A Study of Deep Reinforcement Learning in Autonomous Racing Using DeepRacer Car

Mukesh Ghimire
University of Mississippi

A STUDY OF DEEP REINFORCEMENT LEARNING IN AUTONOMOUS RACING USING DEEPRACER CAR

by
Mukesh Ghimire

A thesis submitted to the faculty of The University of Mississippi in partial fulfillment of the requirements of the Sally McDonnell Barksdale Honors College.

Oxford
May 2021

Approved by

Advisor: Dr. Yixin Chen

Reader: Dr. Farhad Farzbod

Reader: Dr. Tejas Pandya

To my family

ACKNOWLEDGEMENTS

Firstly, I would like to thank my family for the constant love and support throughout my life. I am grateful to my brother, Nootan Ghimire, for motivating me to pursue this thesis and helping me in some of the steeper learning curves. Your lessons have been valuable for my understanding of the material.

A sincere thanks must go to my advisor, Yixin Chen. Your support and guidance throughout the process were invaluable. I am also thankful to the rest of the committee members, Farhad Farzbod and Tejas Pandya for the suggestions along the way, as well as for taking the time to be on the committee. Also, a huge thanks to the Machine Shop Supervisors, Mr. Matt Lowe and Mr. Shannon for their help in building the physical track.

I would also like to acknowledge the research fund from the Sally McDonnell Barksdale Honors College for the purchase of the DeepRacer Car, which made the works done in this thesis come to life.

ABSTRACT

MUKESH GHIMIRE: A Study of Deep Reinforcement Learning in Autonomous Racing Using DeepRacer Car (Under the direction of Dr. Yixin Chen)

Reinforcement learning is thought to be a promising branch of machine learning that has the potential to help us develop an Artificial General Intelligence (AGI) machine. Among the machine learning algorithms, primarily, supervised, semi supervised, unsupervised and reinforcement learning, reinforcement learning is different in a sense that it explores the environment without prior knowledge, and determines the optimal action. This study attempts to understand the concept behind reinforcement learning, the mathematics behind it and see it in action by deploying the trained model in Amazon's DeepRacer car. DeepRacer, a $\frac{1}{18^{\text{th}}}$ scaled autonomous car, is the agent which is trained to race autonomously on a track. Optimum race line coordinates were calculated which allowed the agent to follow the fastest possible route on a given track. The agent was then trained using proximal policy optimization (PPO). Performance metrics such as the average reward per episode and cumulative reward were examined to fine tune the model. To further understand the distribution of action spaces, log analyses tools provided by the amazon was used. Based on the log analysis data, any un-used action was removed for efficient training. The trained model was uploaded into the DeepRacer car to test it in a race track outside of simulation.

Table of Contents

List of Figures **viii**

List of Tables **ix**

List of Symbols **x**

1 Introduction **1**

1.1 Autonomous Systems and Robotics 1

1.2 Reinforcement Learning 2

1.2.1 Reinforcement Learning Framework 3

1.2.2 Reinforcement Learning Approaches 5

1.2.2.1 Value Based RL 5

1.2.2.2 Policy Based RL 6

2 DeepRacer **8**

2.1 The Vehicle 8

2.2 Environment 9

2.2.1 Service Architecture 10

2.2.2 Major Components 12

2.2.2.1 Action Space 12

2.2.2.2 Reward Function 13

2.2.3 Network Architecture 16

2.2.3.1 Hyperparameters of PPO 16

2.2.4 Local Training 18

2.2.5 Training in SageMaker Notebooks 19

3 Development **20**

3.1 Optimum Race Line 21

3.2 Universal Model 23

4 Conclusion **28**

5 Future Work **29**

References **30**

List of Figures

1 Agent-Environment Loop 3

2 RL Framework. 4

3 A game where a mouse collects cheese in a maze 4

4 Maze Example with the Value of Each State 6

5 Neural Network for training policy 7

6 DeepRacer Car. 8

7 Screenshot of the console during training. 9

8 AWS DeepRacer Architecture 11

9 AWS DeepRacer Schematics 11

10 Network Architecture 16

11 Training Metrics. 20

12 Evaluation Metrics. 21

13 Reinvent-base Race Track 22

14 Reward with K1999 Implementation. 22

15 Hyperparameters for the Universal Model. 24

16 Reward Graph for Oval Track. 25

17 Oval Track 25

18 Reward Graph for Reinvent-base Track. 26

19 Reinvent-base Track 26

20 Bowtie Track 27

21 Reward Graph for Bowtie Track. 27

List of Tables

1 AWS DeepRacer Discrete Action Space 12

2 Key-value pairs in params 14

3 Hyperparameters of PPO 18

List of Symbols

a	action
γ	discount rate
$\mathbb{E}$	Expectation
J	objective function
π^*	optimum policy
θ	parameters of the policy
π	policy
P	probability
R	reward function
r	reward
A	set of action
S	set of states
s	state
G	total reward
v	value of state

1 Introduction

1.1 Autonomous Systems and Robotics

Autonomous robots will soon be ubiquitous in the world we live in. From getting the food delivered to our footsteps to driving us to work, autonomous systems are going to be an important aspect of our daily lives. Advancement in the field of Machine Learning and Artificial Intelligence, along with the computing power have allowed the dream of self-driving cars come to life. With increase in air and land traffic, use of autonomous vehicles will lead to efficient use of the available resources.

A system can be said to be “autonomous” when all the dynamic tasks are performed by the system itself [1]. For instance, in the case of self-driving car, it should be able to perform the driving tasks, at all driving environment only using its automated system [1]. Autonomous robots are useful in the scenarios (such as in high air traffic) where human control is either infeasible or not cost-effective [2]. For instance, in the case of air traffic, imagine thousands of airplane flying in close proximity. A slight deviation of one airplane’s trajectory could affect the entire herd of airplanes. This information needs to be relayed to every other airplane that would be affected, and decision be made in split second. This task is painstakingly hard and has a high potential to go awry. However, if the airplanes have a level of autonomy and given that they would be communicating to everyone nearby, the path planning algorithm would instantly update for every airplane. Hence, avoiding the catastrophe much easily and efficiently.

1.2 Reinforcement Learning

Reinforcement learning (RL) is a machine learning framework that is useful in decision-making scenarios when an agent interacts with an environment through trial-and-error to discover the most efficient behavior [3]. It should not be confused with Genetic Algorithm, where an agent also employs the method of trial-and-error, but only the best offspring that survives proceeds to give rise to future generation. Whereas, in RL there is usually one agent that learns the optimal behavior following an iterative process.

Reinforcement Learning resembles biological systems that also learn to adapt to the environment through trial-and-error. The most common example would be training a dog. It is often seen that in order to train a dog to follow certain commands, the trainer uses “treats” to reinforce the correct behavior. In psychology, encouraging actions by the help of reward is referred to as Positive Reinforcement, whereas discouraging a behaviour through the means of punishment is referred to as Negative Reinforcement.

The central goal of reinforcement learning is maximizing the overall reward. Each action of an agent corresponds to a reward, and hence the agent tries to accumulate as much reward as possible in an episode and further tries to repeat the higher-reward yielding actions. In order to assess the nature of the reward, one could say that the agent needs to try every possible action. We can easily see how this could be an issue, and indeed this is one of the challenges in reinforcement learning. The problem, often referred to as the exploration/exploitation trade-off.

As explained in the Reinforcement Learning book [4], for an agent to maximize its rewards, it has to select actions that was found effective in the past, i.e. *exploit* the existing actions and at the same time identify such possible actions. However, to identify such actions, it needs to try newer actions, i.e. it has to *explore*. An agent cannot both *exploit* and *explore* at the same time, and too much of one can prevent us from reaching our goal. There has to be a good balance and a good approximation can be made based on trial-and-error.

1.2.1 Reinforcement Learning Framework

As explained earlier, the reinforcement learning framework consists of an agent, environment, states, actions, and rewards. Figure 1 shows the simplest representation of the RL framework.

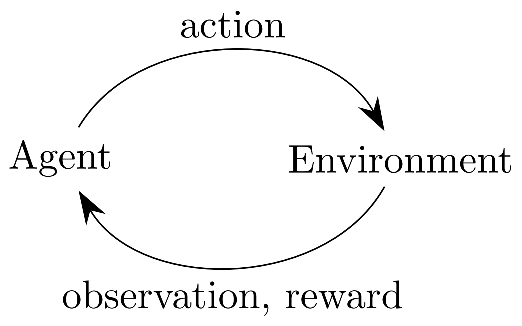


Figure 1: Agent-Environment Loop. [5]

At first, the agent receives the state S_0 from the environment which may contain information such as the image captured by the agent, its speed, and other sensory data. The agent then acts on the provided state S_0 with an action A_0. As a result of the action, the environment switches to a new state S_1 and the environment provides a reward R_1 to the agent. This loop continues until the episode is over and each loop returns current state (S_0) and action (A_0), and future state (S_1) and reward (R_1). A comprehensive representation of the process is shown in figure 2.

The agent's goal is to always maximize the expected reward. This concept is known as the Reward Hypothesis which, as explained in [6], in the most naive approach, can be expressed as the following:

$$G_t = R_{t+1} + R_{t+2} + ... \tag{1}$$

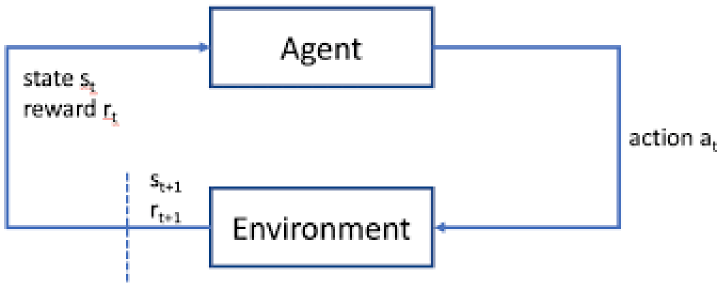


Figure 2: RL Framework.

$$G_t = \sum_{k=0}^{\infty} R_{t+k+1} \tag{2}$$

The equation 2 assumes that the rewards at all instance of time have same weight. However, this cannot be true, for instance, rewards that come sooner are more likely to happen than the rewards that occur much later [6]. Hence, the idea of a discount factor (γ) is essential. Adding γ to equation 2, we get

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \text{ where } \gamma \in [0,1) \tag{3}$$

A classic example of mouse in a maze can be used to visualize this. In figure 3, the goal of the mouse is to collect as much cheese as possible, without getting eaten by the cat. It is apparent that collecting the cheese nearby the mouse is more rewarding than attempting to go for the cheese near the cat.

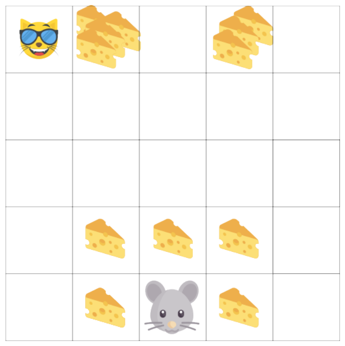


Figure 3: A game where a mouse collects cheese in a maze. [6]

1.2.2 Reinforcement Learning Approaches

1.2.2.1 Value Based RL

Value based RL is used in Q-learning algorithm where the objective is to maximize the total expected discounted reward (or minimize the total discounted expected cost) [7]. The total expected discounted reward, here known as the "value function", is the maximum expected future reward that the agent receives at each state.

An agent following the Q-learning algorithm in an environment is expressed as a finite Markov Decision Process (MDP) [7]. MDPs, in the simplest term, refer to the RL framework mentioned in the section 1.2.1. It is a set of tuple (S, A, P, R, γ), where S is the state, A is the action, P is the state transition probability function, R is the reward function, and γ is the discount factor. At each time step, the agent in some state $s_t \in S$ chooses an action $a_t \in A$, assigns it to the environment which results in a state transition to $s_{t+1} \in S$ with the probability $P(s_{t+1}|s_t, a_t)$. It is then provided with the expected reward of $R(s_t, a_t, s_{t+1})$. Finally, the value of each state is the total expected reward that the agent can accumulate over the future beginning in that state.

$$v_\pi(s_t, a_t) = \mathbb{E}_\pi \left[\, R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \mid s_t, a_t \, \right] \tag{4}$$

Consider a maze in figure 4, where an agent starting in the left has to reach the goal on the right. The numbers in the figure are the results of the value function, i.e. each cell correspond to the maximum expected future reward for that given state. The agent is able to complete the maze following the optimization goal of maximizing the future reward.

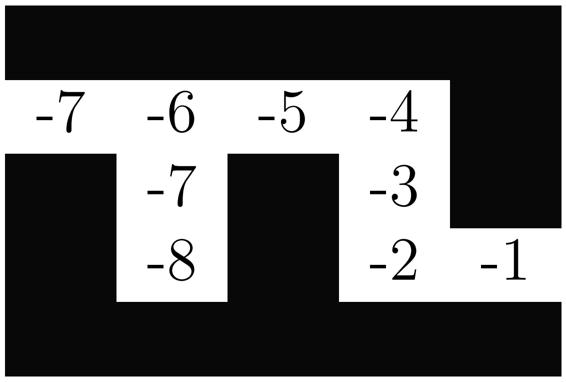


Figure 4: Maze Example with the Value of Each State. [6]

Algorithm 1: Q-Learning Algorithm.

```
Result: Optimum Q Table
Initialize Q Table;
while not end of training do
    choose action, a;
    perform action;
    measure reward;
    update Q;
end
```

1.2.2.2 Policy Based RL

Policy (π) refers to a function that maps states to action i.e. it takes in the current state as input and decides what action to take next. In policy-based learning, the goal of the agent is to learn the optimum policy (π^*) that maximizes the expected reward. Policy could be either deterministic or stochastic. A deterministic policy always returns one action for the given state, whereas a stochastic policy returns the probability distribution of the action for the given state. Equation 5 and 6 represent how the actions are selected in deterministic and stochastic policies respectively.

$$a = \pi(s) \tag{5}$$

$$\pi(a|s) = P(A \mid s) \tag{6}$$

To solve an RL problem, the goal is to find an optimum policy, which is able to "recommend" actions that produce maximum expected reward. The optimum policy can be found by directly training the policy, which can be done using deep neural networks. In figure 5, the policy network takes in a frame from the game and produces a probability distribution over the action space (L or R) [8]. In order to train the policy, a function is required which can be used to quantify the quality of the policy. As our goal is to maximize the expected reward, the expected rewards can be used as the objective function.

$$J(\theta) = \mathbb{E}_{\pi\theta}\left[\sum_{t=0}^{T} \gamma r(s_t, a_t)\right] \tag{7}$$

where θ represents parameters of the policy.

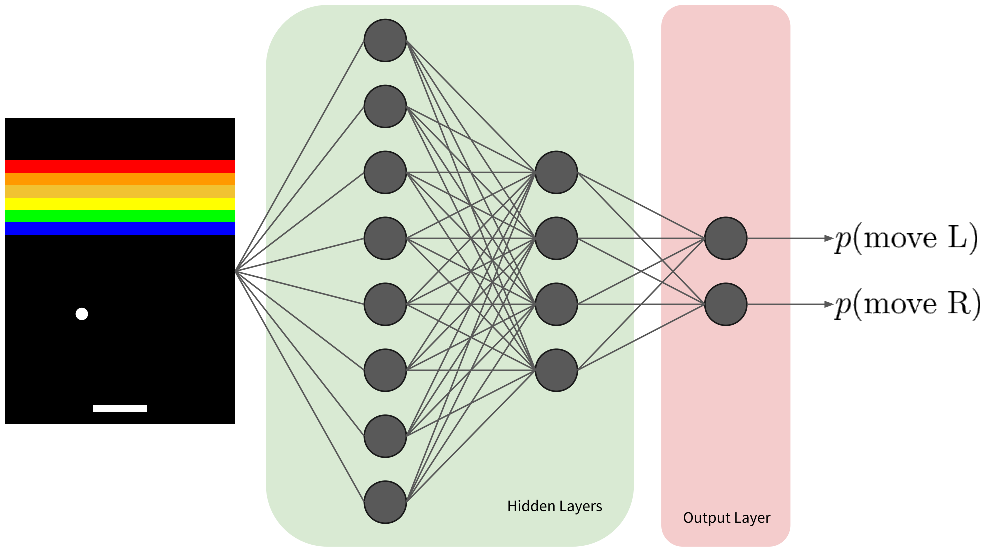


Figure 5: Neural Network for training policy. [8]

2 DeepRacer

2.1 The Vehicle

DeepRacer is a fully autonomous 1/18th scale model of a racer car driven using reinforcement learning. It was developed by Amazon for users of all level to learn about reinforcement learning [9]. It allows users to learn reinforcement learning by doing, primarily through their DeepRacer Console. Users have the option to deploy their model to the vehicle after they train the model. Figure 6 shows the front and side view of the physical car that was used to test the model in real life scenario.

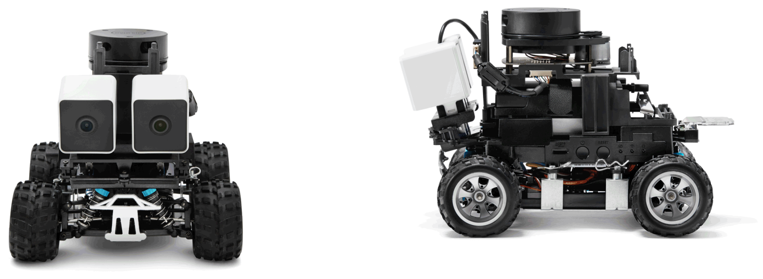
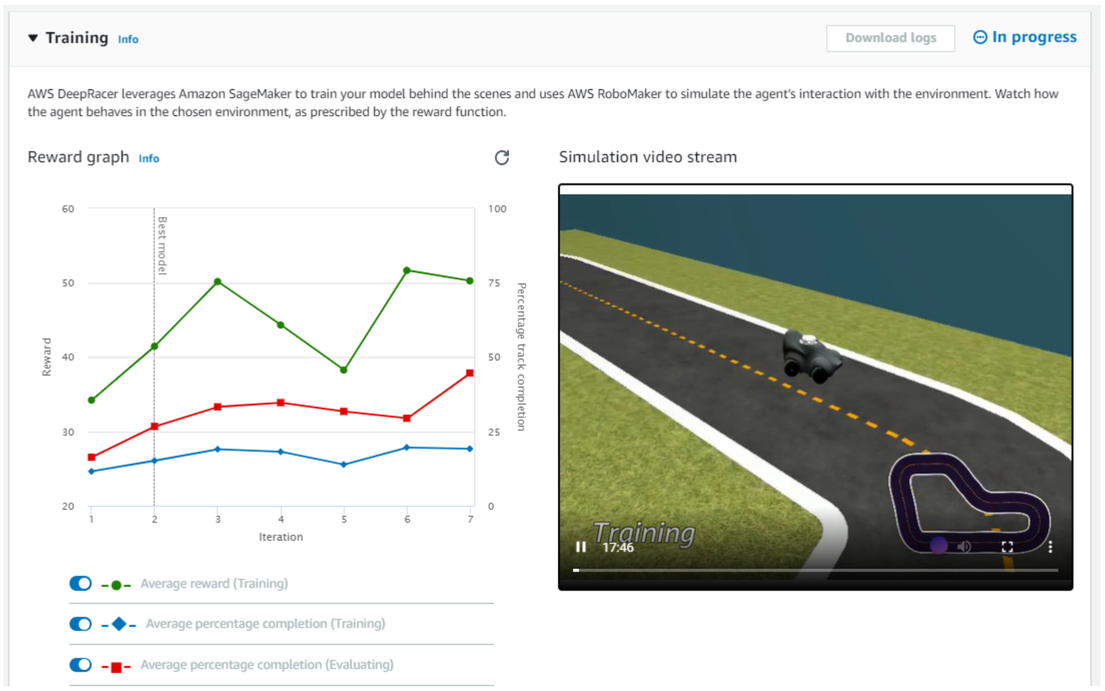

(a) Front View of the car (b) Side view of the car

Figure 6: DeepRacer Car.

The vehicle is able to run autonomously by running the inference based on the reinforcement learning model that is uploaded by the user [9]. It can also be operated manually using its internal console. The vehicle is powered by brushed motor and its speed is controlled using a voltage regulator that controls the motor. The steering is controlled by the servomechanism [9]. As shown in figure 6, the vehicle used in this project was equipped with steoreo cameras and a LiDAR sensor to enable object avoidance and head-to-head racing. For the purpose of this thesis, only time-trial race setting was used. However, LiDAR was still used in order to provide robustness to the model as building a good DIY physical track on which the vehicle is supposed to race is difficult. The forward facing stereo cameras help the car learn the depth information from the images.

2.2 Environment

The primary method of training a model to run the deepracer car autonomously is via the AWS DeepRacer console. The console is an interactive platform that allows users to monitor training and evaluation of the model while displaying the primary log metrics (reward, percent completion during training and evaluation). The console view during training is shown in figure 7.



▼ Training Info
Download logs
In progress
Reward graph Info
Simulation video stream
Best model
Reward
Percentage track completion
Iteration
Average reward (Training)
Average percentage completion (Training)
Average percentage completion (Evaluating)
Training
17:46

AWS DeepRacer leverages Amazon SageMaker to train your model behind the scenes and uses AWS RoboMaker to simulate the agent's interaction with the environment. Watch how the agent behaves in the chosen environment, as prescribed by the reward function.

Figure 7: Screenshot of the console during training.

2.2.1 Service Architecture

The AWS Deepracer environment is comprised of SageMaker, RoboMaker along with the other AWS cloud services [9]. SageMaker is an AWS machine learning platform that allows to train ML models and RoboMaker is a cloud service for developing, testing and deploying robotic solutions. DeepRacer encorporates these platforms to train reinforcement learning models and to create virtual agent and environment using SageMaker and RoboMaker respectively. It uses cloud storage platform S3 to store trained models along with the training logs and other related artifacts [9].

AWS RoboMaker generates a virtual environment for the agent to drive along a defined track within the AWS DeepRacer architecture. The agent operates in accordance with the policy network model that has been trained in SageMaker up to a certain point. An episode is described as a run that begins at the starting line and ends at the finishing line or off the track [9]. The course is divided into segments of a fixed number of steps for each episode. "Experiences" are cached in *Redis* as an experience buffer in each segment. Experience buffer is defined as an ordered list of the tuples of (state, action, reward, new state) for each steps [9]. *Redis* is an in-memory database that is used by AWS DeepRacer as an experience buffer to select training data to train the policy neural network. SageMaker randomly pulls training data from the experience buffer in batches and feeds it to the neural network to update the weights. The revised model is then stored in S3 for SageMaker to use in order to generate more experiences. This loop runs until the training is completed. In the very first episode of the training, the experienced buffer is initialized with random actions. Figures 8 and 9 illustrate this architecture. Using this setup is beneficial because it allows running multiple simulations to train a model on several different segments of a track simultaneously or to train the model in multiple tracks simultaneously [9].

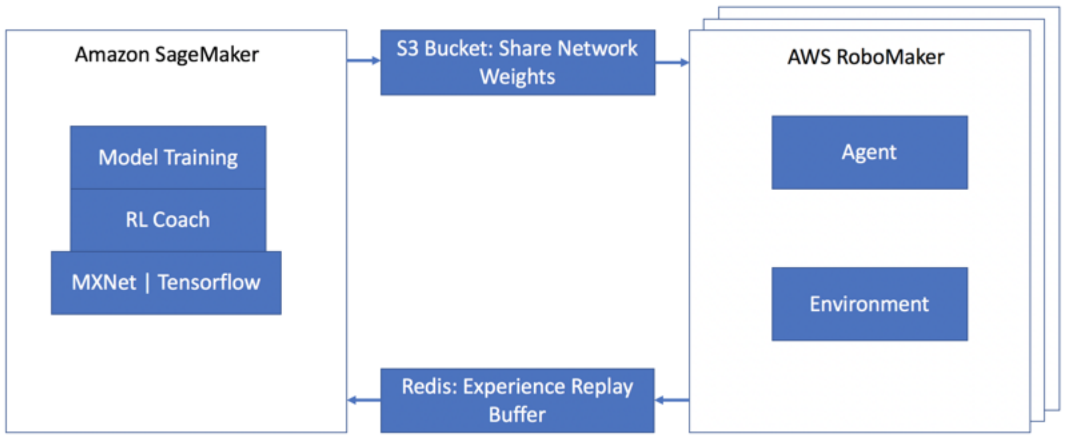


Figure 8: AWS DeepRacer Architecture. [9]

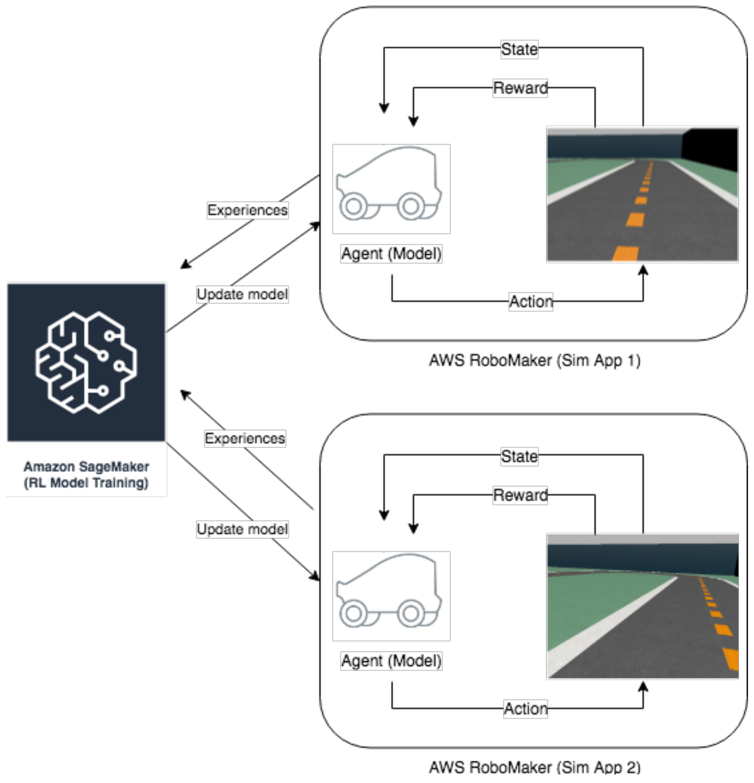


Figure 9: AWS DeepRacer Schematics. [9]

2.2.2 Major Components

The primary components that the users have the freedom to explore are the Action Space and the Reward Function.

2.2.2.1 Action Space

Action space refers to the set of all valid actions or choices available to an agent while it interacts in the environment. Action space can be further classified into discrete or continuous action space. A discrete action space, by definition, is just a set of all possible action spaces for each state. In terms of the reinforcement learning problem, the underlying algorithm selects one action from the set of actions based on given state. In the case of the DeepRacer car, there are two actions – steering (direction) and speed. Based on the input to the camera and LiDAR sensor, the neural network selects a speed and direction for the car. The default discrete action space is shown in table 1.

Table 1: AWS DeepRacer Discrete Action Space. [9]

Action number	Steering	Speed
0	-30 degrees	0.4 m/s
1	-30 degrees	0.8 m/s
2	-15 degrees	0.4 m/s
3	-15 degrees	0.8 m/s
4	0 degrees	0.4 m/s
5	0 degrees	0.8 m/s
6	15 degrees	0.4 m/s
7	15 degrees	0.8 m/s
8	30 degrees	0.4 m/s
9	30 degrees	0.8 m/s

Unlike discrete action space, a continuous action space allows the agent to select an action from a range of values for each state [9]. Similar to the discrete case, the agent selects direction-speed pair based on the environmental situation that is received from the camera and LiDAR inputs. However, in the continuous action space, the agent has a range of options to pick from. There is a trade off between performance and training time for these two action spaces. While continuous action space provides a better optimization as there are range of values to pick from and the agent could select the optimum value pairs to improve performance than compared to a discrete action space where the agent is forced to pick from a set of allowable actions. However, the fact that there are range of values in a continuous action space means that the agent has to train longer, which increases resource utilization.

2.2.2.2 Reward Function

Reward is the central idea in reinforcement learning. RL is guided by reward hypothesis, which basically states that the goal of the agent is to maximize the expected reward. But, how do we reward the agent while it is training? The answer is with the help of a “Reward Function”. Reward function is one of the most important parts of the AWS DeepRacer platform, which essentially provides the motivation to the agent to take actions. In the case of the DeepRacer, the reward function is a python function that takes in a dictionary object of parameters containing present state information and returns a numerical estimation of reward. Inside the function, the user can “reward” a certain action or “penilize” it. User has the choice to provide a fixed reward or a reward that is a function of the parameters. The general outline of a reward function is shown below.

```python
def reward_function(params) :
    reward = ...
    return float(reward)
```

The `params` dictionary contains key-value pairs of the state measurements shown in table 2. It is not necessary to use all of these parameters in designing the reward function. A simple reward function might also lead to a satisfactory performance. However, some or all of the parameters might come in handy when writing reward functions for complicated tracks that may require the agent to focus on several different parameters simultaneously to complete the race.

Table 2: Key-value pairs in `params`. [9]

key	value	detail
all_wheels_on_track	Boolean	flag to indicate if the agent is on the track
x	float	agent's x-coordinate in meters
y	float	agent's y-coordinate in meters
closest_objects	[int, int]	zero-based indices of the two closest objects to the agent's current position of (x, y)
closest_waypoints	[int, int]	indices of the two nearest waypoints
distance_from_center	float	distance in meters from the track center
is_crashed	Boolean	Boolean flag to indicate whether the agent has crashed
is_left_of_center	Boolean	Flag to indicate if the agent is on the left side to the track center or not
is_offtrack	Boolean	Boolean flag to indicate whether the agent has gone off track
is_reversed	Boolean	flag to indicate if the agent is driving clockwise (True) or counter clockwise (False)
heading	float	agent's yaw in degrees
objects_distance	[float,]	list of the objects' distances in meters between 0 and track_length in relation to the starting line
objects_heading	[float,]	list of the objects' headings in degrees between -180 and 180
objects_left_of_center	[Boolean,]	list of Boolean flags indicating whether elements' objects are left of the center (True) or not (False)
objects_location	[(float, float),]	list of object locations [(x,y), ...]
objects_speed	[float,]	list of the objects' speeds in meters per second
progress	float	percentage of track completed
speed	float	agent's speed in meters per second (m/s)
steering_angle	float	agent's steering angle in degrees
steps	int	number steps completed
track_length	float	track length in meters
track_width	float	width of the track
waypoints	[(float, float),]	list of (x,y) as milestones along the track center

A simple example of reward function that encourages the agent to closely follow the center line is shown below [9]. This reward function determines how far away the car is from the center line of the track and assigns higher reward if it is closer to the center line. This reward function leverages only two parameters – ‘track_width’ and ‘distance_from_center’.

```python
def reward_function(params):
    '''
    Example of rewarding the agent to follow center line
    '''

    # Read input parameters
    track_width = params['track_width']
    distance_from_center = params['distance_from_center']

    # Calculate 3 markers that are increasingly further away from
    the center line
    marker_1 = 0.1 * track_width
    marker_2 = 0.25 * track_width
    marker_3 = 0.5 * track_width

    # Give higher reward if the car is closer to center line and
    vice versa
    if distance_from_center <= marker_1:
        reward = 1
    elif distance_from_center <= marker_2:
        reward = 0.5
    elif distance_from_center <= marker_3:
        reward = 0.1
    else:
        reward = 1e-3  # likely crashed/ close to off track

    return reward
```

2.2.3 Network Architecture

The default algorithm in DeepRacer is Proximal Policy Optimization (PPO) algorithm. Recently, aws has also added another algorithm called Soft Actor Critic (SAC) algorithm. PPO and SAC are similar in the sense that they both learn a policy and value function at the same time [9]. However, their strategies vary. For this project, only PPO was used as SAC is a very recent addition to the platform.

PPO uses two neural networks namely a policy network and a value network. The policy network, also known as the actor network decides which action to take based on the input (camera image and LiDAR input) [10]. The value network, also known as the critic network estimates the cumulative reward based on the inputs [10]. Out of the two networks, the policy network interacts with the simulator and gets deployed to the car. Figure 10 shows the architecture of the network.

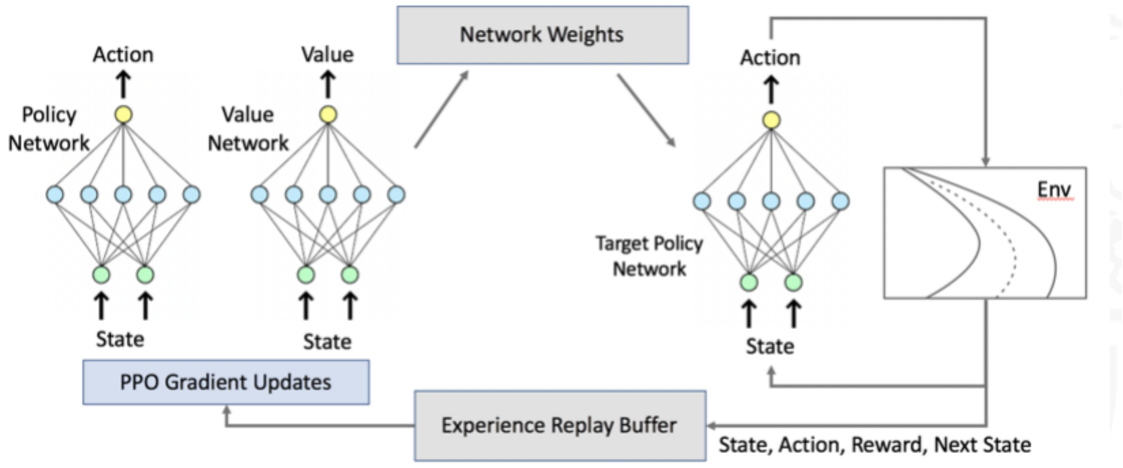


Figure 10: Network Architecture. [10]

2.2.3.1 Hyperparameters of PPO

Hyperparameters are the variables that affect the training process and are not an intrinsic property of the model. Choosing optimal hyperparameters, unfortunately, is an empirical process. There are no formulation to find the best set of hyperparameters and requires systematic experimentation to derive. Before discussing about the hyperparameters of the DeepRacer model, it is necessary to understand a few terminologies. We have discussed *experience, experience buffer* and *episode* before. The other two terms are *batch* and *training data.* A batch is an ordered list of experiences, representing a portion of simulation over a period of time, and is used to update the policy network weights [9]. A training data is a set of random batches from an experience buffer which is also used for training the policy network weights [9].

The hyperparameters of the PPO, along with their descriptions are listed in table 3.

Table 3: Hyperparameters of PPO. [9]

Hyperparameters	Description
Gradient descent batch size	The number recent vehicle experiences sampled at random from an experience buffer and used for updating the underlying deep-learning neural network weights
Number of epochs	The number of passes through the training data to update the neural network weights during gradient descent
Learning rate	The learning rate controls how much a gradient-descent (or ascent) update contributes to the network weights
Entropy	A degree of uncertainty used to determine when to add randomness to the policy distribution. The added uncertainty helps the AWS DeepRacer vehicle explore the action space more broadly
Discount factor	A factor specifies how much of the future rewards contribute to the expected reward. The larger the Discount factor value is, the farther out contributions the vehicle considers to make a move and the slower the training
Loss type	Type of the objective function used to update the network weights Valid values: **Huber loss, Mean Squared error loss**
Number of experience episodes between each policy-updating iteration	The size of the experience buffer used to draw training data from for learning policy network weights. Valid values: Integer between 5 and 100.

2.2.4 Local Training

Using AWS services online can be a costly option as training reinforcement learning models is a computationally expensive process requiring GPUs. Fortunately, due to huge community involvement and collaboration between the developers, several local training environments are available. However, due to compatibility issues and lack of powerful GPUs, local training was not feasible for this project. The setup that was intended to use allowed user to train DeepRacer without the use of the DeepRacer console, SageMaker, or RoboMaker services [11]. All of the services could be installed locally in the system and the model could be trained completely offline. A good future project would be to utilize these resources to setup local training as it provides significantly higher freedom and is cheaper.

2.2.5 Training in SageMaker Notebooks

The training method discussed earlier included the DeepRacer console, which provides an integrated experience to train and evaluate DeepRacer models. The console uses SageMaker and RoboMaker behind the scenes to allow the user to train and evaluate the models seamlessly [9]. The SageMaker notebook provides a “jailbreak” experience of AWS DeepRacer by giving us more control over the training/simulation process and RL algorithm tuning. Figure 9 in an earlier section shows an example of distributed RL training across SageMaker and two RoboMaker simulation environments that perform *rollouts* – execute a fixed number of episodes using the current model or policy. The rollouts collect agent experiences (state-transition tuples) and share this data with SageMaker for training. SageMaker updates the model policy which is then used to execute the next sequence of rollouts. This training loop continues until the model converges [9].

3 Development

The initial step in this project was to come up with an effective reward function. At first, the default reward function was used to train the network. The hyperparameters were also set to default values. The training time was set to one hour. After training, the model was also evaluated in the same track for three trials. The total reward per each episode and percent completion is shown in figure 11.

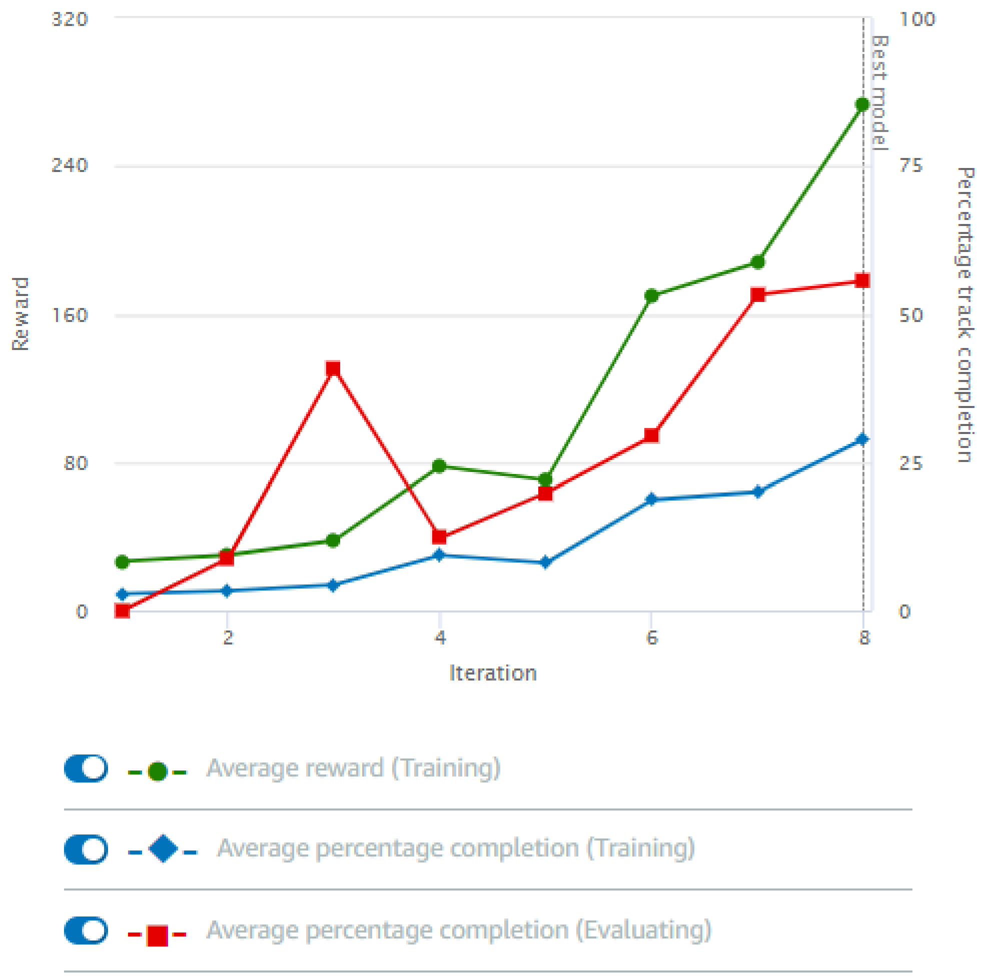
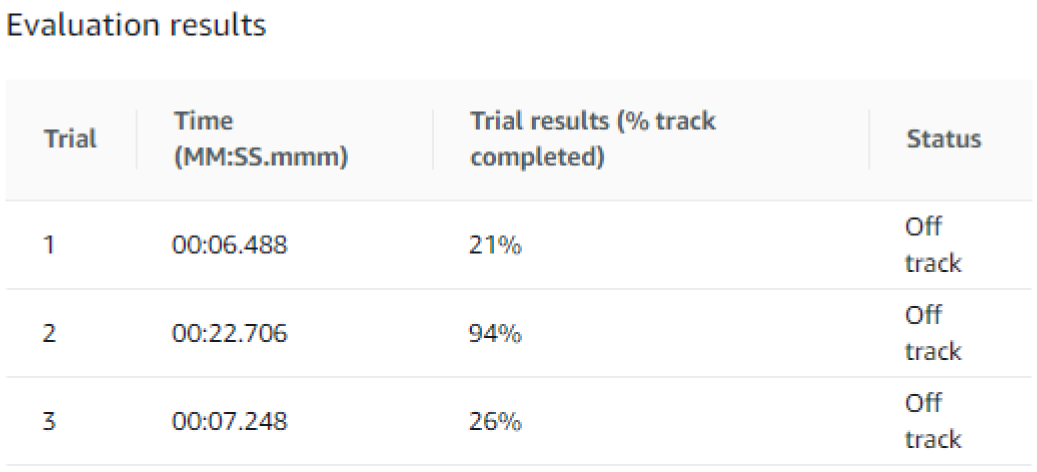


Figure 11: Training Metrics.

Evaluation metrics are shown in figure 12. The evaluation trials were not able to fully complete the track. The second trial was able to complete 94% of the track before getting off track. The probable reason behind the failure was less training time.


Evaluation results

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:06.488	21%	Off track
2	00:22.706	94%	Off track
3	00:07.248	26%	Off track

Figure 12: Evaluation Metrics.

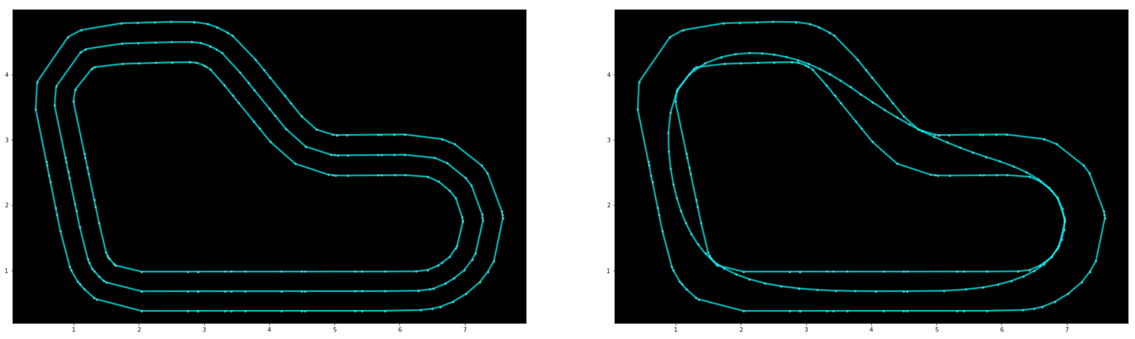
3.1 Optimum Race Line

In order to optimize the reward function so as to enable the agent to finish the race faster. The algorithm was introduced in the PhD thesis of Remi Coulom [12]. The idea is to calculate the optimum race line path for a given track and encourage the agent to follow that path. A race track is defined using three sets of co-ordinates – outer and inner boundaries along with the mid-point. Using the algorithm mentioned in [12], optimum race line for any track represented using the three coordinate sets, [13] developed a python function to generate the race track with the “optimum” path. An example of this method in use is shown in figure 13.

In the K1999 algorithm, c_i of the track is the curvature at each point $\vec{x}_i$ and is computed as the inverse of the circumscribed circle for points $\vec{x}_{i-1}$, $\vec{x}_i$ and $\vec{x}_{i+1}$ [12]. The curvature is positive for curves to the left and negative for curves to the right. The points are initially set at the center of the track.

Algorithm 2: K1999 PATH OPTIMIZATION ALGORITHM. [12]

```
for $i$ = 1 to $n$ do
    $c_1 \leftarrow c_{i-1}$
    $c_2 \leftarrow c_{i+1}$
    set $\vec{x}_i$ at equal distance to $\vec{x}_{i-1}$ so that $c_i = \frac{1}{2}(c_1 + c_2)$
    if $\vec{x}_i$ is out of the track then
        Move $\vec{x}_i$ back onto the track
```



(a) Original Race Track (b) Calculated Race Line

Figure 13: Reinvent-base Race Track. [13]

The model was trained using the reward function provided in [13] that implemented the K1999 Race-Line Optimization algorithm. The reward accumulated during training is shown in figure 14.

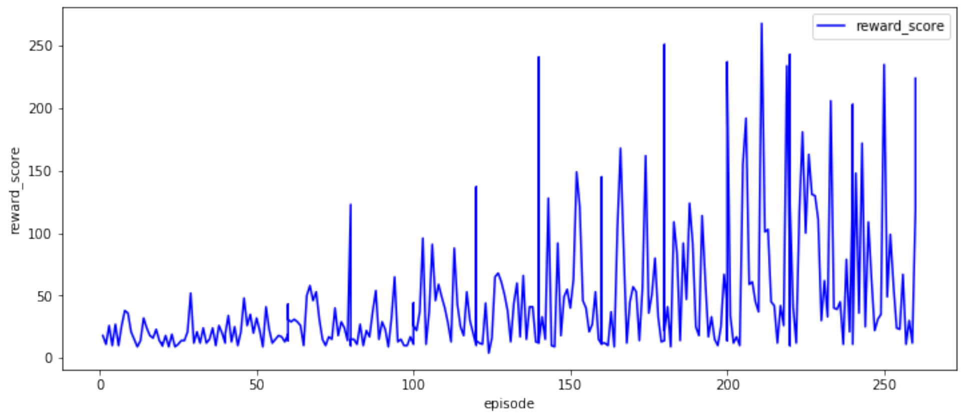


Figure 14: Reward with K1999 Implementation.

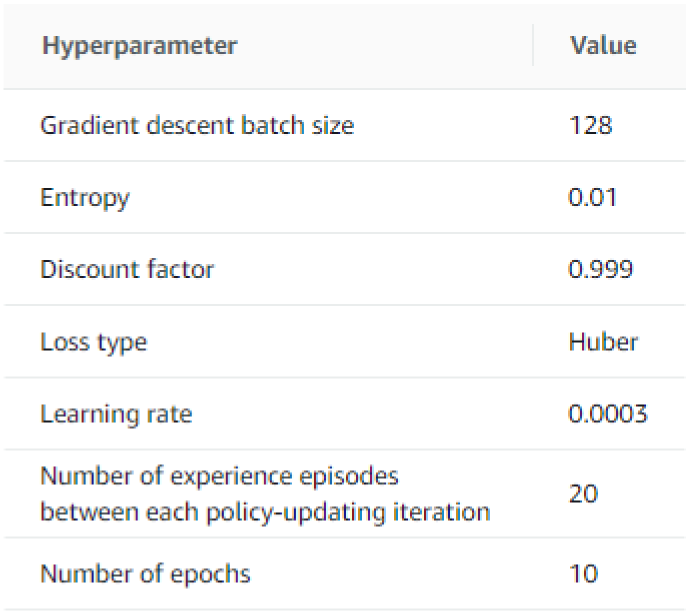
Though the rewards in figure 14 seem good, it is not in the best interest for the purpose of this work. Training with this reward function has a high chance of over-fitting to a track. Implementing the optimum race-line track allows the model to perform best only on the track it was trained on. Since the end goal of this thesis is to run the DeepRacer car on a physical track, it is necessary that the model is able to tackle unknown track. The track designed for testing the car might not resemble the tracks that the model is trained on. Hence, it is necessary that we create a “universal” model.

3.2 Universal Model

Training DeepRacer models for one track provides a satisfactory result for that particular track and is not really a robust model that can be used across all the available tracks in the DeepRacer. Inspired from [14], a more generalized reward function was used to train a model on multiple tracks by cloning a model after each training session. The reward function that was used for this purpose adapted from [14] is shown below. The hyperparameters for the universal model training is shown in figure 15.

```python
def reward_function(params):
    reward = 0.001
    if params["all_wheels_on_track"]:
        reward += 1
    if abs(params["steering_angle"]) < 5:
        reward += 1
    reward += ( params["speed"] / 8 )

    return float(reward)
```



Hyperparameter	Value
Gradient descent batch size	128
Entropy	0.01
Discount factor	0.999
Loss type	Huber
Learning rate	0.0003
Number of experience episodes between each policy-updating iteration	20
Number of epochs	10

Figure 15: Hyperparameters for the Universal Model.

At first the model was trained on the oval track for an hour. The reward graph is shown in figure 16 and the Oval track is shown in figure 17. The training time was set to one hour because the model would be cloned to train it on another track. By cloning, the current network weights would stay the same so that the "knowledge" from training in this track would carry over to the next. It can also be seen from the reward graph that the model was not able to complete the track during training. The best model is also shown in the graph based on the completion of the track.

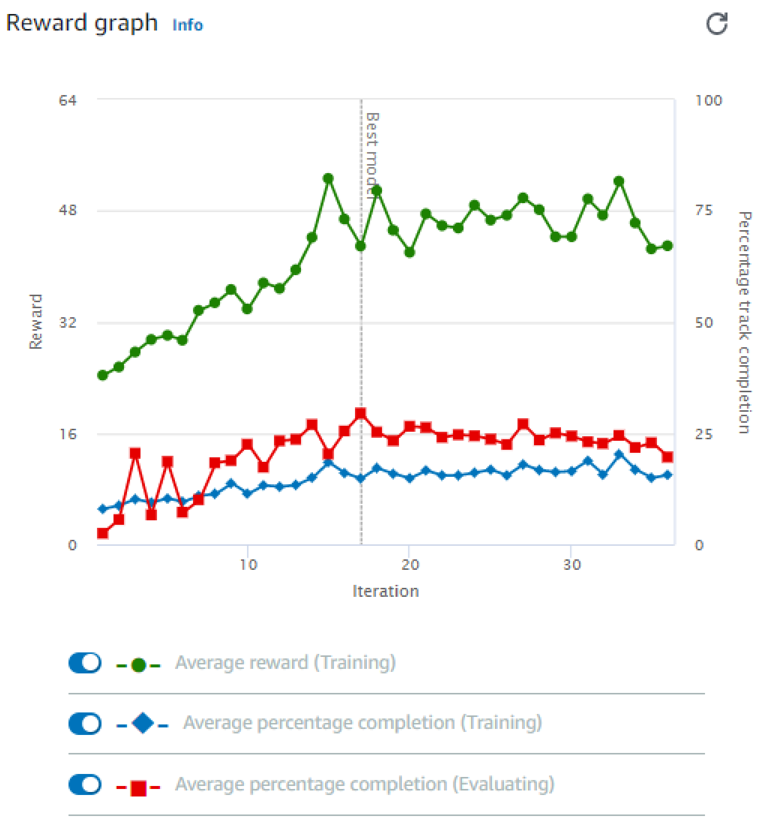


Figure 16: Reward Graph for Oval Track.

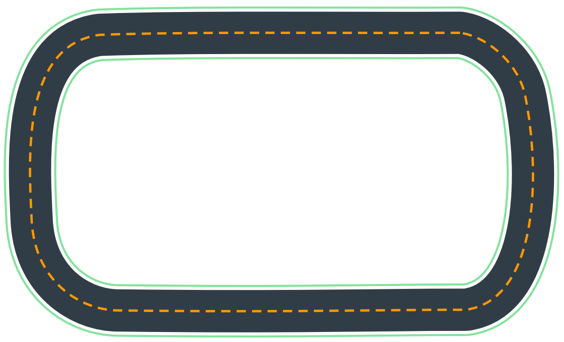


Figure 17: Oval Track.

The model was cloned to train on the Reinvent-base track. Since it is a cloned model, the reward function and the hyperparameters were the same. Figures 18 and 19 are the reward graph and the track layout respectively. Both the reward and track completion showed significant improvement in this training session.

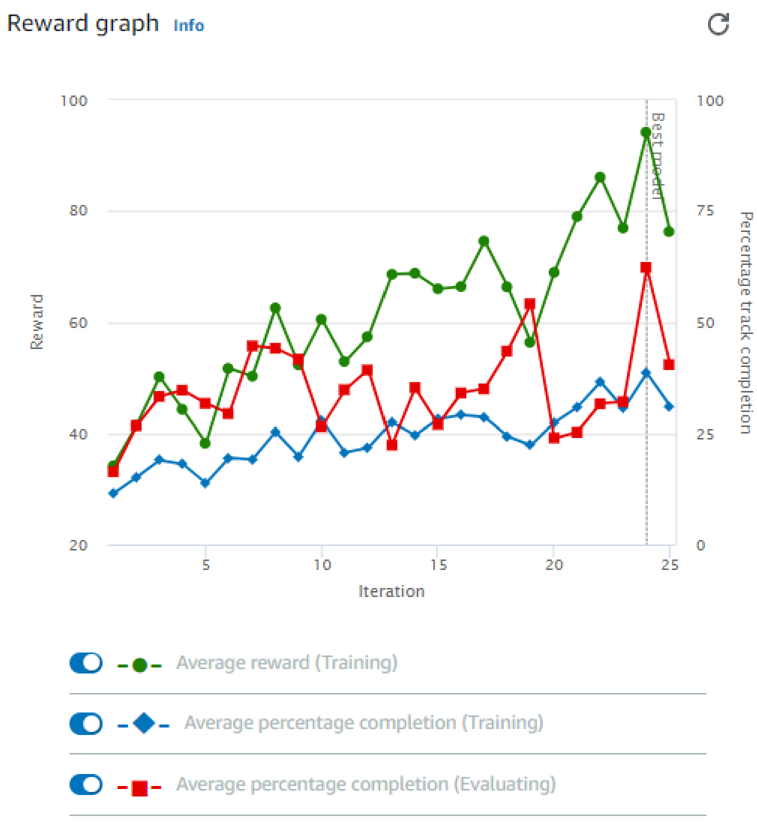


Figure 18: Reward Graph for Reinvent-base Track.

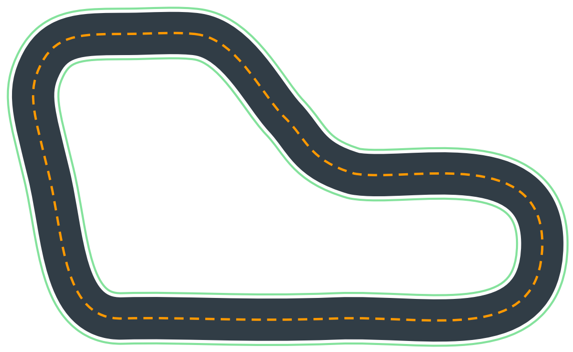


Figure 19: Reinvent-base Track.

This model was cloned to further train on a rather difficult track – Bowtie track. The track is shown in figure 20. This track turned out to be a challenging one due to the bow shape. The agent was not able to comprehend the curve in the middle as it could "see" the track on the other side of the curve and wanted to take a "shortcut" every time it encountered those curves. As a result the completion rate fell down in this training session, which can be seen in the reward graph in figure 21. The reward was also impacted possibly due to the same reason. Since the model performance was good on the reinvent track, it was further trained to get even better results.

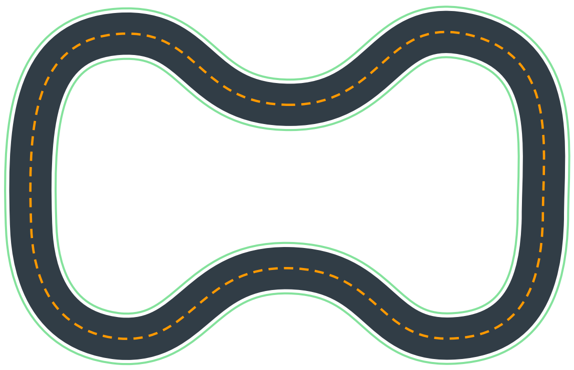


Figure 20: Bowtie Track.

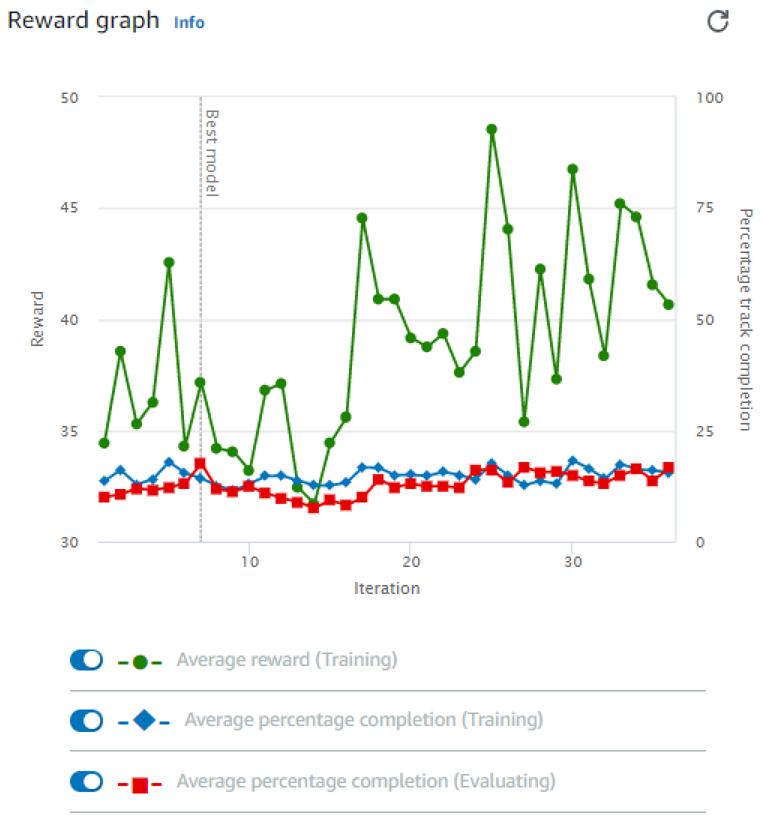


Figure 21: Reward Graph for Bowtie Track.

4 Conclusion

AWS DeepRacer is a good platform to understand reinforcement learning and see it in action. It provides easy-to-understand instructions for beginners to get into RL as well as allows an "advanced" user to access the backdoor. Different reward functions along with the state of the art optimization algorithms were used to train the models with the aim to run the physical car autonomously. However, building the track required high precision and meticulousness as the training method employed could produce satisfactory result only when the track resembled the training track. AWS has provided a step-by-step guide to building tracks with the purchase of their track bundle. Using the model trained in this project, the car was able to navigate around an empty room with race track like boundaries. With appropriate track building tools, the car would be able to successfully navigate the race-track it was trained on just like during the simulated evaluation.

5 Future Work

This project was rather comprehensive which meant dealing with every variable in the DeepRacer model training. A good approach for future work would be to keep everything constant except for one variable, such as the effect of varying reward scaling, effect of different hyperparameters. Topics like these provide a deeper understanding of the material and the results would be highly interpretable.

Another aspect that hampered the flow of this project was lack of resources for training. The AWS DeepRacer service, though convenient, is a costly option to train and evaluate the model. Reinforcement learning method is a training intensive process, that is, the more it gets to interact with the environment the more it learns. Despite several attempts to setup local training, due to lack of GPU and CPU resources and other compatibility issues, local training could not be set up successfully. For future purposes it would be wise to setup the training platform on a powerful Ubuntu machine so as to eliminate any compatibility issues. Developers around the world have gotten together to create a local environment that seamlessly connects the AWS servers to allow model submission for both verification and competition. The models developed during this project helped to understand the concept of reinforcement learning. With appropriate resources and adequate hours of training, models could be submitted in the future to participate in both virtual and physical races.

References

[1] A. Faisal, T. Yigitcanlar, M. Kamruzzaman, and G. Currie, "Understanding autonomous vehicles: A systematic literature review on capability, impact, planning and policy," *Journal of Transport and Land Use*, vol. 12, Jan. 2019. DOI: 10.5198/jtlu.2019.1405. [Online]. Available: https://www.jtlu.org/index.php/jtlu/article/view/1405.

[2] S. Bensalem, M. Gallien, F. Ingrand, I. Kahloul, and T.-H. Nguyen, "Designing autonomous robots," *IEEE Robotics Autom. Mag.*, vol. 16, pp. 67–77, 2009. DOI: 10.1109/MRA.2008.931631. [Online]. Available: https://doi.org/10.1109/MRA.2008.931631.

[3] N. Akalin and A. Loutfi, *Reinforcement learning approaches in social robotics*, 2020. arXiv: 2009.09689 [cs.RO].

[4] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[5] "Openai documentation: Getting started with gym," Accessed: February 21, 2021. [Online]. Available: https://gym.openai.com/docs.

[6] T. Simonini, "Deep reinforcement learning course v2.0," Accessed: February 21, 2021. [Online]. Available: https://www.simoninithomas.com/deep-rl-course.

[7] C. Szepesvári and M. L. Littman, "A unified analysis of value-function-based reinforcement-learning algorithms," *Neural Computation*, vol. 11, no. 8, pp. 2017–2060, 1999. DOI: 10.1162/089976699300016070. eprint: https://doi.org/10.1162/089976699300016070. [Online]. Available: https://doi.org/10.1162/089976699300016070.

[8] M. Deshpande, *Deep reinforcement learning: Policy-based methods*, Accessed: March 16, 2021. [Online]. Available: https://mohitd.github.io/2019/01/20/deep-rl-policy-methods.html.

[9] *Deepracer documentation*, Accessed: February 10, 2021. [Online]. Available: https://docs.aws.amazon.com/deepracer/latest/developerguide/.

[10] *Deepracer car*, Accessed: February 10, 2021. [Online]. Available: https://cse.buffalo.edu/~avereshc/rl_spring20/Xingtong_Li.pdf.

[11] *Aws deepracer community wiki*, Accessed: February 21, 2021. [Online]. Available: https://wiki.deepracing.io.

[12] R. Coulom, "Reinforcement learning using neural networks, with applications to motor control," Ph.D. dissertation, Institut National Polytechnique de Grenoble-INPG, 2002.

[13] C. D. Thompson, *Discovering race lines in deepracer track geometries.* [Online]. Available: https://github.com/cdthompson/deepracer-k1999-race-lines.

[14] S. Pletcher, *Aws deepracer experimentation*, Accessed: February 27, 2021. [Online]. Available: `https://github.com/scottpletcher/deepracer`.