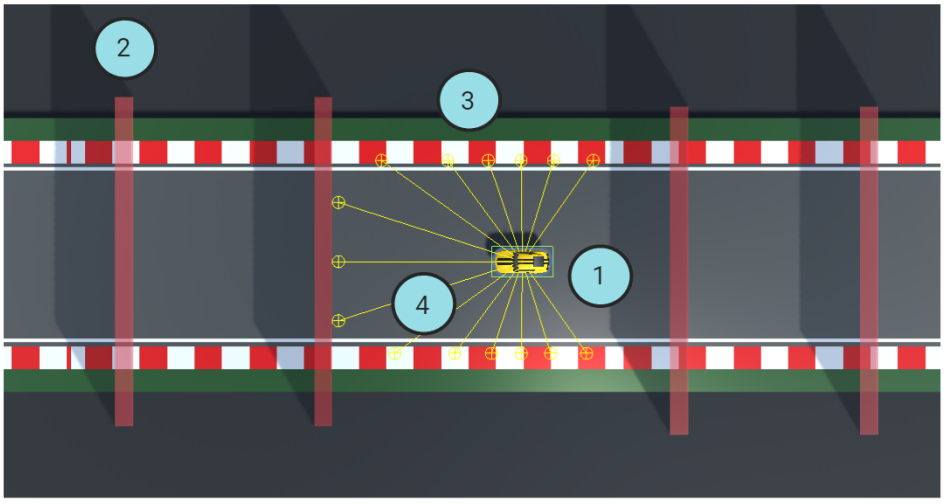

FIGURE 4.3: Agent Inputs

4.4 TRANSFER LEARNING

Upon training the models on the basic tracks developed using the PPO Algorithm, we then move on to creating complex tracks which we cover in section **??** for the purposes of studying the results obtained by using Transfer Learning in our environment.

We study the performance obtained by TL when the similarity is high (intra basic track) and similarity is lower (basic track - complext track). It is to be noted that the latter is more dissimilar only when compared to the former and that in both cases the source task and the target task have the same task structure, satisfying the requirement to perform TL in RL [**?**]. We peform these experiments to understand the difficulties associated with using TL in RL to leverage learned representation.

To measure the performance of TL in RL in our environment, we measure the cumulative reward obtained at the end of fixed amount of steps on the target task and compare it with the cumulative reward obtained by training for the same

number of steps on the target task with no prior knowledge. We also monitor the variance in episode length as it is an indicator of the stability of the agent on the track.

4.5 ADVERSARIAL TRAINING

Finally, we perform Adversarial Attack and Adversarial Training in our environment using noise based attacks on the observations(**??**) and on the action space (**??**).

4.5.1 Attack on Observation

The observations perceived by the agent are the current speed of the car and the distance and angle at which the checkpoints and walls of the track are located at. For the purposes of attacking the observation space, we add noise to the checkpoint positioning resulting in sub-optimal checkpoint orientation on track. Upon passing through a 'attacked checkpoint', we attack the successive checkpoint and finally all checkpoints are reset at the time of lap completion.

Algorithm 2 Adversarial Attack on Checkpoints

Require: checkpointAtttackFlag = True

checkpointList[] ← track.getCheckpointList()
nextCheckpoint = 0
while lapCompleted = False **do**
X_{disp} ← Random.RandInt(-4, 4)
Y_{disp} ← Random.RandInt(-4, 4)
checkpointList[nextCheckpoint].position += (X_{disp}, Y_{disp})

nextCheckpoint = (nextCheckpoint + 1) % totalCheckpoints

if nextCheckpoint = 0 **then** ▷ All checkpoints passed
lapCompleted ← True
reset positions of all checkpoints
end if
end while

4.5.2 Attack on Action Space

In this attack, we randomly pick actions ignoring the output provided by the policy of the agent during a fraction of time steps determined by a parameter called *actionAdversaryThreshold*.

Algorithm 3 Adversarial Attack on Actions

Require: actionAdversaryThreshold ≥ 0, policy π

action $\leftarrow \pi$.getAction()

forward = action[0]

turn = action[1]

if Random.Rand() $\geq$ actionAdversaryThreshold **then**

$forward \leftarrow$ Random.RandInt(-1, 2)

$turn \leftarrow$ Random.RandInt(-1, 2)

end if

car.forward = forward

car.turn = turn

We perform experiments and train agents on all the basic and complex tracks using both the attacks as a precursor and repeat the experiments performed for TL in **??** and study the performance of TL obtained using AT as a precursor and in the absence of AT.

CHAPTER 5

EXPERIMENTAL SETUP

In this chapter we discuss the method used to measure the complexity of any given race track following which we move on to describe tracks developed for performing our experiments using the custom environment discussed in Chapter **??**. We have developed 5 different types of tracks in two levels of complexity. The first level of complexity is called 'Basic Tracks' . The second level of tracks are inspired by AWS Deepracer [**?**] challenge tracks and are inspired by real world race tracks with much higher level of complexity and are labelled 'Complex Tracks'.

5.1 TRACK COMPLEXITY MEASURE

Before proceeding with the Basic and Complex Tracks, we will introduce the features used to quantify the complexity of a track. Features indicating the complexity of the track in decreasing order of complexity are as follows:

1. Chicanes
2. Sharp Turns
3. Sweeping Turns
4. Distance of Track

5.1.1 Chicanes

Chicanes refer to a tight sequence of corners in alternate directions. They are usually inserted into a circuit to slow the cars, often just before a high-speed corner [**?**]. These are the most complex for the agent to navigate due to slow speed required and precise control of the turn inputs in order to pass the chicane without hitting the sidewalls of the track. The leeway for error is the least when navigating a chicane due to this nature. Figure **??** depicts an example of a chicane.

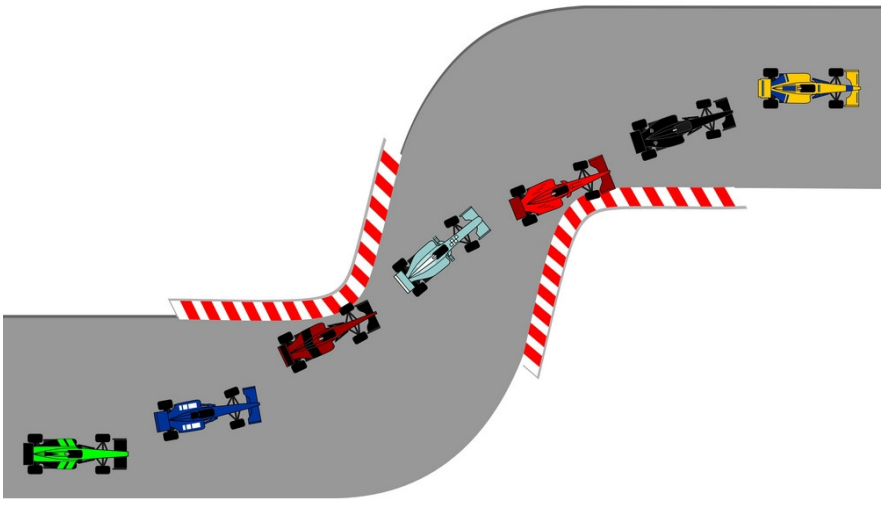

FIGURE 5.1: Chicane example (Image from Vector Stock)

5.1.2 Sharp Turns & Sweeping Turns

The difference betwen Sharp Turns and Sweeping Turns is the radius of the turn. Turns with a lower radius are more difficult to navigate and the average speed of the car(agent) is lower and these are called sharp turns. Sweeping turns are turns having a larger radius of curvature allowing for faster speeds and minimal

steering input allowing for more leeway for errors in the inputs. Figure **??** shows the different types of turns.

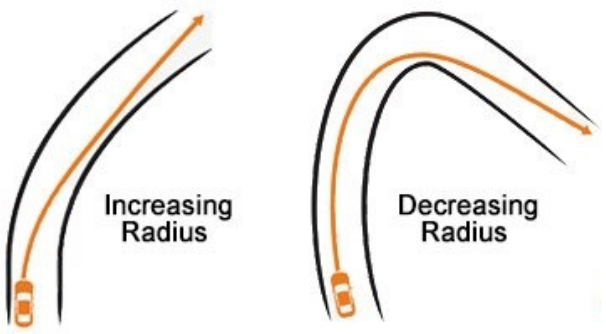

FIGURE 5.2: Sweeping vs Sharp Turns (Image from Go-Kart-Source)

5.1.3 Distance

Distance of the track is the least important feature for track complexity. For the same given distance, the presence of more chicanes and/or turns determines the real world complexity in navigating the track. Thus, distance of the track is the least important feature to determine the complexity of a track. However, it is to noted that longer the distance, the chances for making a mistake increases due to the time taken to complete a lap around the track.

5.1.4 Computing Complexity

In order to quantify the complexity of a track, we use a custom metric given by

$$Complexity = (4 \times Chicanes) + (2 \times SharpTurns) + SweepingTurns \qquad (5.1)$$

The complexity of all the tracks developed in our custom environment are provided in table **??**.

5.2 BASIC TRACKS

In this section we cover the basic tracks we developed as a part of Phase I of our experimentation. We developed the Oval track, shown in figure **??**, which is the simplest of the tracks in terms of layout as a proof of concept tracks to ensure all our supporting systems like the checkpoint-ing systems were working as expected in our custom built environment. To test the the agent behaviour when more complicated maneuvers are required, we developed the One Kink Track and Two Kink track as shown in figures **??** & **??**.

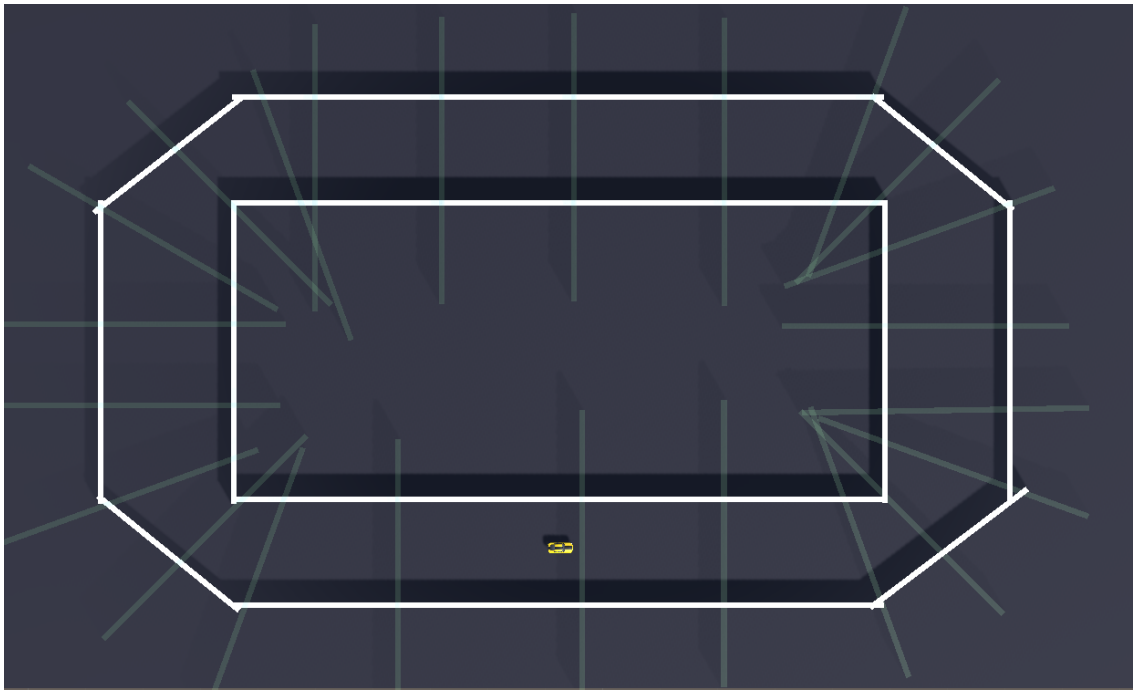

FIGURE 5.3: Oval Track

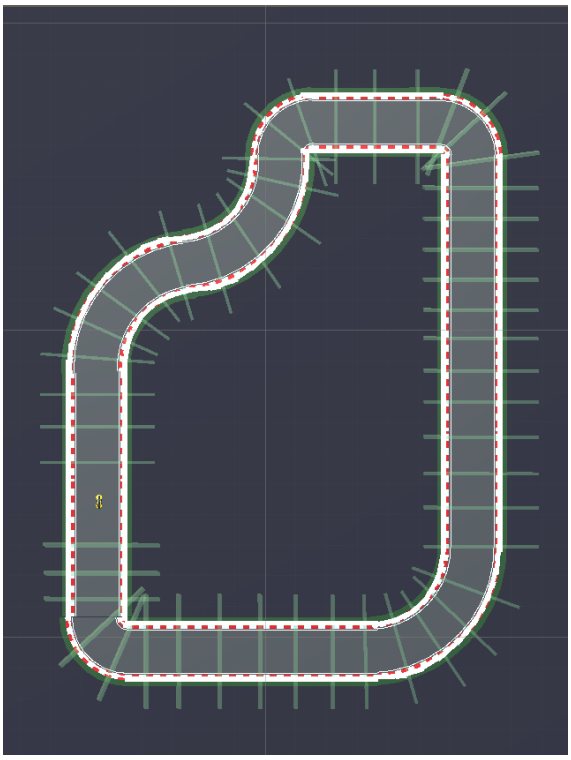

FIGURE 5.4: One Kink Track

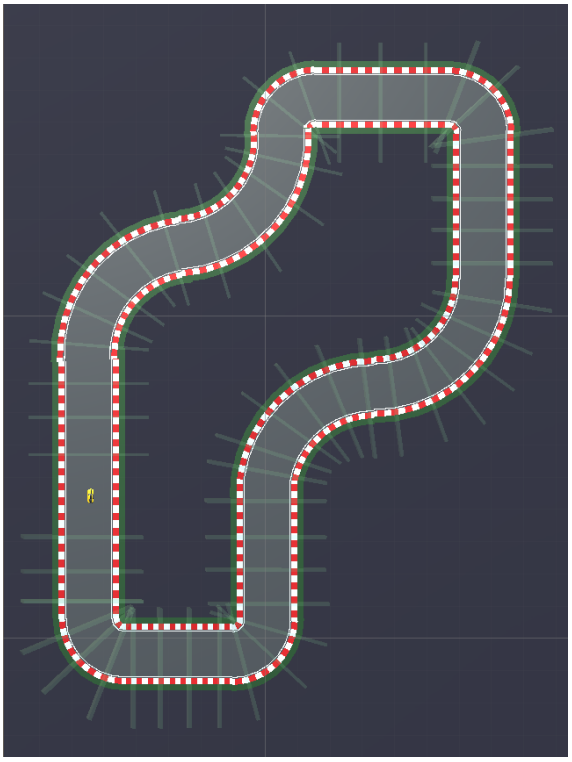

FIGURE 5.5: Two Kink Track

5.3 COMPLEX TRACKS

For developing 'complex tracks', we have used a real world Formula 1 race track of barcelona and a track used as a part of AWS deepracer challenge namely AWS Asia Pacific Track to serve as inspirations to test the agent performance in tracks closer to real world. These tracks are longer and feature more complicated track layout in comparison to the basic tracks discussed in section **??**.

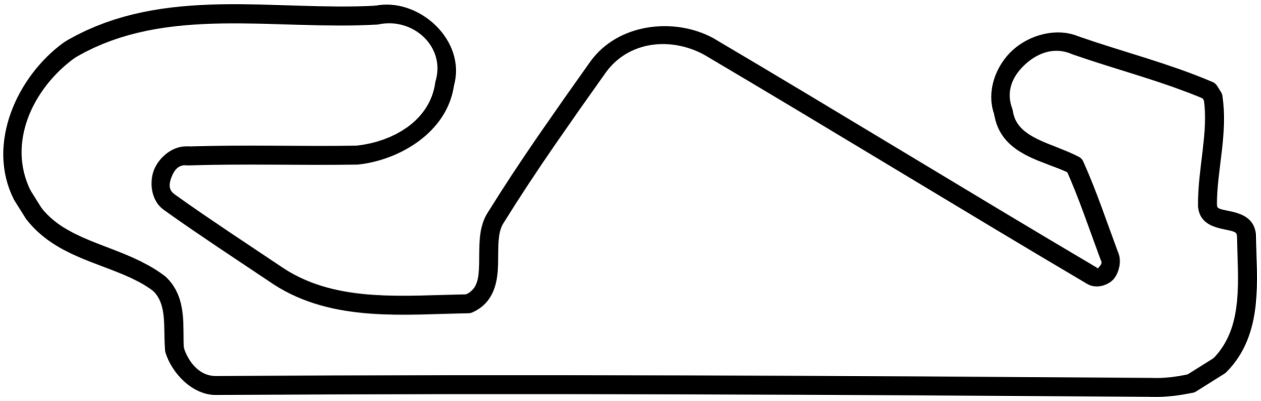

FIGURE 5.6: Original Barcelona Racerack (Image from Clipartmax)

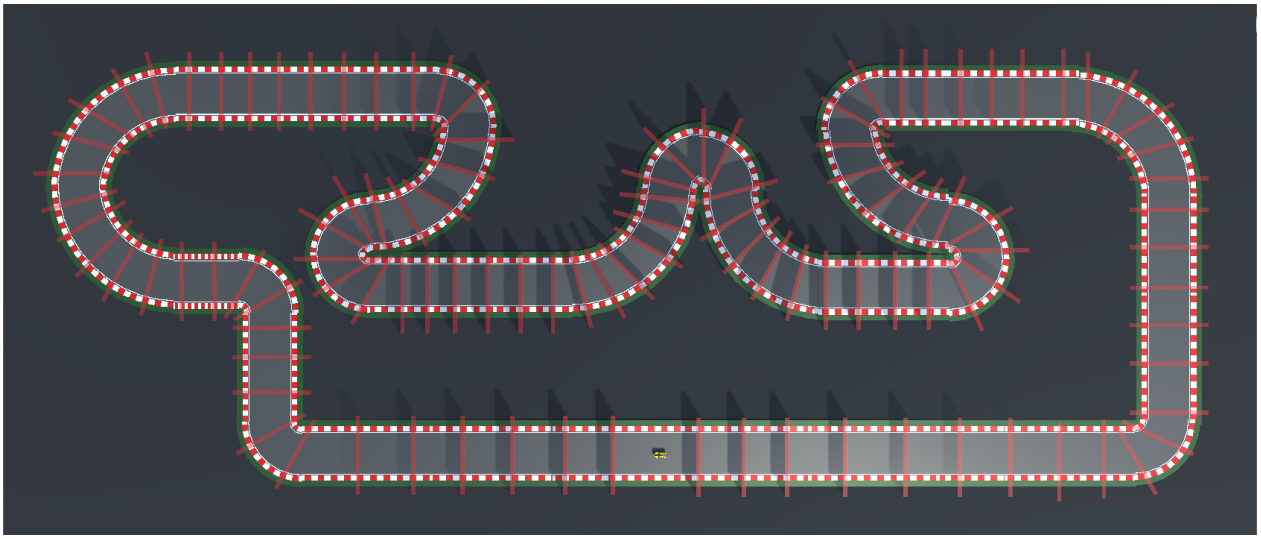

FIGURE 5.7: Barcelona Track in Unity

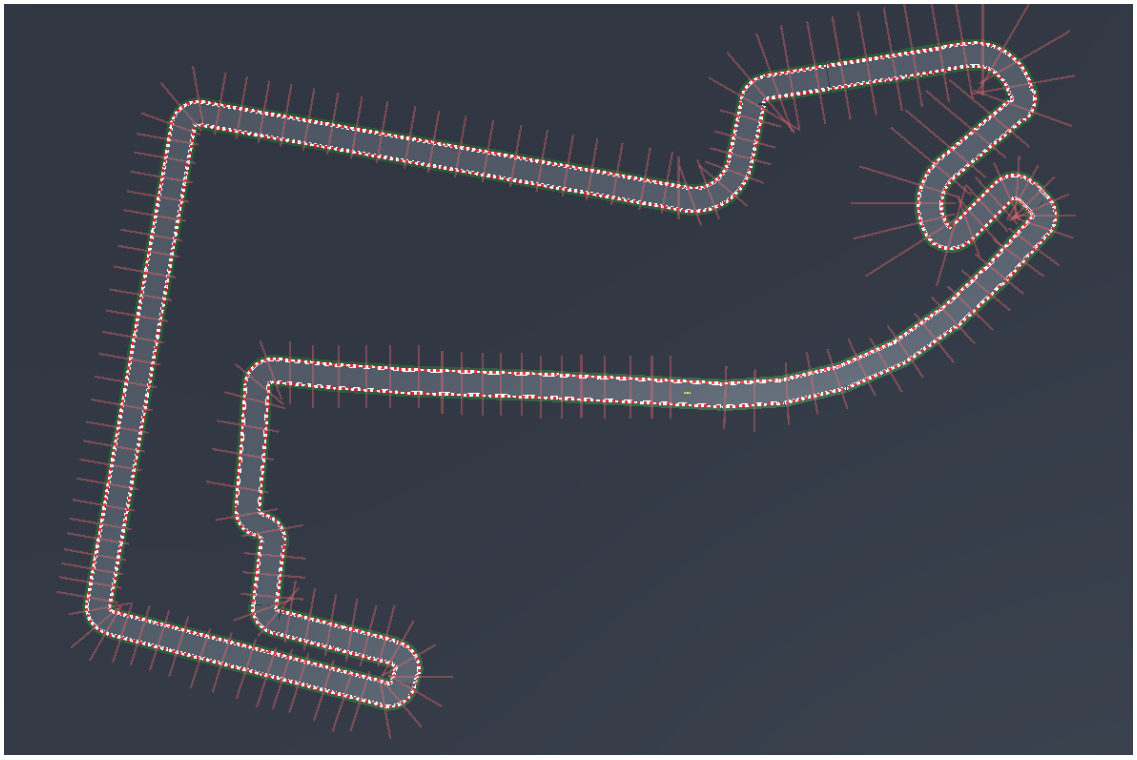

FIGURE 5.8: AWS Asia Pacific Track

Type	**Track**	**Chicanes (C)**	**Sharp Turns (SH)**	**Sweeping Turns (SW)**	**Complexity (4C+2SH+SW)**
Basic	Oval	0	4	0	8
	One Kink	1	3	1	11
	Two Kink	2	4	0	16
Complex	Barcelona	0	8	6	22
	AWS Track	1	8	4	24

TABLE 5.1: Complexity of the various tracks developed

5.4 ML AGENTS

MLagents is an open source package that provides a framework to train and develop intelligent agents in the Unity game engine environment. It supports multiple methods of training intelligent agents using reinforcement learning, imitation learning, and other machine learning methods through the Python API provided. ML Agent contains 4 major components as shown in image **??**

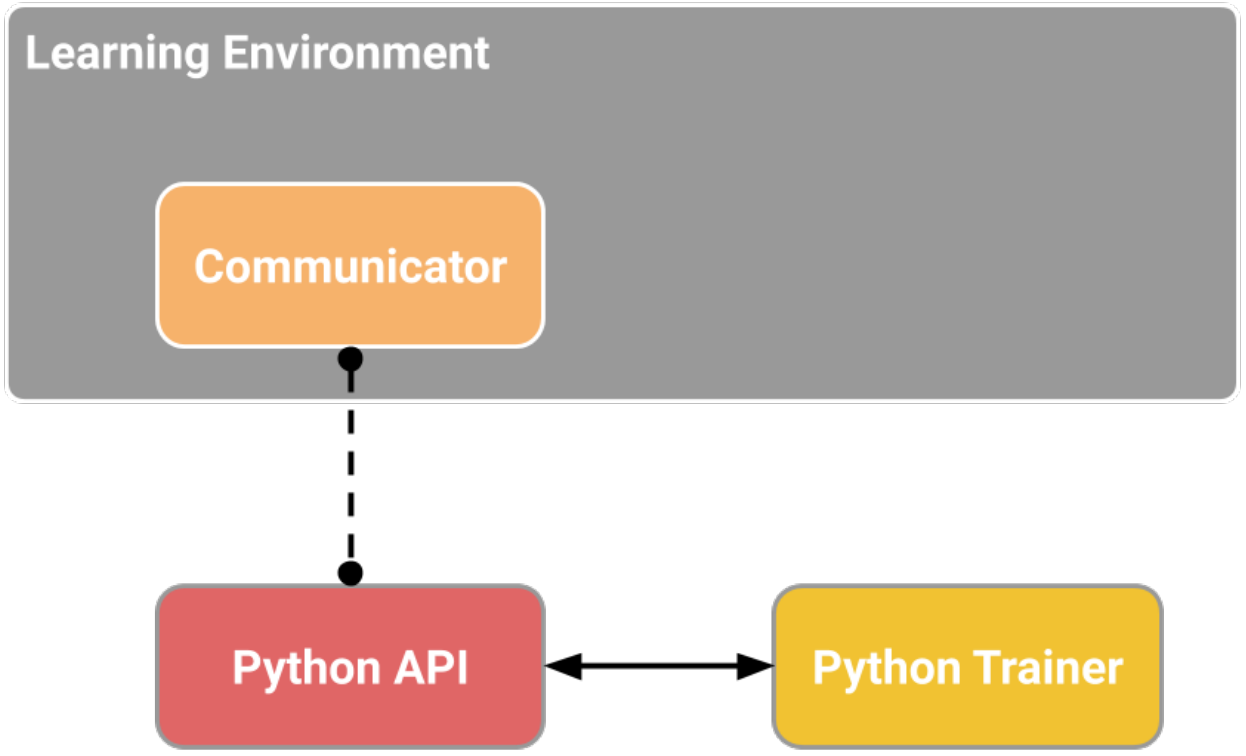

FIGURE 5.9: Components of ML Agents Framework (Image from ML Agents Github)

- **Learning Environment:** Refers to the Unity scene created for experimentation. This scene serves as the environment for the agent to interact with and receive observations and take actions in.
- **External Communicator:** Feeds the data from the Unity scene and passes it on to the python API

- **Python API:** Provides for a method to directly access the Learning Environment from the external python code to interact and manipulate the learning environment

- **Python Trainer:** Is the place where the Algorithm used to train the agent is located. The algorithm has access only to the Python API and are written using PyTorch [**?**]

5.5 PPO NETWORK & HYPER-PARAMETERS

The algorithm we use to train our agents in the environment is the PPO Algorithm covered in section **??**. PPO Contains two function approximators representing the actor and the critic respectively and these are developed using the neural networks as shown below in image **??**. Both the networks consist of 3 layers with 256 hidden units present in each of the layers. The Actor network returns the action probabilities and the Critic returns the value function.

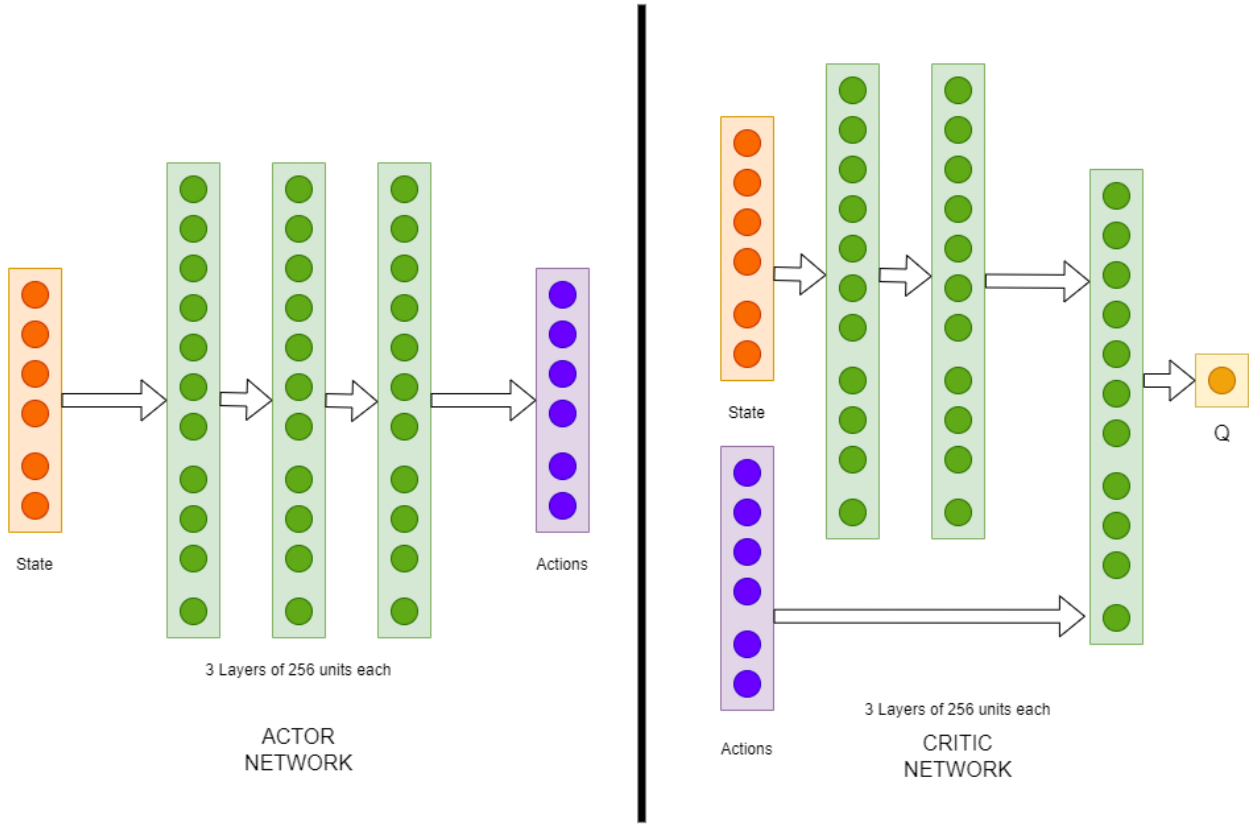

FIGURE 5.10: PPO Network Architecture

All the models were trained for 8-12 million steps depending on complexity. A batch size of 1024 was used with a buffer which stores experiences before making updates of the size 10240. All the agents were trained using 3 passes of the experience buffer to improve the stability of the training. Training episode ends when the reward accumulated becomes negative ($R0$) or if the agent successfully completed a lap around track. At episode completion, the progress of the agent is reset on the track and the agent is reset at start location with 0 initial velocity. We use an initial learning rate of 3×10^{-2} with a linear learning rate scheduler to ensure that the updates are not to large with the progress.

CHAPTER 6

PERFORMANCE ANALYSIS

In this chapter we discuss the results obtained across the 3 phases of our experimentation done as a part of this project.

- **Phase 1** Covers the baseline scores obtained on all tracks
- **Phase 2** Covers the results obtained by performing Transfer Learning between tracks within the same level and across levels.
- **Phase 3** The final phase of our project covers the result obtained by training the agents in the presence of the adversaries we have devised and the results obtained by transferring the knowledge learnt from these models.

For all these phases, we present the graphs obtained during the training of our models. The first graphs shows the cumulative reward obtained during the training process and our objective is to *maximise* the reward obtained. The second graphs shows the average length of an episode of agent as the training progresses. Having a lower value is *better* but is not mandatory as this is an indicator of how stable the agent is with sudden dips indicating crashes on the track.

6.1 PHASE I - BASELINE PERFORMANCE

6.1.1 Oval Track - Basic Track

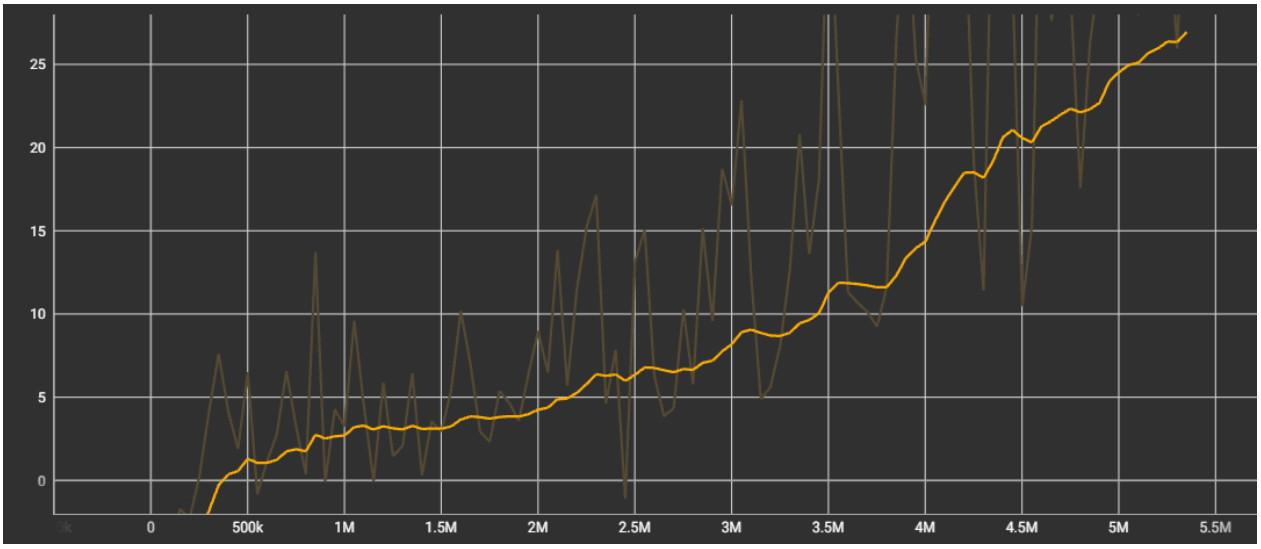

FIGURE 6.1: Oval Track Cumulative Reward.
X-axis: Time steps
Y-axis: Cumulative Reward

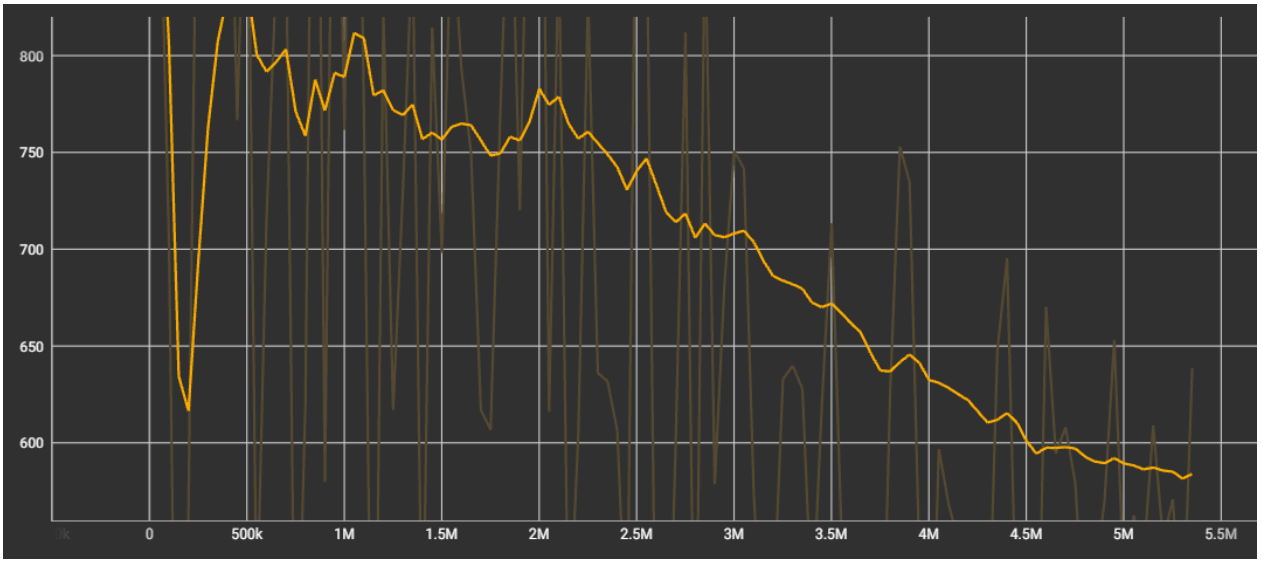

FIGURE 6.2: Oval Track Episode Length.
X-axis: Time steps
Y-axis: Episode Length

6.1.2 One Kink Track

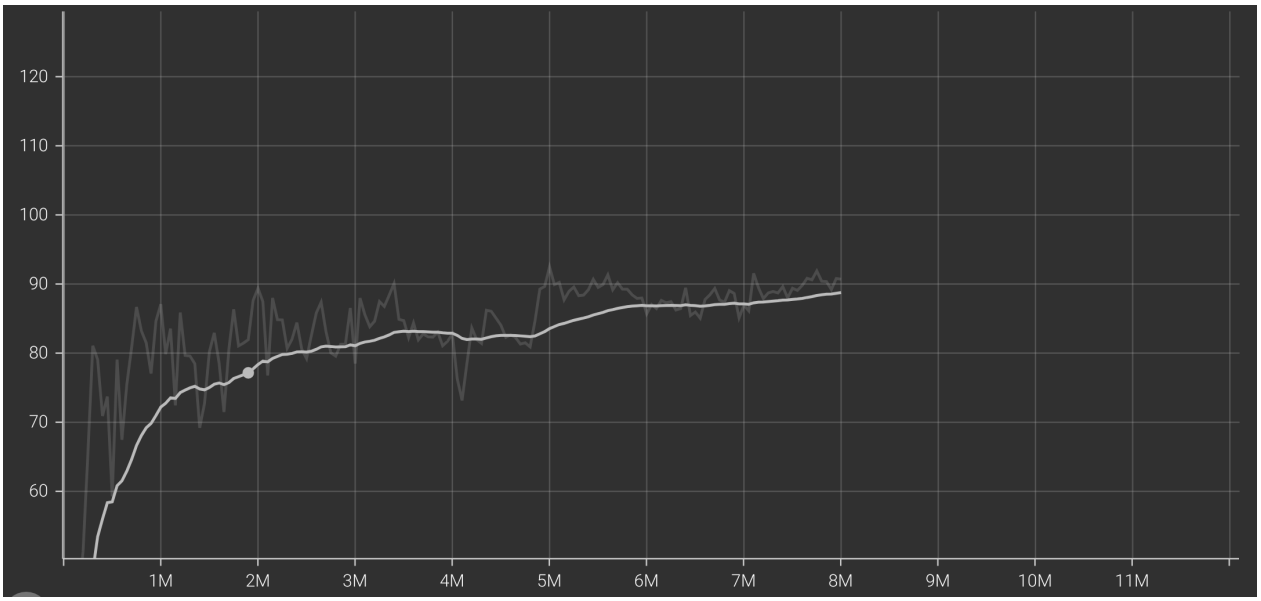

FIGURE 6.3: One Kink Track Cumulative Reward.
X-axis: Time steps
Y-axis: Cumulative Reward

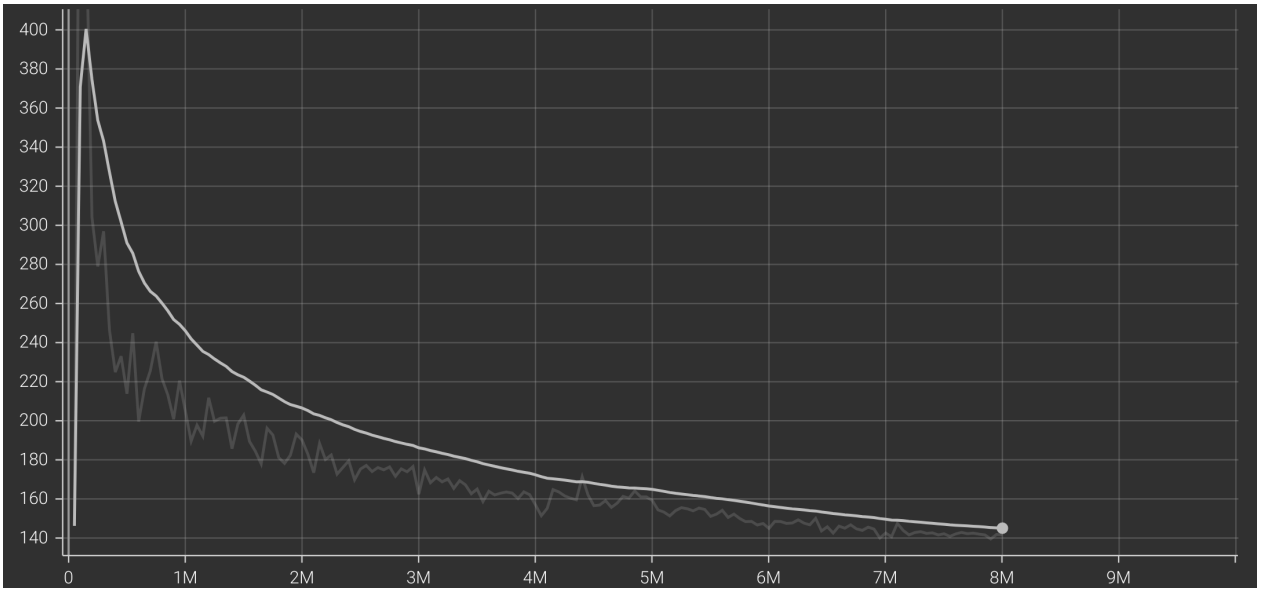

FIGURE 6.4: One Kink Track Episode Length.
X-axis: Time steps
Y-axis: Episode Length

6.1.3 Two Kink Track

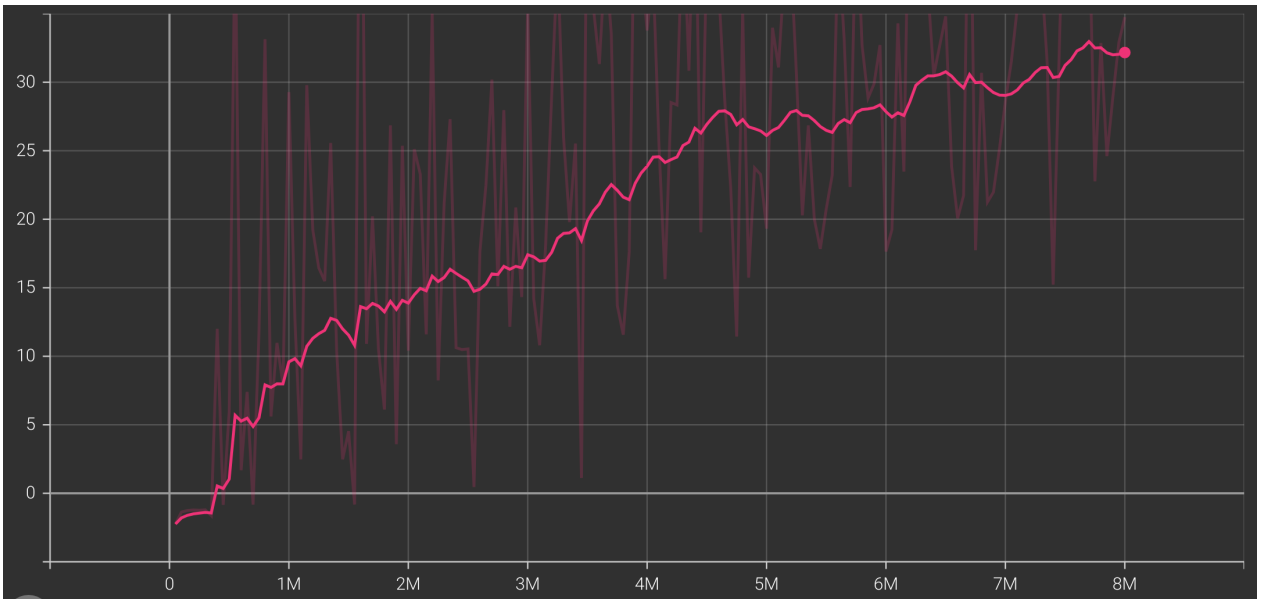

FIGURE 6.5: Two Kink Track Cumulative Reward.
X-axis: Time steps
Y-axis: Cumulative Reward

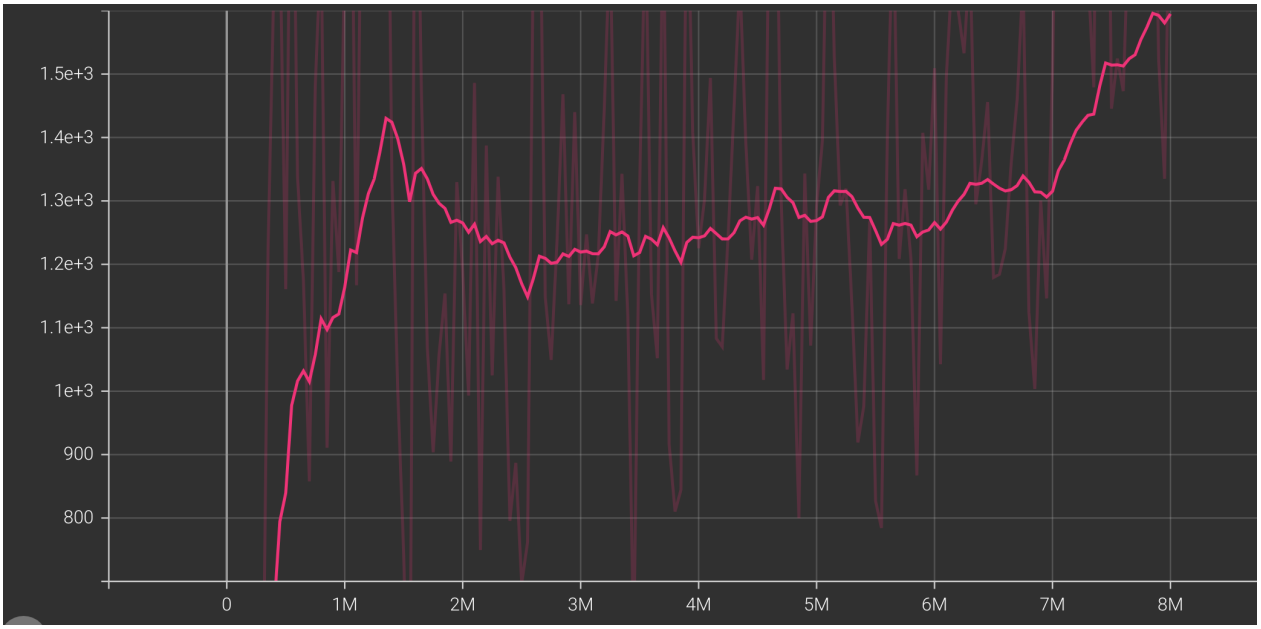

FIGURE 6.6: Two Kink Track Episode Length.
X-axis: Time steps
Y-axis: Episode Length

6.1.4 Barcelona

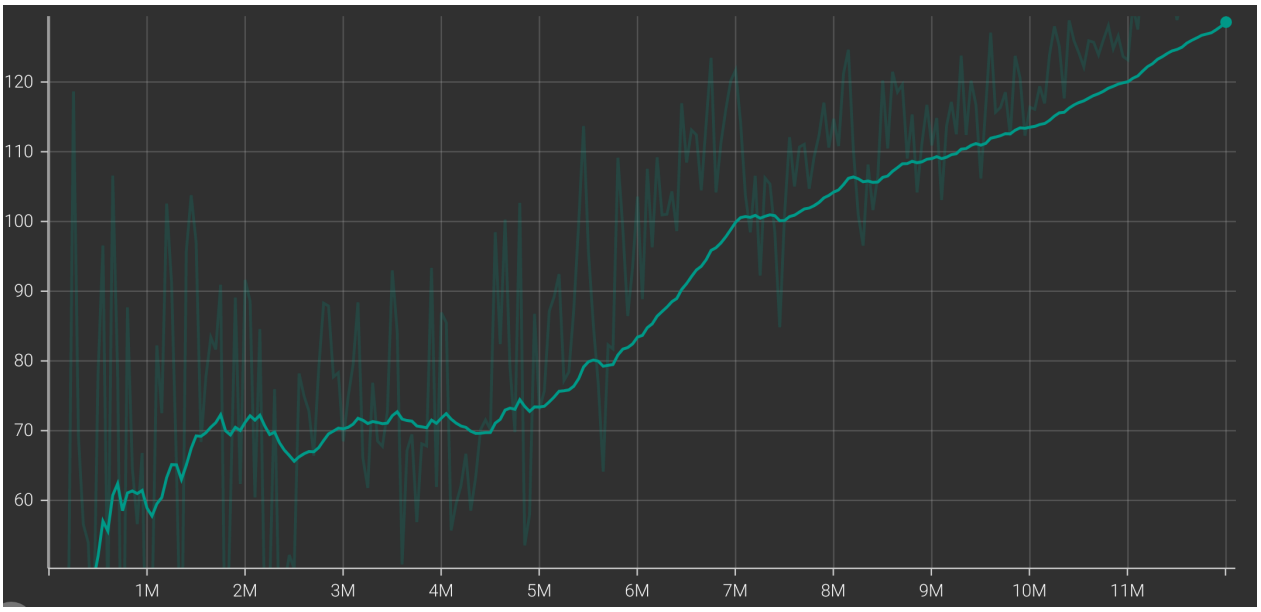

FIGURE 6.7: Barcelona Track Cumulative Reward.
X-axis: Time steps
Y-axis: Cumulative Reward

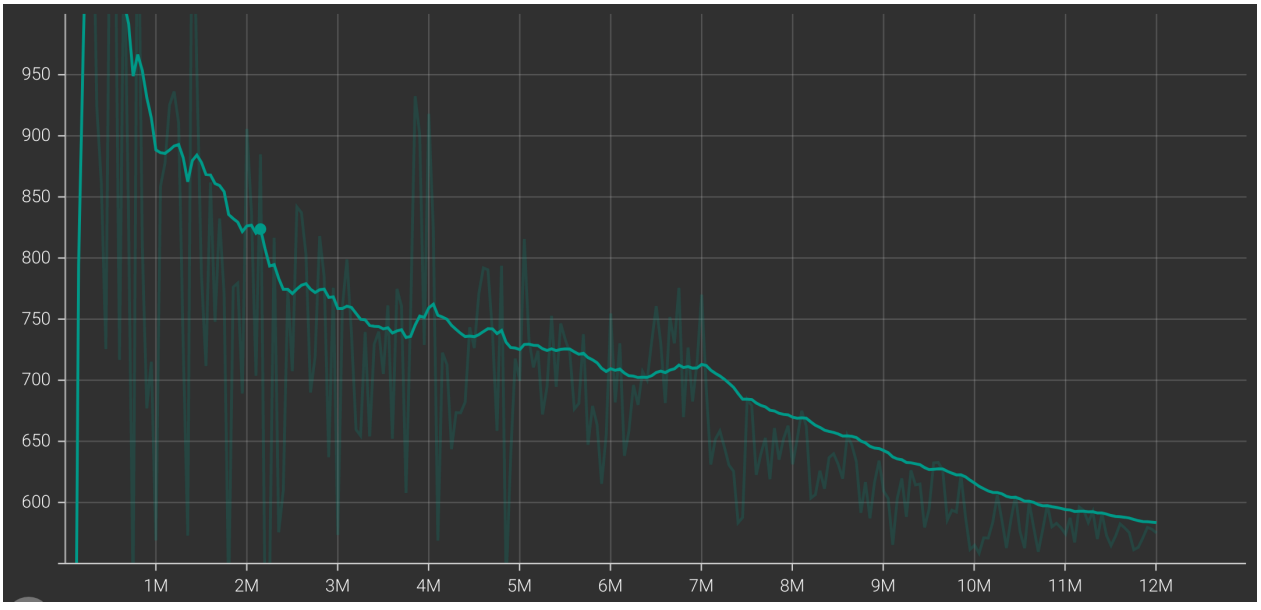

FIGURE 6.8: Barcelona Track Episode Length.
X-axis: Time steps
Y-axis: Epsiode Length

6.1.5 AWS Track

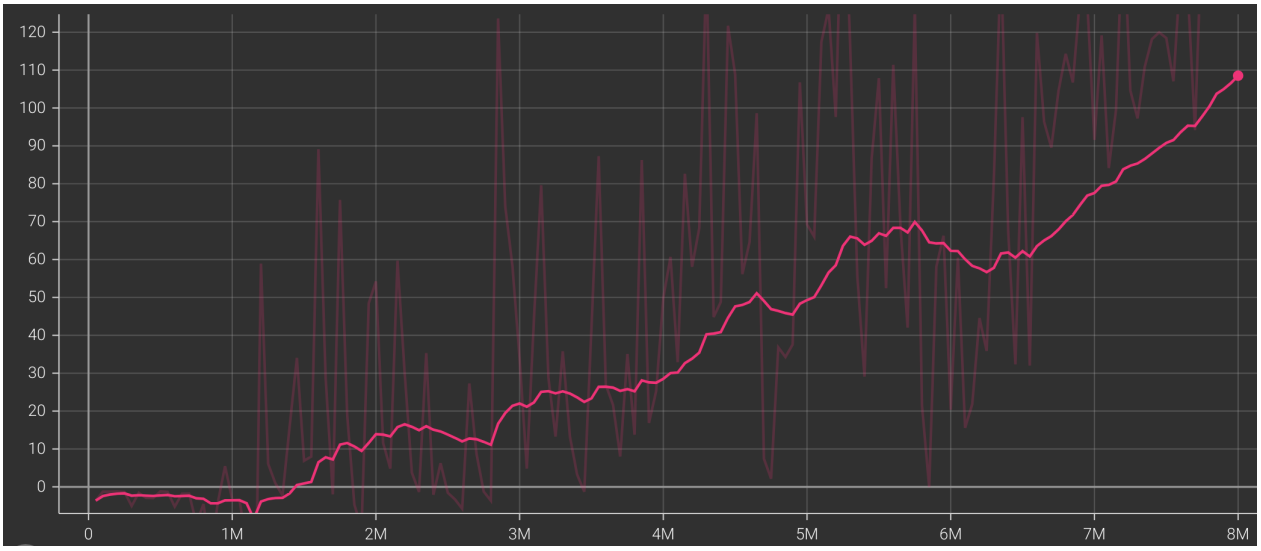

FIGURE 6.9: AWS Track Cumulative Reward.
X-axis: Time steps
Y-axis: Cumulative Reward

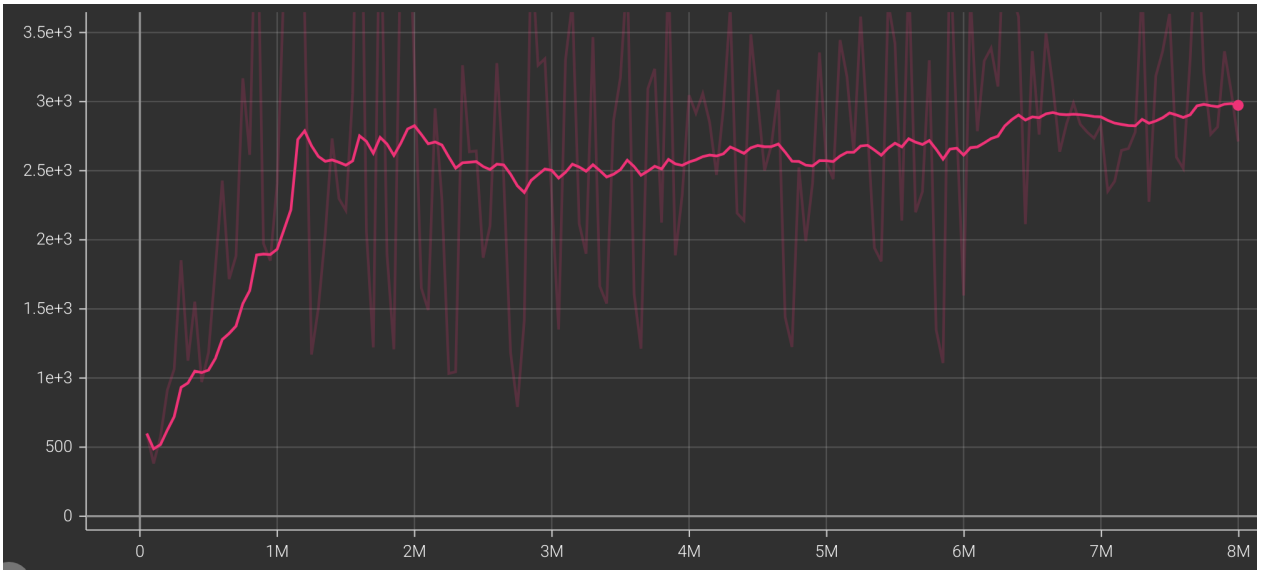

FIGURE 6.10: AWS Track Cumulative Reward.
X-axis: Time steps
Y-axis: Episode Length

In this phase we record the baseline performance of the PPO algorithm on all the tracks that we have developed in table **??**. These scores will be used in the

upcoming sections to quantify the effects and improvement of Transfer Learning and Adversarial Training.

Type	Track	Steps (in Million)	Reward
Basic	Oval Track	5	41.2
	One Kink	8	90.7
	Two Kink	8	34.75
Complex	Barcelona	12	128.6
	AWS Track	8	136.8

TABLE 6.1: Average reward scores of baseline models

6.2 PHASE II - TRANSFER LEARNING

6.2.1 One Kink - Two Kink TL Performance

Here we compare the performance obtained by model that uses transfer learning in comparison to the baseline model of the Two Kink track with the source track (One Kink) and target track (Two Kink) being tracks of the same class of complexity namely ‘Basic Tracks’.

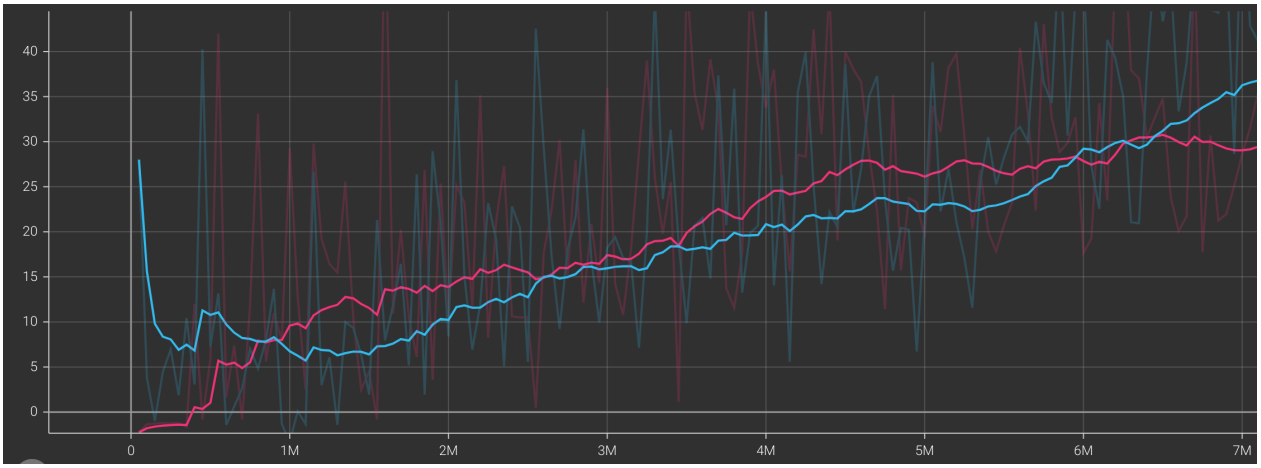

FIGURE 6.11: *Blue*: Transfer Learned model, *Pink*: Baseline model.
X-axis: Time steps
Y-axis: Cumulative Reward

The graph in figure **??** above represents the rewards obtained with Transfer Learning and without Transfer Learning on our Two Kink Track. The Transfer Learned model was trained first on the One Kink track for 1 Million steps and transferred for a further 7 Million steps on the Two Kink Track. Some key observations made are:

- The initial performance of the Transferred Model is higher as it has prior knowledge about the first few steps to be made which are track independent.
- As the training further progresses, the scores obtained are very similar across both the models till the 6 Million step mark
- The Transfer Learned model proceeds to performs better on the target task and achieves 13.1% more reward(36.3 vs 32.1) at the end of TL training.
- While the results are positive, Transfer Learning in RL is still more volatile than in the case of DL and ML. This is further highlighted by the similar performance of the two models.

6.2.2 Two Kink - AWS TL Performance

In this experiment we perform transfer learning between tracks of two levels of complexity. Two Kink track is the most complex 'Basic Track' and AWS track is the most complicated track that we use in our experimentation.

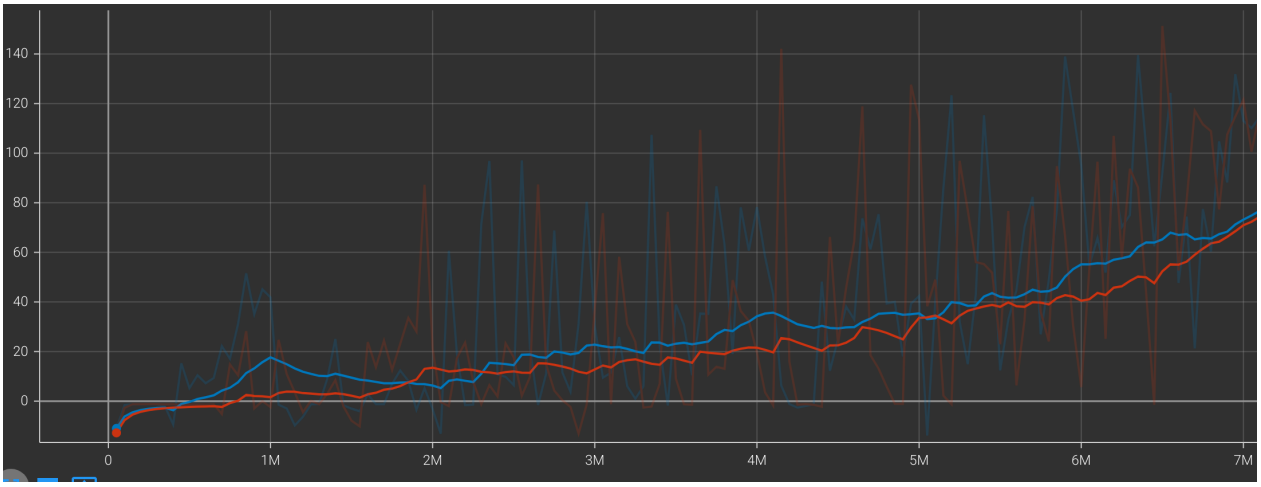

FIGURE 6.12: *Blue*: Transfer Learned model, *Red*: Baseline model.
X-axis: Time steps
Y-axis: Cumulative Reward

The graph shown in figure **??** above represents the rewards obtained with Transfer Learning and without Transfer Learning on our AWS Track. The Transfer Learned model was trained first on the Two Kink track for 1 Million steps and transferred for a further 7 Million steps on the AWS Track. Some key observations made are:

- The performance of the Transfer Learned model is consistently around 5-7% better than the baseline model from the start of training.
- Around the 6.5 Million steps mark, the gap shrinks to a negligible difference with no significant improvement at the end of training

This further fortifies the findings that TL is much more challenging in RL than in DL or ML.

6.3 PHASE III

6.3.1 Transfer Learning after Adversarial Training

In this section we shall explore the effects and results of performing Adversarial Training. We shall discuss its effects on transfer learning between 2 tracks of similar complexity(One Kink track and Two Kink track), and between tracks of varying complexity(Two Kink Track and AWS Track). We are using 2 different types of random noise attacks: Adversarial attack on the Observations(AT-TL) and the Adversarial Attack on the actions(AAT-TL)

6.3.1.1 One Kink to Two Kink

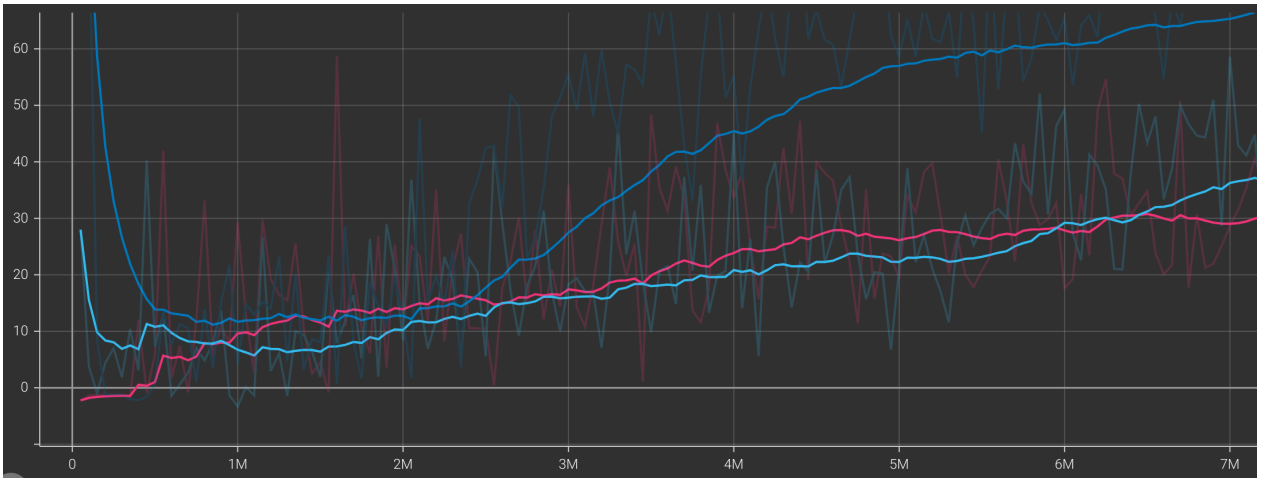

FIGURE 6.13: *Dark-Blue*: Adversarially Trained(Observation Attack) Transfer Learned model, *Light-Blue*: Transfer Learned model, *Red*: Baseline model.
X-axis: Time steps
Y-axis: Cumulative Reward

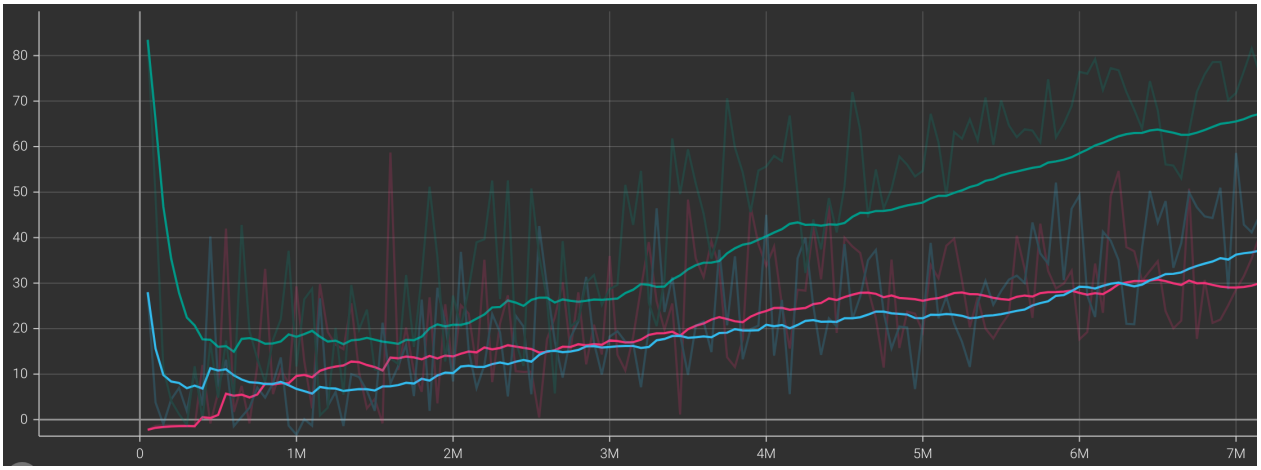

FIGURE 6.14: *Green*: Adversarially trained(Action Attack) transfer learned model, *Light-Blue*: Transfer Learned model, *Red*: Baseline model.
X-axis: Time steps
Y-axis: Cumulative Reward

Various Action Adversary Models were trained using different action adversary thresholds. It was observed that values between 0.2 to 0.5; we see a significant drop in performance when $threshold > 0.6$.

- The AT model reaches an average reward of 65.3 at 7 million steps, which is is significantly better than the Transfer Learning model (36.3).
- This is a 79.9% improvement over the average reward obtained by the simple Transfer learned model
- We also observe that the AAT model is similar in performance to the AT model and reaches a score of 65.5 at the end of 7 million steps.
- This is again a 80.4% improvement over the simple Transfer learned model.

We can see that performing Adversarial Training followed by Transfer learning in both cases seems to be extremely effective; this may be attributed to the similarity

of the layout of the tracks. The agent is able to better learn the state representations when we use adversarial training and is thus performing better after transfer learning.

6.3.1.2 Two Kink to AWS

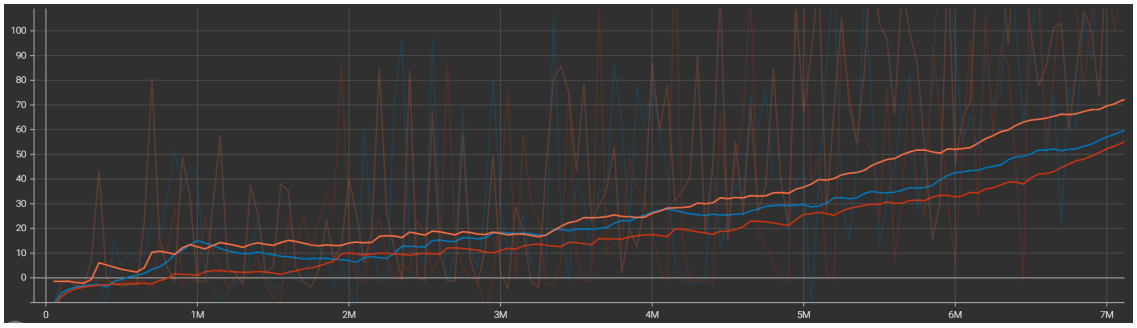

FIGURE 6.15: *Orange*: Adversarially Trained(Observation Attack) Transfer Learned model, *Blue*: Transfer Learned model, *Red*: Baseline model.
X-axis: Time steps
Y-axis: Cumulative Reward

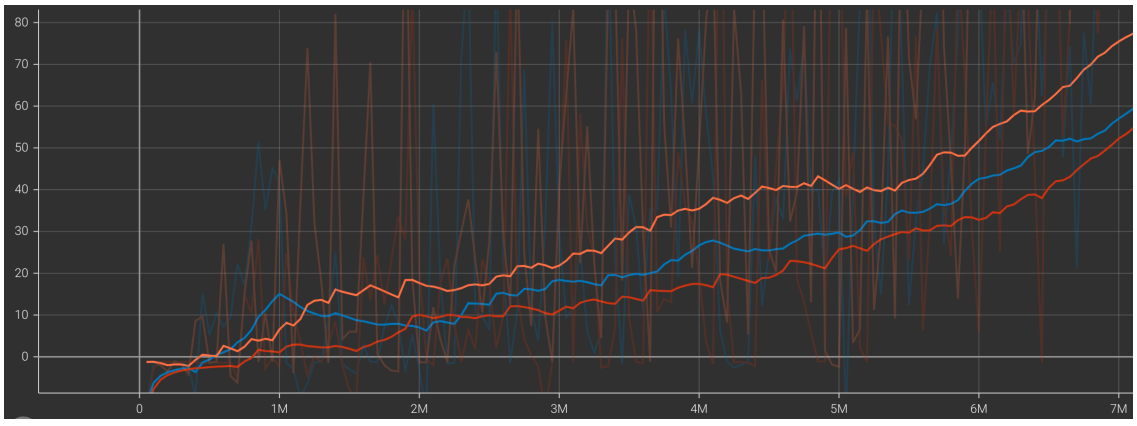

FIGURE 6.16: *Orange*: Adversarially Trained(Action Attack) Transfer Learned model, *Blue*: Transfer Learned model, *Red*: Baseline model.
X-axis: Time steps
Y-axis: Cumulative Reward

- The AT-TL model achieve a score of 69.6 and the AAT-TL model achieves a score of 75.5 at the end of 7 million steps.

- The AT-TL model gives us an improvement of 21.9% over the TL model and similarly the AAT-TL model gives us an improvement of 32.2%.
- Given the stark differences in complexity of the Two Kink Track and the AWS Track, this improvement in performance is a significant improvement over the results seen when performing vanilla transfer learning.

Table **??** shows the consolidated results obtained across Phase 2 and Phase 3 of our experimentation.

Source Track	**Target Track**	**Training Type**	**Steps** ($\times 10^6$)	**Average Reward**	**%improvement (wrt TL)**
OneKink	TwoKink	TL	7	36.3	-
		AT - TL	7	65.3	79.9
		AAT - TL	7	65.5	80.4
TwoKink	AWS	TL	7	57.1	-
		AT - TL	7	69.6	21.9
		AAT - TL	7	75.5	32.2

TABLE 6.2: Average reward of different Adversarial Attacks and Transfer Learned Models
AT - Observation Attack
AAT - Action Attack

CHAPTER 7

CONCLUSION AND FUTURE WORK

Reinforcement Learning has become a field that has found practical applications in numerous avenues and has become widely adopted to solve a wide range of industry problems. In our project, we investigated the effectiveness of adversarial training in improving the results of transfer learning in reinforcement learning, and conducted experiments for the same purpose.

We conducted our experiments on 5 different tracks, namely, Oval Track, One Kink Track, Two Kink Track, Barcelona and AWS Track. Out of these 5 tracks, we use the One Kink, Two Kink and AWS Track for transfer learning and adversarial training.

In the first phase of our project, we conducted baseline training experiments to quantify our performance when training on a single track from scratch. This experiment was performed on all 5 tracks.

In the second phase of our project, we trained models for a moderate amount of time(around 1 to 2 million time steps) on a simple track, and then transferred the trained model to a more complex track. After transferring the model, we continued training on the new track for about 8 million time steps. When compared with the baseline trained from scratch, the transfer learnt models were able to achieve 5% to 7% better performance when given the same amount of training time.

In the third phase of our project, we introduced adversarial training into our regimen in two different methods. In the first method, we performed a random

noise attack, in order to perturb the sensory inputs of the agent. In the second method, we add an action adversary to the agent which picks random actions during the training process to disturb the inputs to the agent. Both of these methods perform significantly better than vanilla transfer learning by approximately least 60%.

With these results, we have shown that adversarial training significantly improves the effectiveness of transfer learning in reinforcement learning, and helps the agents generate better representations of their states. In the future, we plan to examine the effectiveness of other adversarial attacks in the future, such as gradient-based attacks.

Appendix A

CUSTOM CHECKPOINTING SYSTEM IN UNITY