Nhan Mai

Apply reinforcement learning in AWS DeepRacer

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

1 April 2022

Abstract

Author:	Nhan Mai
Title:	Apply Reinforcement Learning in AWS DeepRacer
Number of Pages:	46 pages
Date:	1 April 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solution
Supervisors:	Erik Pätynen, Senior Lecturer
	Anne Pajala, Senior Lecturer

Reinforcement learning is a machine learning algorithm that has the potential to aid in the development of an AGI system. Among the various types of machine learning algorithms, RL is unique in that it explores the environment without prior knowledge and chooses the appropriate action while the others focus on handling the data.

AWS DeepRacer is a self-driving 1/18th size race car designed to simulate real-world conditions while testing RL models on a physical track. The project aims to gain a better understanding of RL, the mathematics underlying it, and to observe it in action by deploying the trained model in Amazon's DeepRacer automobile. [1].

To fine-tune the model, performance indicators such as the average reward per episode and cumulative reward were investigated. To gain a better understanding of the distribution of action spaces, Amazon's log analysis capabilities were used. Any wasted action was deleted for effective training based on the log analysis data. The model is uploaded as soon as the training was finished to test it in the race track.

The results may be utilized as general suggestions for training models and enhancing RL using AWS DeepRacer. By using the strategies described in the thesis, it is possible to develop more robust and stable models.

Keywords: Machine Learning, Neural Networks, Reinforcement Learning, Autonomous, DeepRacer, Amazon Web Service, automation.

List of Abbreviations

Contents

1 Introduction 1

1.1 Autonomous system and robotic 1
1.2 Overview of Machine Learning 1

2 Theoretical background 2

2.1 Reinforcement Learning 3
2.2 DeepRacer 5
2.2.1 The Vehicle 6
2.2.2 The environment 9
2.2.3 Model 9
2.2.4 Action Space 10
2.2.5 Reward Function 11
2.2.6 Hyperparameters 15
2.2.7 Underfitting and Overfitting Model 18

3 Methods 21

3.1 Anaconda 21
3.2 Network Architecture 22
3.2.1 PPO 22
3.2.2 SAC 23
3.3 Services Architecture 23
3.4 Sagemaker Notebooks for training 25

4 Results 26

4.1 Overview 26
4.2 Agent Parameters 30
4.3 Development 32
4.3.1 Optimize Race Line 32
4.3.2 Universal Model 34

5 Conclusion 39

References 41

List of Abbreviations

RL: Reinforcement Learning.

ML: Machine Learning.

NN: Neural Network.

GDBS: Gradient Descent Batch Size.

EB: Experience Buffer.

AWS: Amazon Web Services.

CSC: Common Service Center.

DIY: Do it yourself.

LiDAR: Light Detection and Ranging.

UI: User Interface.

PPO: Proximal Policy Optimization.

SAC: Soft Actor Critic.

1 Introduction

1.1 Autonomous system and robotic

Autonomous robots will become increasingly prevalent in the world we live in. From delivering food to our footsteps to driving us to work, autonomous technologies will play a significant role in our daily lives. ML and artificial intelligence advancements, combined with increased processing power, have enabled the fantasy of self-driving automobiles to become a reality. With increased air and land traffic, the deployment of autonomous vehicles will result in more effective resource utilization.

When a system performs all dynamic tasks on its own, it is said to be autonomous [2]. For example, a self-driving car should be capable of performing driving tasks in all driving scenarios only through the use of its automated system [2]. Autonomous robots are advantageous in situations (such as high air traffic) when human control is either impossible or impractical [3]. Consider air traffic, for example. Imagine thousands of airplanes flying in close proximity. A slight deviation in the trajectory of one airplane might have a domino effect on the entire swarm of airplanes. This information must be communicated to all other aircraft that may be impacted, and a decision must be made in a split second. This task is insanely difficult and has a significant probability of going wrong. However, if the airplanes have some degree of autonomy and are communicating with each other, the path planning algorithm would update instantly for each flight. As a result, the calamity can be averted considerably more easily and efficiently.

1.2 Overview of Machine Learning

ML is exploding in popularity in the modern day and is likely to continue to do so in the coming years. Our world is overloaded with data which is generated every day at a dizzying speed. According to CSC, a provider of Big Data and Analytics

Solutions, the amount of data generated by 2020 is estimated to be 44 times that in 2009 [4]. As a result, it is critical to comprehend data and gain insights in order to gain a deeper understanding of the human world. Today's data sets are so massive that conventional procedures are no longer applicable. In certain instances, manually evaluating data and constructing prediction models is impractical, time-consuming, and wasteful. On the other hand, ML delivers trustworthy, repeatable outcomes and learns from previous computations.

In recent years, ML has grown in popularity. Earlier, ML was strongly limited by the requirements for computer power and the quantity of accessible data. Additionally, the improvement of algorithms has contributed to the development of ML. There are several applications for ML, such as voice and image recognition, and new ones are continually being developed.

In contrast to conventional programming, ML involves the computer discovering solutions to given problems by analyzing the data it is provided. ML is distinct from traditional programming in that, rather than providing the computer with data and rules from which to generate results, the computer is provided with data and results from which the computer will determine the rules and methods for achieving the desired results on its own. Supervised learning, unsupervised learning, and reinforcement learning are all distinct classifications of ML.

This thesis will discuss RL with AWS DeepRacer. Recommendations for additional research are made, as this thesis can only provide a high-level overview of AWS DeepRacer. This thesis aims to provide a summary of AWS DeepRacer and its uses as well as to provide recommendations for effective RL rules to apply while creating models with AWS DeepRacer.

2 Theoretical background

Human beings have acquired knowledge through trial and error. The learning process is driven by reward mechanisms that incentivize particular behaviors. The goal is to increase the number of positive responses and decrease the

number of negative responses through iterative methods. This teaches individuals how to communicate with their world and how to overcome complicated barriers. [5]

Reinforcement Learning RL is based on the concept of trial and error as a result of interactions with the environment.

2.1 Reinforcement Learning

Reinforcement Learning RL is a useful framework for decision-making in scenarios where an agent communicates with its environment via trial and error to determine the most effective action. RL is concerned with sequential decision-making in order to accomplish its objectives.

RL is similar to how biological systems adapt to the environment via trial and error. The most common instance is training a dog. The trainer usually uses treats to reward the correct behavior when training a dog to obey certain commands. Positive Reinforcement is used in psychology to refer to behaviors that are encouraged, whereas Negative Reinforcement is used to refer to behaviors that are discouraged through punishment. [5]

RL is, in a nutshell, the process by which a decision-maker known as an agent gets information about its environment and learns to pick behaviors that result in the greatest reward for the agent. For the previous example, agent is a dog. Environment is the “world” which agent interacts, such as a garden. Action are performed by the agent in the environment, such as running around, sitting, or playing balls. Rewards are issued to the agent for performing good actions. [5]

Figure 1 shows the simplest representation of RL framework

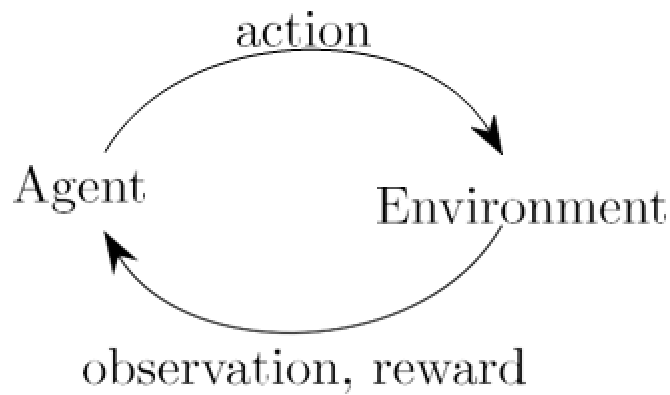


Figure 1: Agent-Environment Loop. [6]

At first, the agent takes the state S0 from its environment, which may include information such as the captured image, its speed, and other sensory data. The agent then performs action A0 on the specified state S0. The environment changes state to S1 as a result of the action, and the environment rewards the agent with a reward R1. This loop repeats until the episode concludes, returning the current state (S0) and action (A0), as well as the future state (S1) and reward (R1). Figure 2 shows the entire process in detail.

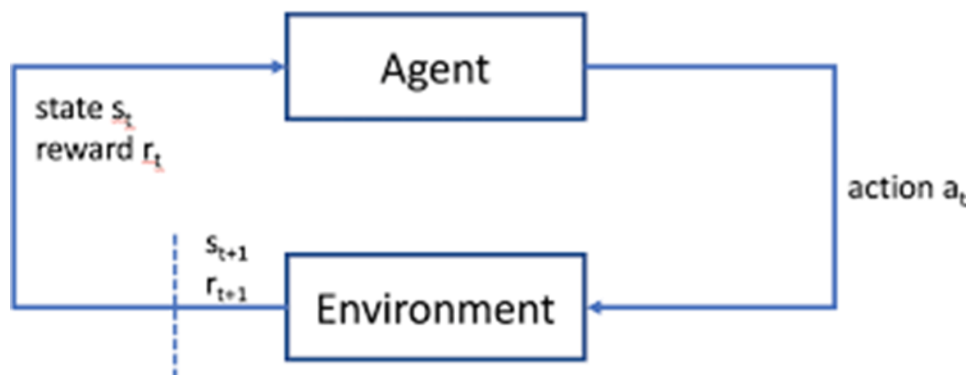


Figure 2: Reinforcement Framework

A typical illustration of this is a mouse in a maze. The mouse's objective in figure 3 is to collect as much cheese as possible without being captured by the

cat. Collecting cheese nearby the mouse is more rewarding than trying to get cheese nearby the cat.

Figure 3: A game where a mouse collects cheese in a maze. [7]

2.2 DeepRacer

The agent in DeepRacer is represented by a NN whose function approximates the agent's policy. The environment state corresponds to the image captured by the vehicle's front camera (see Figure 4), while the agent's actions are defined by speed and steering angle.

If the agent completes the race, it will earn good prizes; if it does not, it will get negative rewards as punishment. An episode begins by positioning the agent along the track. The episode concludes with the agent completing a lap, going off-track, or colliding with an item or another vehicle. [1]

Figure 4. DeepRacer state

2.2.1 The Vehicle

DeepRacer is a fully autonomous 1/18th-scale model of a race car developed by Amazon that uses RL to run. The purpose of the product is to educate users at all stages of RL [8]. Through their DeepRacer Console, customers can conduct RL. Users can deploy the model to the vehicle after being trained. Figure 5 shows the front and side views of the physical car used to test the model in a real-world context and all of the DeepRacer parts in figure 6 separately. Information in details can be found in table 1 and 2. [1]

(a) Front View (b) Side view

Figure 5: DeepRacer Car Views.

Figure 6: AWS DeepRacer parts.

The car is capable of autonomous operation by doing inference using the RL model submitted by the user [8]. Additionally, it can be operated manually via the internal console. The vehicle is propelled by a brushed motor, and its speed is controlled by a voltage regulator. The servomechanism controls the steering [8]. Only time-trial race setting is used in this thesis for its purpose. However, LiDAR was utilized to ensure the model's durability, as creating a solid DIY physical track is tough. The forward-facing stereo cameras assist the vehicle in determining the depth information included in the photos.

Table 1: AWS DeepRacer vehicle parts 1. [1]

Components	Comments
Vehicle Chassis [1]	Includes a front-mounted camera for capturing vehicle driving experiences and the compute module for autonomous driving. You can view images captured by the camera as a streaming video on the vehicle's device console. The chassis includes a brushed electric motor, an electronic speed controller (ESC), and a servomechanism (servo)
Vehicle body shell [2]	Remove this when setting up the vehicle.
Micro-USB to USB-A cable [3]	Use this to support USB-OTG functionality.
Compute battery [4]	Use this to power the compute module that runs inference on a downloaded AWS DeepRacer reinforcement learning model.
Compute battery connector cable [5]	Use this USB-C to USB-C cable to connect the compute module with the battery. If you have a Dell compute battery, this cable will be longer.
Power cable [6a]	Use this to connect the power adaptor to a power outlet.

Table 2: AWS DeepRacer vehicle parts 2. [1]

Components	Comments
Vehicle Chassis [1]	Includes a front-mounted camera for capturing vehicle driving experiences and the compute module for autonomous driving. You can view images captured by the camera as a streaming video on the vehicle's device console. The chassis includes a brushed electric motor, an electronic speed controller (ESC), and a servomechanism (servo)
Vehicle body shell [2]	Remove this when setting up the vehicle.
Micro-USB to USB-A cable [3]	Use this to support USB-OTG functionality.
Compute battery [4]	Use this to power the compute module that runs inference on a downloaded AWS DeepRacer reinforcement learning model.
Compute battery connector cable [5]	Use this USB-C to USB-C cable to connect the compute module with the battery. If you have a Dell compute battery, this cable will be longer.
Power cable [6a]	Use this to connect the power adaptor to a power outlet.

2.2.2 The environment

The AWS DeepRacer console is the primary technique for developing a model to run the car autonomously. The console is an interactive platform that enables users to monitor and evaluate the model's training and assessment phases while also providing the model's primary log data. Figure 7 illustrates the console UI.

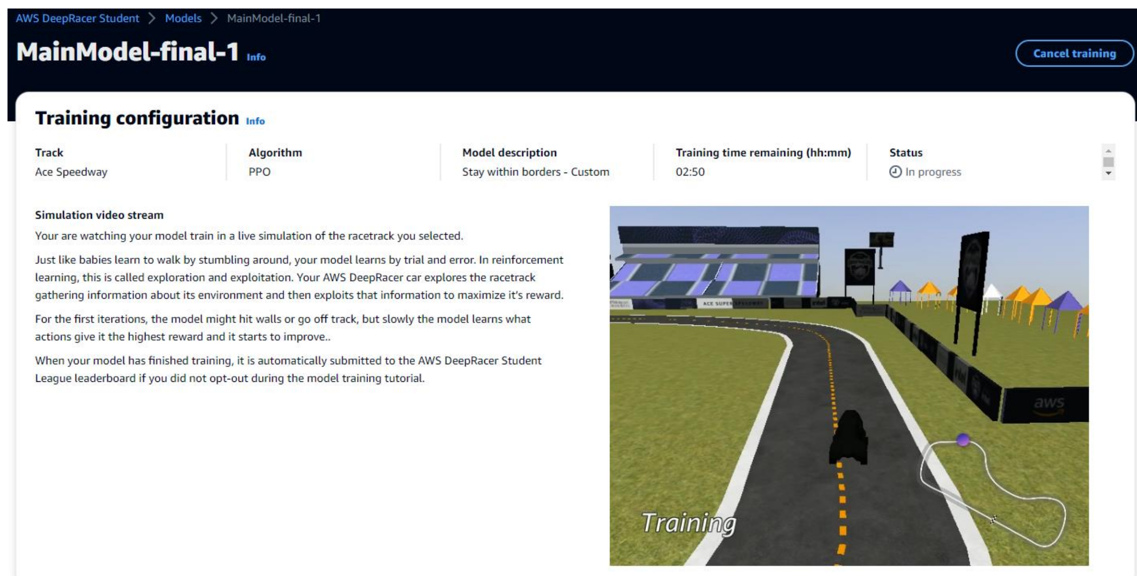


Figure 7: The UI during training.

2.2.3 Model

DeepRacer's model is a critical component. The model contains all the characteristics and settings that comprise the agent's operating environment. It is composed of three distinct parts: state, action, and reward for acting. The policy is the agent's decision-making strategy, with the state of the environment as the data and the desired action as the output. In DeepRacer, the policy is frequently represented by a deep NN, also known as the RL model. Every training session produces a single model. Even if the training is terminated prematurely, a model is generated. Model is immutable, and is unable to modify after generation. However, it is possible to clone a model and then train it with different parameters. [1]

2.2.4 Action Space

RL requires a limited set of behaviors or options from which an agent may pick while interacting with its environment. That is the definition of action space which may be either continuous or discrete. [1].

A discrete action space in a finite set provides all the activities an agent may do each state. The NN of AWS DeepRacer responds to each situation on the track by allocating the agent a specific speed and angle of turn based on data from its camera or LiDAR sensor. The options are presented in predefined action numbers, each of which corresponds to a specific steering angle and throttle combination for the car. Table 3 illustrates the default discrete action space.

Table 3: AWS DeepRacer Discrete Action Space. [8]

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

A continuous action space, as contrast to a discrete action space, lets the agent to choose an action from a value range per state [8]. 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. In addition, it lacks a pre-set list of actions, as indicated in Figure 8.

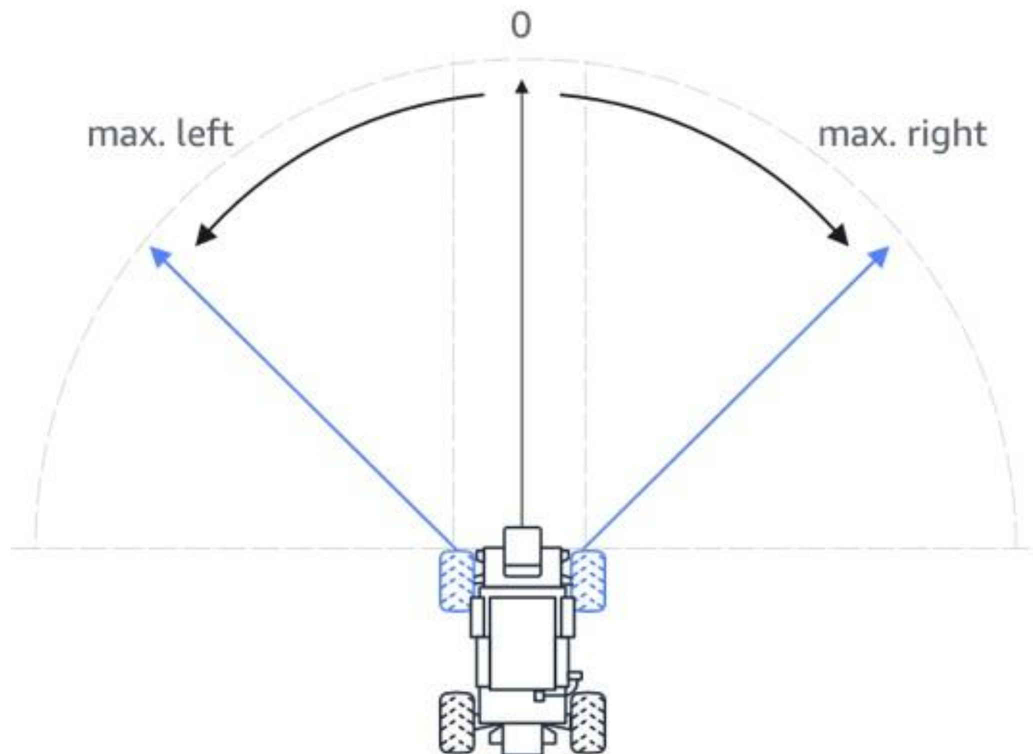


Figure 8. Continuous action space.

While continuous action space allows for better optimization since it contains a range of values from which the agent can choose the optimal value pairings to increase performance, the discrete action space constrains the agent to choose from a limited list of permissible actions. However, because a continuous action space has a range of values, the agent must train longer, increasing resource use.

2.2.5 Reward Function

Policy

The agent's policy outlines how it pick an action in given circumstances. The policy would select activities that optimize total benefit. It will not take the acts

that result in an immediate increase in reward. In a nutshell, the policy always prioritizes accomplishing its primary goal, meaning that it may select behaviors that are suboptimal in their current state. [5]

Value Function

The value function represents the state's lasting quality. It is the cumulative benefit that the agent anticipates from the current condition to the future state. Reward is a measurement of the agent's instant performance in the present state. The value function quantifies the agent's performance in the long term. This reveals that a low reward is not always associated with a low reward function or in any other way [5].

Reward

Each step requires the agent to do an action, and the value function indicates how effective it was. This is referred to a reward. As stated earlier, the agent's aim is to optimize its total benefits.

Rewards may award regularly or infrequently. Dense incentives are those that appear frequently, while sparse rewards are those that appear infrequently [5]. In DeepRacer, for instance, remaining on race is a dense reward since the vehicle will either stay on track for the full length of the race or will drive off. Dodging an object in DeepRacer is a meagre prize, as even with six objects on the track, they are still quite close to the track.

Reward Function

The reward function is a critical component of the AWS DeepRacer platform because it effectively drives the agent to act. In the instance of the DeepRacer, the reward function is a Python function that accepts a dictionary object of parameters providing information about the current state and produces a numerical estimate of the reward. With the function, the user can "reward" or "penalize" a particular activity. The user may provide a set reward or one that is dependent on the parameters. The following diagram illustrates the overall structure of a reward function. Below is a sample outlines of a reward function.

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

Figure 9: Basic outline of a reward function.

The params dictionary stores the key-value pairs corresponding to the state measurements listed in table 4. It is not necessary to incorporate all of these parameters into the reward function's design. A basic reward function may also result in acceptable performance. However, some or all of the parameters may be useful for creating reward functions for complex tracks that require the agent to focus on multiple parameters concurrently to complete the race.

Table 4: Key-value pairs in params [8].

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

Below is a simple example of a reward function which encourages the DeepRacer car to follow the centerline. The reward function calculates a reward depending on the distance between the vehicle and the track's centreline. The closer to the centerline the car is, the higher reward is given to it. This reward function is at the very basic level contains only two parameters: track_width and distance_from_center

```
1  def reward_function(params):
2      # Example of rewarding the agent to follow center line
3
4      # Read input parameters
5      track_width = params['track_width']
6      distance_from_center = params['distance_from_center']
7
8      # Calculate 3 markers that are at varying distances away from the
         center line
9      marker_1 = 0.1 * track_width
10     marker_2 = 0.25 * track_width
11     marker_3 = 0.5 * track_width
12
13     # Give higher reward if the car is closer to center line and vice
         versa
14     if distance_from_center <= marker_1:
15         reward = 1.0
16     elif distance_from_center <= marker_2:
17         reward = 0.5
18     elif distance_from_center <= marker_3:
19         reward = 0.1
20     else:
21         reward = 1e-3 # likely crashed/ close to off track
22
23     return float(reward)
24
```

Figure 10: Sample code from AWS for “Follow the centreline” function

2.2.6 Hyperparameters

Entropy

The entropy measures the uncertainty in policy distribution. The model with a large entropy shows higher variance behaviors comparing to the one with a lower entropy. More increased entropy allows a more complete search of action space. [1]

Data point

Is defined as a set of [s, a, r, s’] which respectively stands for state, action, reward and new state. Data point is also known as experience. [1]

Episode

During an episode, the agent either performs an action in its environment to collect rewards or gets out of it. Episode durations may change from each other. A set of data points constitutes an "episode".

Experience buffer

The EB contains a set amount of sorted data points gathered from episodes of varying lengths during the training time. It corresponds to images used as the source for DeepRacer's input data. [1]

Batch

A subset of EB represents a section of the simulation during a specified period and are used to adjust the weights. [1]

Training data

A collection of batches randomly picked from the EB. The training data is utilized to update the weights of the policy network. [1]

Gradient descent batch size

GDBS is the number of recently sampled agent data points from an EB that are utilized to update the NN’s weights. Random sampling reduces the connection. Larger sizes give more effective and reliable weight updates, but the training period of the model is likely to be longer and slower. [1]

Epoch

The number of epochs indicates how often training data is transmitted through the NN to update the weights. [1]

Learning rate

The learning rate determines how quickly the model learns by adjusting the amount by which GDBS alters the values of NN weights. Due to the frequently updating weights, a higher learning rate allows quicker training. However, if it is set too high, the model cannot converge. [1]

Discount factor

Specifies the range of future state rewards that the agent will estimate using the reward function when deciding whether to take an action. The higher the value, the more steps are considered while acting, but cause the training is slower as a result. [1]

Loss type

The loss function is responsible for updating the NN's weights. The objective of the loss type is to perform cumulative adjustment in the policy over time, such that it transitions from random to policy influenced actions. DeepRacer supports two types of losses: Huber and Mean squared error. [1]

Both function identically when updating the weights is minor. When the weights are changed more significantly, differences occur. Compared to the Huber loss, the mean squared error starts growing in larger increments.

Episode between each policy updating iteration

The size specifies the number of data points contained within an EB. Larger buffers bring in more stable but slower updates. [1]

2.2.7 Underfitting and Overfitting Model

When a statistical model is trained using an excessive amount of data, it is said to over fit. When a model is trained with such a large amount of data, it begins learning from the noise and incorrect data entries in the data set. The model fails to appropriately classify the data because there are too many details and noise. Non-parametric and non-linear approaches are responsible for overfitting since these ML algorithms have the aim to develop the model based on the dataset, allowing them to create unrealistic models.

It is critical to master the art of detecting overfitting and underfitting in ML. Both of them cause significant effects on how your model performs unless you take actions to prevent them or at the very least minimize them on model performance.

Overfitting

Overfitting happens when a machine learning model attempts to cover more data points than are present in the supplied dataset. Due to this, the model begins caching noise and incorrect values included in the dataset, which reduces the model's accuracy and performance. The risk of overfitting increases according to the amount of model training. The more we train our model, the higher the risk of an overfitted model. Figure 11 below shows an example graph of an overfitting model where the model attempts to handle the whole dataset. [9]

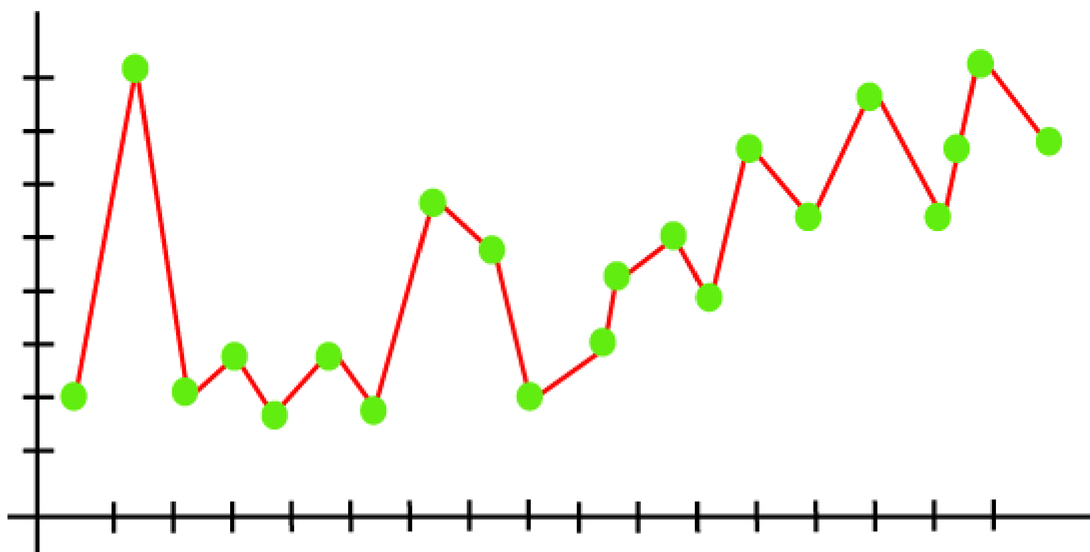


Figure 11: A graph of an overfitting model.

In RL, an overfitting model performs perfectly within its training environment and drops its performance dramatically in any different environment. In the thesis project case, overfitting model can be recognized when testing the DeepRacer car performance on other track.

Undefitting

Underfitting happens when a machine learning model fails to catch the fundamental trend in the dataset. The training progress sometimes can be interrupted early to avoid overfitting, so the model may not gain enough knowledge from the dataset. Consequently, it may be unable to identify the appropriate fit for the predominant trend in the data. Underfitting happens when a model cannot learn sufficiently from the dataset, leading to decreased accuracy and inaccurate predictions. Figures 12 below shows an example graph of an overfitting model. [9]

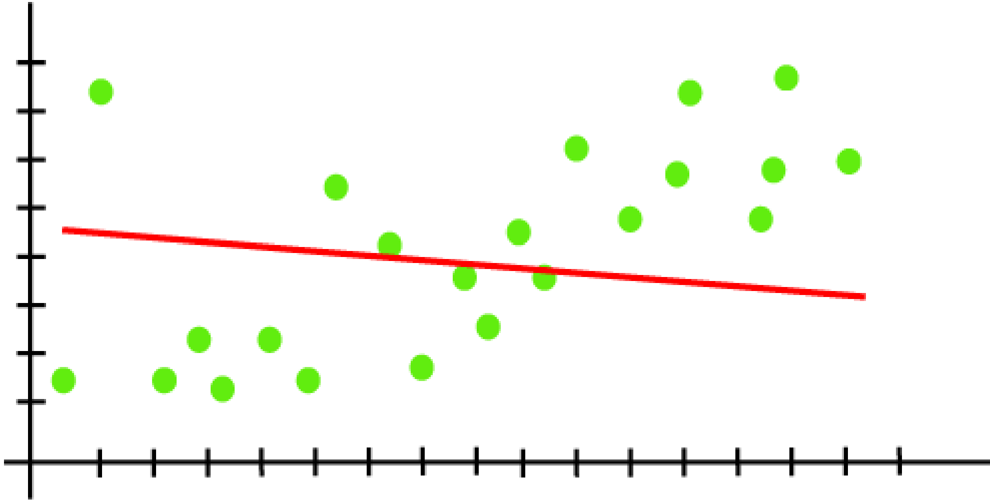


Figure 12: A graph of an underfitting model

In RL, underfitting occurs when the model stop training before exploring all the action space and environment. The agent's model is immediately identifiable when the agent is unable to complete even one lap.

Convergence

Convergence indicates that the model is operating optimally. Convergence is the perfect position to train a model. After convergence, it is predicted the model will begin to overfit and its performance in environments other than the training environment would steadily drop. As the agent traverses its action zone, entropy decreasing with time is necessary. The model has converged when the model's rewards, average progression, and entropy begin to equalize. [10]

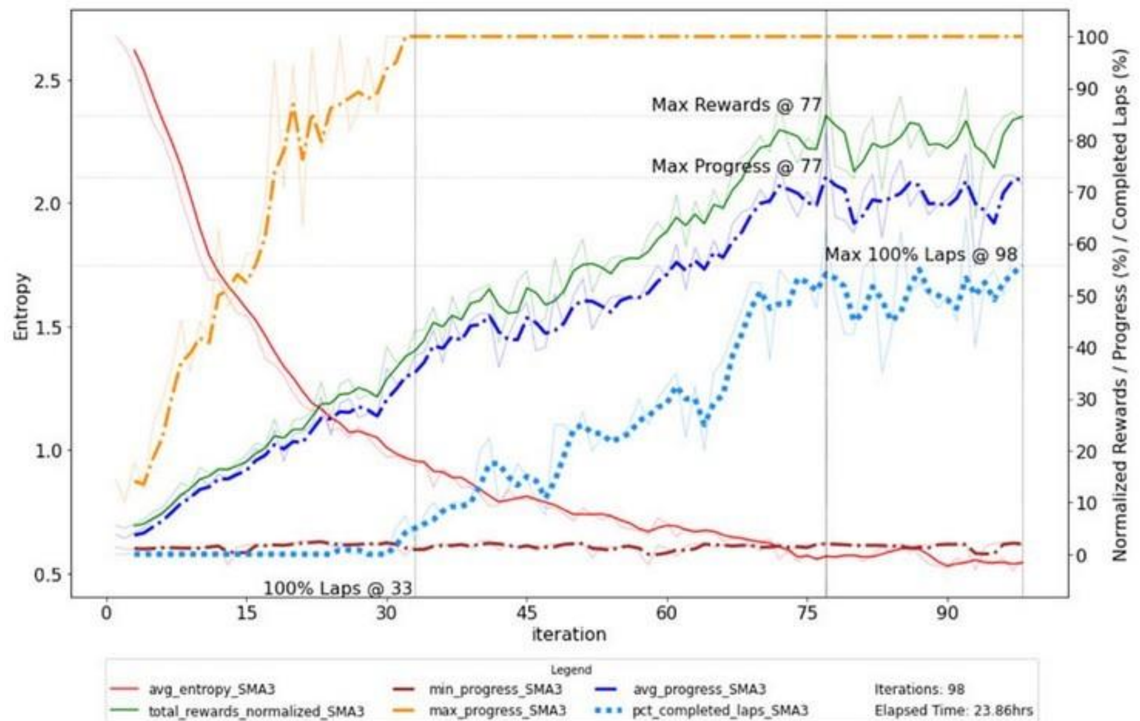


Figure 13: The convergence at three peaks at the end of the training session.

3 Methods

3.1 Anaconda

Conda is a free and open-source for managing packages and environments system on Windows, macOS, and Linux. Conda installs, executes, and updates packages and their dependencies in a matter of seconds. Additionally, it simplifies the process of creating, saving, loading, and switching between local computer environments. Although it was designed to package and distribute Python applications, it is capable of packaging and distributing software written in any language. The Sagemaker and Robomaker logs were analyzed using the Anaconda.

3.2 Network Architecture

Proximal Policy Optimization (PPO) is the default algorithm in DeepRacer. AWS just introduced a new algorithm named the Soft Actor Critic (SAC) algorithm. PPO and SAC are similar in that they both simultaneously learn a policy and value function. However, their techniques differ significantly in three key methods, as illustrated in Table 5.

Table 5: Comparing PPO and SAC

PPO	SAC
Works in both discrete and continuous action spaces	Works in a continuous action space
On-policy	Off-policy
Uses entropy regularization	Adds entropy to the maximization objective

As illustrated in the table, each algorithm has its own set of requirements and techniques for learning. It is unnecessary to go through the mathematical details of the algorithms in this context; rather, the emphasis should be on how they were used in DeepRacer and the benefits of either algorithm.

3.2.1 PPO

PPO makes use of two NNs, called the policy and value networks. The policy network, alternatively referred to as the actor network, determines the action to execute in response to the input. The value network also referred to as the critic network, uses the inputs to estimate the cumulative reward [12]. The policy network is the one that communicates with the simulator and is applied to the car. Figure 14 illustrates the network's architecture.

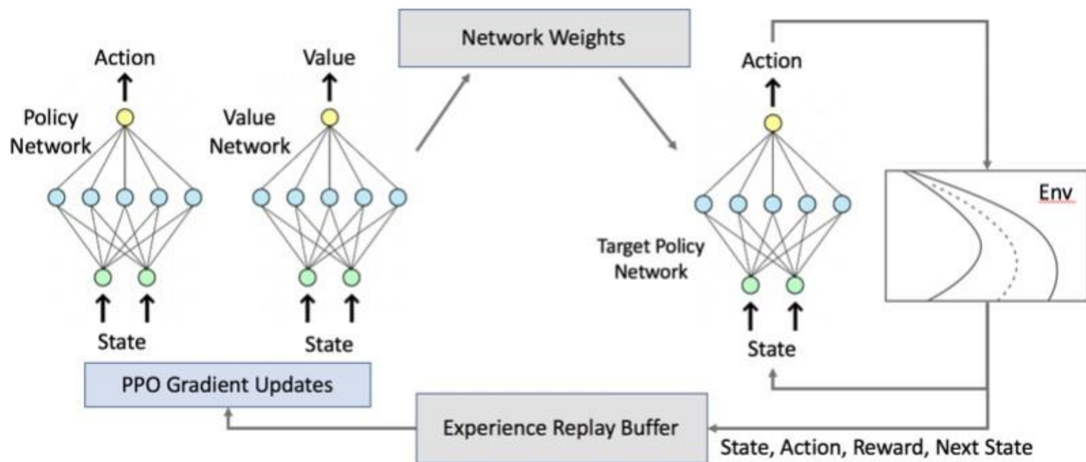


Figure 14: Network Architecture. [12]

PPO applies on-policy learning, which means that it derives its value function from the present policy through exploration. On-policy algorithms typically require more data for training but are more stable in return. Over time, the policy becomes less random, as weight changes incentivize exploiting previously discovered benefits.

3.2.2 SAC

SAC applies off-policy learning, which incorporates behavior policy and objective policy. The behavior policy communicates with the environment and collects data used to formulate the target policy. The target policy can benefit from the experiences of earlier policies. Off-policy algorithms are generally less stable, but they require less training data. [1]

3.3 Services Architecture

SageMaker and RoboMaker, as well as other AWS cloud services, are included in the AWS Deepracer environment. SageMaker is an AWS ML platform that provides model training, whereas RoboMaker is a cloud service for the development, testing and deployment of robotic products. DeepRacer integrates them to train models and build virtual agents and environments via SageMaker

and RoboMaker. It stores trained models, as well as training logs and other associated artifacts, on the cloud storage platform S3 [11].

AWS RoboMaker builds a virtual environment for the agent to run on a predefined track within the AWS DeepRacer architecture. SageMaker's trained policy network model is followed by the agent's actions. Each lap is marked by a single episode. The course is organized into episodes that are each comprised of a defined number of steps. Redis, an in-memory database, is used by AWS DeepRacer as an EB for selecting training material for the policy NN to train. Redis caches "experiences" in each segment as an EB. The EB is described as a sorted list of tuples representing each step's state, action, reward, and new state. SageMaker extracts training data from the EB at random intervals and feeds it to the NN in batches to update the weights. SageMaker then uses the improved model saved in S3 to build new experiences. This loop will not stop unless the training is finished. The EB starts with random actions in the very first episode. As training advances, there should be less unpredictability in the agent's activities. This architecture is shown in figures 14 and 15. This configuration is advantageous because it allows the execution of numerous simulations simultaneously to train a model on various segments of a track or multiple tracks.

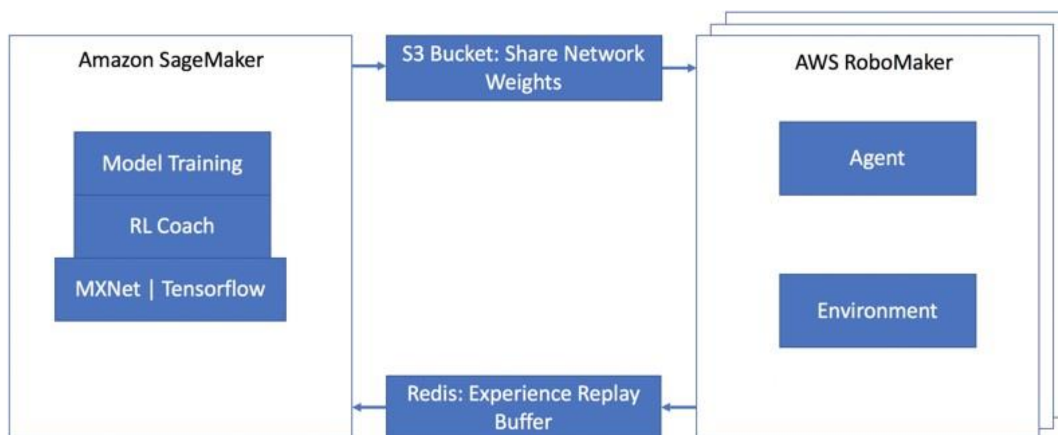


Figure 14: AWS DeepRacer Service Architecture

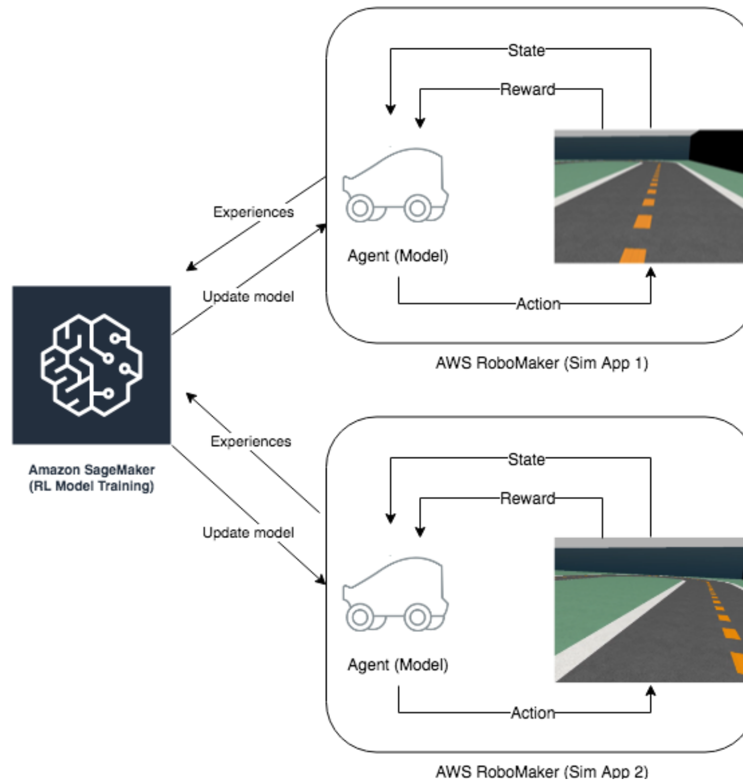


Figure 15: AWS DeepRacer Schematics.

3.4 Sagemaker Notebooks for training

The training approach mentioned before included the DeepRacer console, which integrates the training and evaluation of DeepRacer models. The console utilizes SageMaker and RoboMaker in the background to enable seamless model training and evaluation [8]. The SageMaker notebook allows us to "jailbreak" AWS DeepRacer by granting us additional control over the training/simulation process and RL algorithm customization. In a previous section, we discussed how to distribute RL training across SageMaker and two RoboMaker simulation environments that perform rollouts – the execution of a specified number of episodes using the existing model or policy. The rollouts gather agent experiences (state-transition tuples) and send them to SageMaker for training purposes. SageMaker modifies the model policy, which is

subsequently utilized to perform the subsequent rollout sequence. This training cycle is repeated until the model achieves convergence [8].

4 Results

4.1 Overview

The first stage in this project was to develop an efficient reward function. Initially, the network was trained using the default reward function. Additionally, the hyperparameters were set to their default values. The training session was scheduled for one hour. Following training, the model was tested for three trials on the same track. Figure 16 illustrates the overall reward for each episode and the percentage of completion.

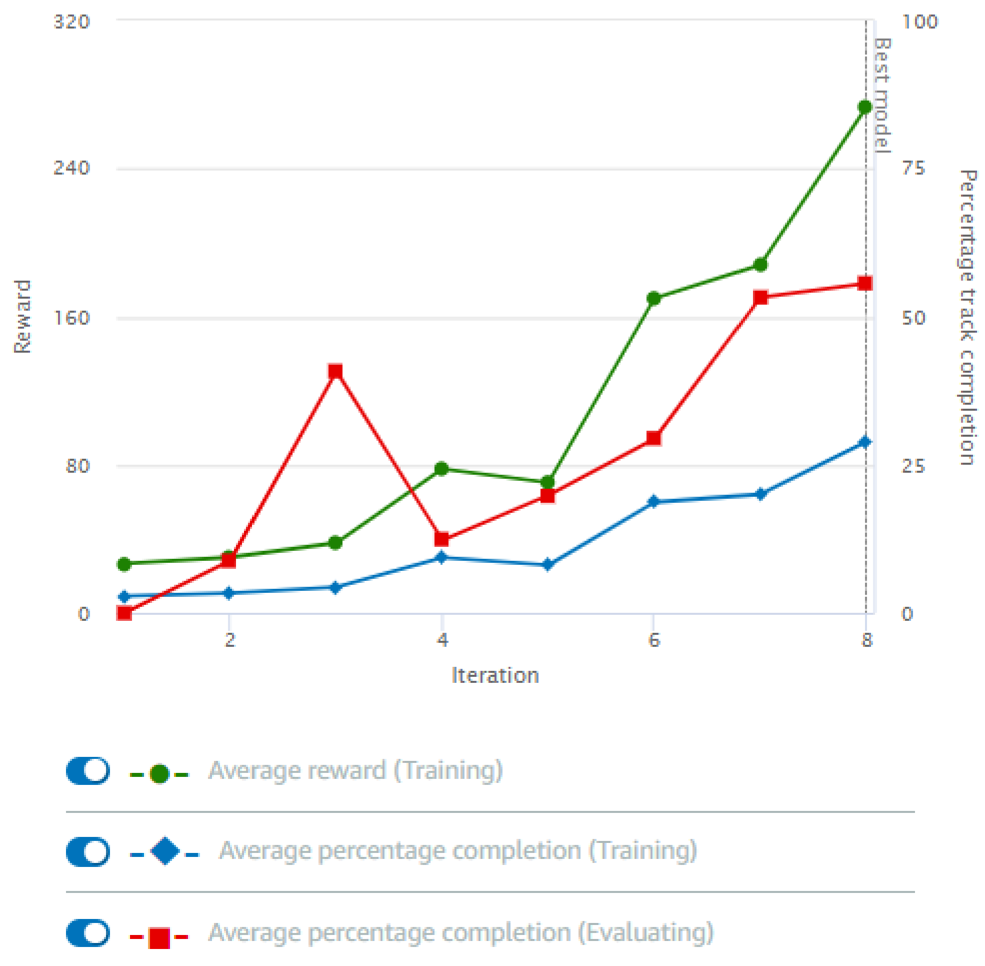


Figure 16: Training Stats.

Figure 17 illustrates the evaluation metrics. The evaluation trials were unable to finish the track in its entirety. The second trial completed 94 percent of the track before deviating. The most possible reason for the failure was insufficient 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 17: Evaluation Stats

Two hours of training with default hyperparameters is recommended. Although modifying hyperparameters may be necessary, it is important to remember that modifying default hyperparameters will change the required minimum amount of training time to convergence. In general, the effect of two hours of training on easier time trial courses can indicate if the model needs retraining or just analyze the log and move on.

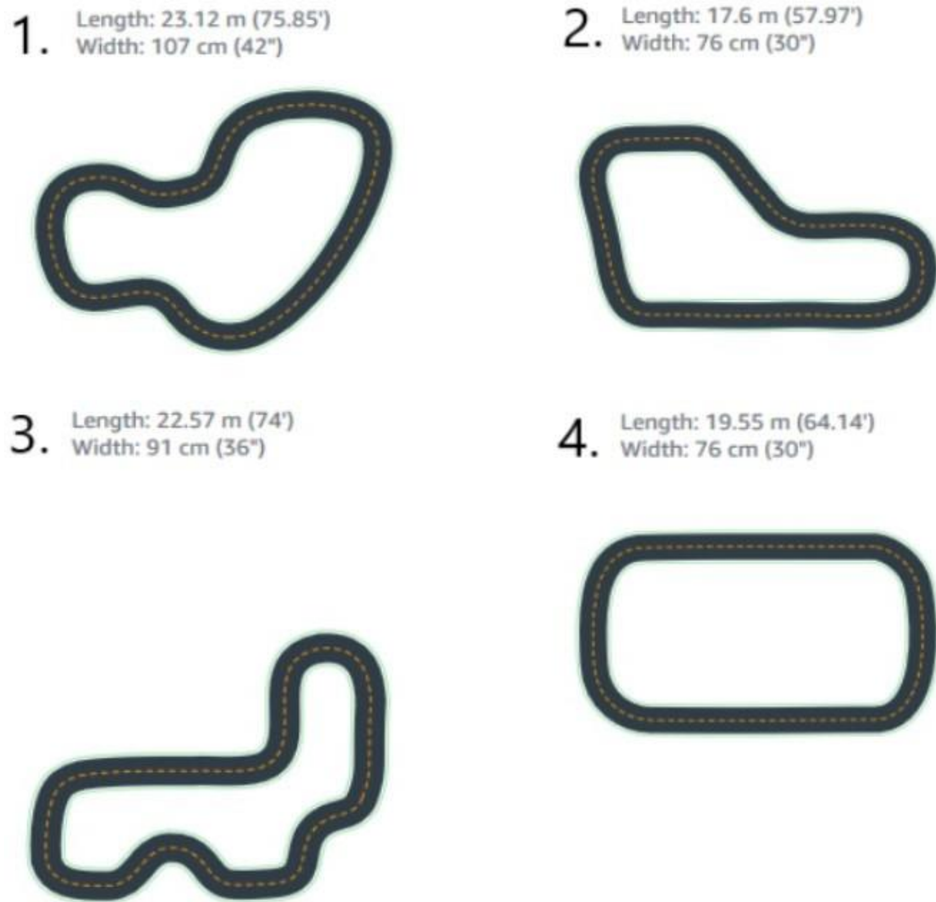
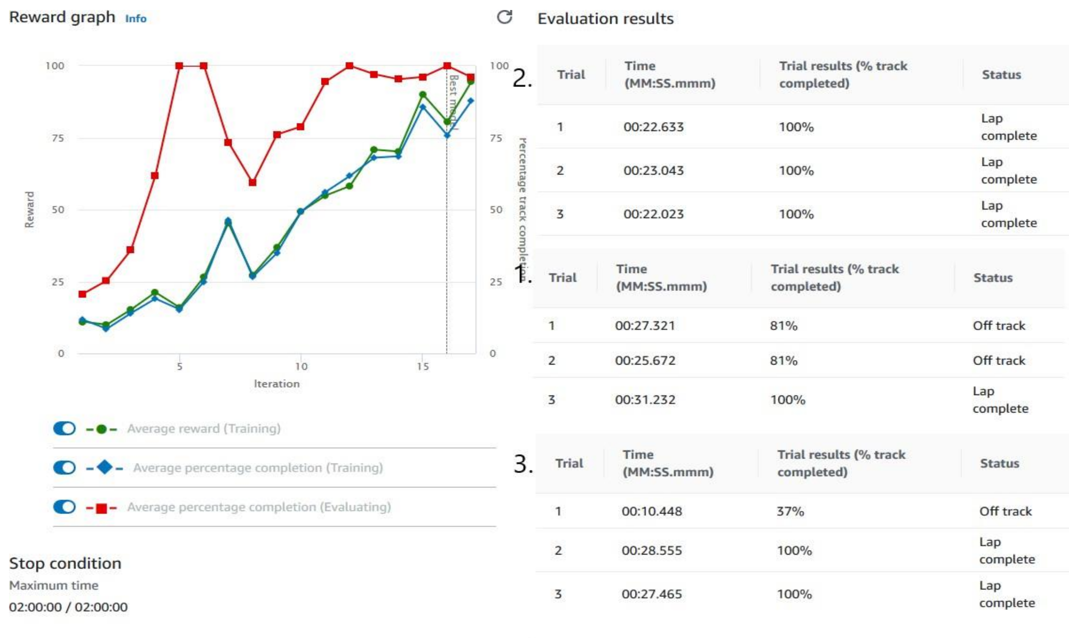


Figure 18: Racing tracks



2.

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:22.633	100%	Lap complete
2	00:23.043	100%	Lap complete
3	00:22.023	100%	Lap complete

1.

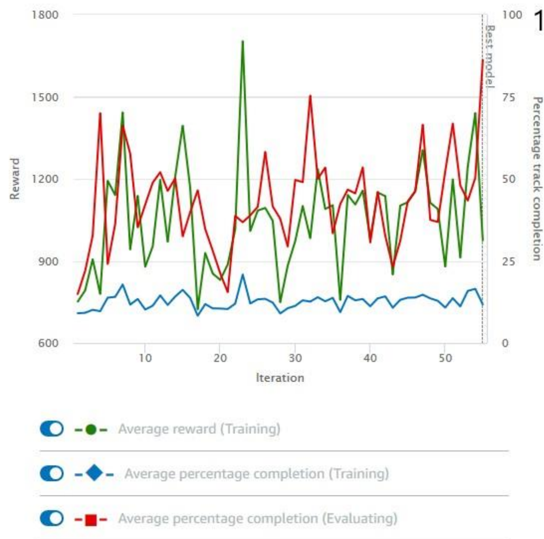
Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:27.321	81%	Off track
2	00:25.672	81%	Off track
3	00:31.232	100%	Lap complete

3.

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:10.448	37%	Off track
2	00:28.555	100%	Lap complete
3	00:27.465	100%	Lap complete

Figure 19: The very first models.

Reward graph Info



Stop condition
Maximum time
02:00:00 / 02:00:00

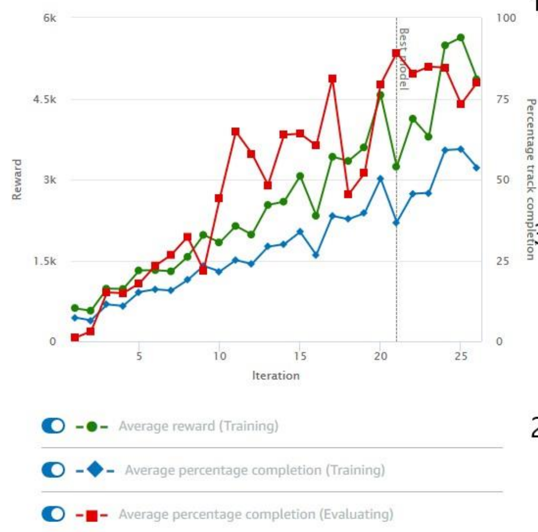
Evaluation results

1.

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:13.328	85%	Off track
2	00:12.055	72%	Off track
3	00:13.801	93%	Off track

Figure 20: An unstable model when training with high entropy hyperparameter.

Reward graph Info



Stop condition
Maximum time
02:00:00 / 02:00:00

1.

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:15.917	100%	Lap complete
2	00:15.754	100%	Lap complete
3	00:16.467	100%	Lap complete

3.

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:01.294	6%	Off track
2	00:04.073	22%	Off track
3	00:04.076	28%	Off track

2.

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:01.139	6%	Off track
2	00:01.212	8%	Off track
3	00:01.199	7%	Off track

Figure 21: More stable model after an adjust hyperparameter

4.2 Agent Parameters

Both continuous and discrete action spaces have distinct advantages. The majority of models developed for this research were discrete models. It looked like discrete models did a little higher performance than continuous models at the start of the testing period, but this could be because they had less time to learn. Continuous models will almost certainly train slower. There are insufficient data to determine which type of action space is preferred.

The most key parameter in an agent's parameters is its speed. The speed of the model has the greatest effect on its performance. It is possible that the model will make fewer mistakes in hard turns if its speed is increased. In addition, training may take longer since the model will deviate off course more often during the first phases of the training process as well as the agent may begin to drift in corners if the steering angle is very high. The models become less stable as their speed rises. The experiment provided no solid evidence for the location of the agent's speed sweet spot.

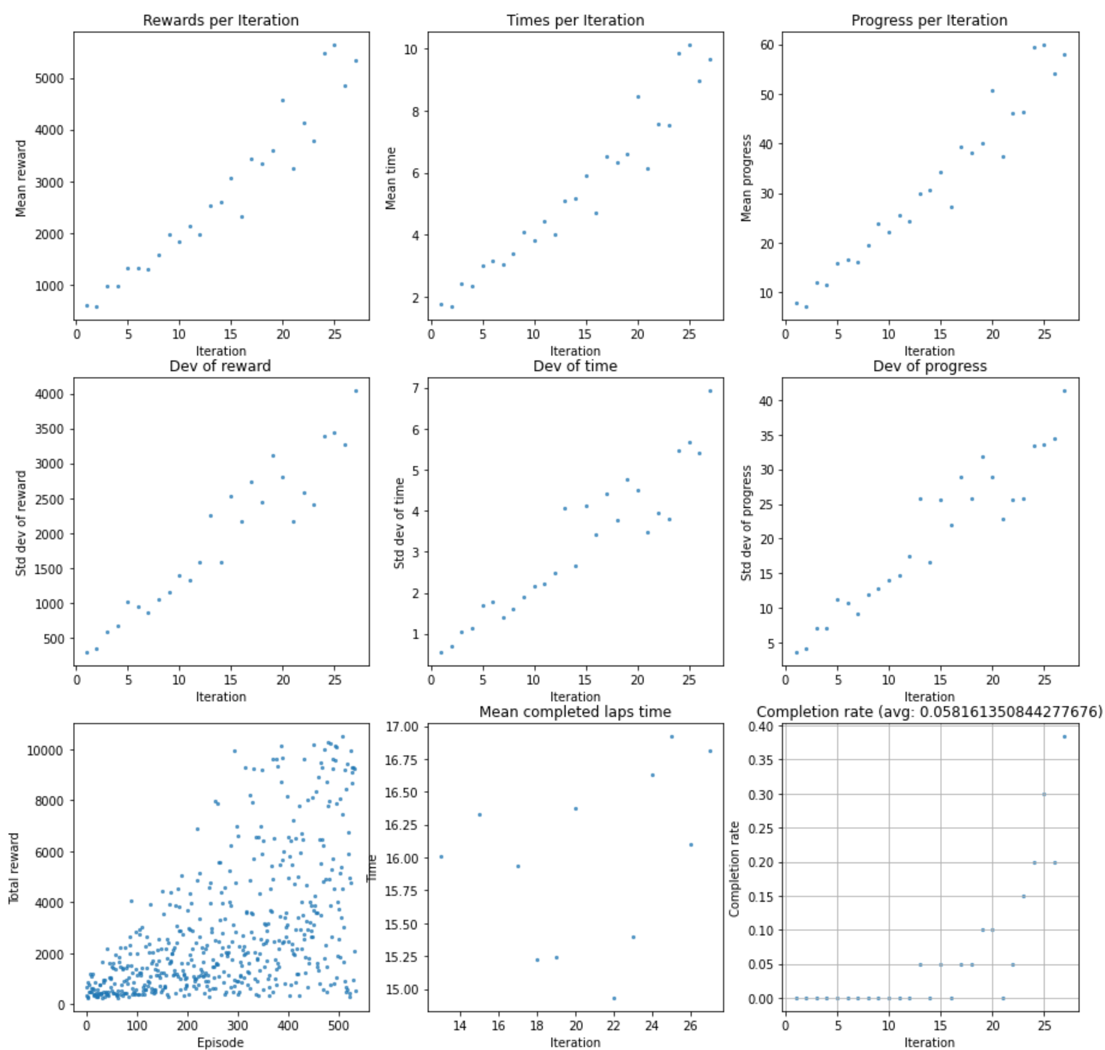


Figure 22: Model metrics

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	Episode #	Training It	Epoch	In Heatup	ER #Transi	ER #Episod	Episode Le	Total step	Epsilon	Shaped Tr	Wall-Clocl	Discounte	Discounte	Discounte	Discounted Return/Min		
2	1	0	0	0	952	1	952	952	0.1	6328	0.014751	2388.322	1075.116	4062.438	107.2		
3	2	0	0	0	1483	2	531	1483	0.1	3523.99	51.54001	1429.643	826.92	2770.388	-39.83043		
4	3	0	0	0	2446	3	963	2446	0.1	6522.4	144.5621	2479.307	1116.295	4143.109	107.2		
5	4	0	0	0	3430	4	984	3430	0.1	6491.987	239.336	2450.497	1070.938	4088.871	107.2		
6	5	0	0	0	4394	5	964	4394	0.1	6515.2	332.2045	2515.707	1130.584	4098.251	107.2		
7	6	0	0	0	5356	6	962	5356	0.1	6407.2	424.4243	2516.754	1126.015	3994.067	107.2		
8	7	0	0	0	5671	7	315	5671	0.1	1961.59	455.0547	863.48	530.0335	1692.978	-39.83043		
9	8	0	0	0	5969	8	298	5969	0.1	1843.993	484.2054	785.4539	509.7603	1613.152	-27.92312		
10	9	0	0	0	6722	9	753	6722	0.1	5075.982	556.5823	1964.767	1078.4	3603.716	-71.4091		
11	10	0	0	0	6984	10	262	6984	0.1	1456.788	581.6819	571.2216	397.6045	1309.746	-47.74881		
12	11	0	0	0	7934	11	950	7934	0.1	6587.2	672.4895	2487.229	1212.927	4232.793	107.2		
13	12	0	0	0	8112	12	178	8112	0.1	710.385	689.3337	287.2046	191.2106	660.5831	-59.59671		
14	13	1	0	0	0	0	969	9081	0.1	6616	2737.018	2556.812	1222.868	4252.722	107.2		
15	14	1	0	0	968	1	968	10049	0.1	6443.2	2836.287	2423.433	1085.366	4105.524	107.2		
16	15	1	0	0	1944	2	976	11025	0.1	6781.6	2929.318	2548.349	1169.107	4303.013	107.2		
17	16	1	0	0	2624	3	680	11705	0.1	4182.38	2994.068	1646.743	893.7625	3068.729	-47.74881		
18	17	1	0	0	3025	4	401	12106	0.1	2679.193	3032.341	1151.692	649.2728	2221.806	-27.8992		
19	18	1	0	0	3997	5	972	13078	0.1	6882.4	3126.181	2609.467	1211.076	4354.703	107.2		
20	19	1	0	0	4307	6	310	13388	0.1	1918.39	3155.414	850.5184	489.7608	1658.044	-39.83043		
21	20	1	0	0	5295	7	988	14376	0.1	6442.371	3250.855	2470.964	1132.314	4066.971	107.2		
22	21	1	0	0	6124	8	829	15205	0.1	5663.192	3330.277	2199.272	1126.816	3847.687	-31.8962		
23	22	1	0	0	6712	9	588	15793	0.1	4031.198	3387.389	1667.584	876.717	3056.769	-7.997999		
24	23	1	0	0	7682	10	970	16763	0.1	6615.189	3480.457	2470.43	1199.496	4227.323	107.2		
25	24	1	0	0	8666	11	984	17747	0.1	6479.984	3575.229	2422.474	1174.906	4104.475	107.2		
26	25	1	0	0	9639	12	973	18720	0.1	6411.174	3669.422	2467.023	1154.389	4018.794	79.63557		
27	26	2	0	0	805	1	805	19525	0.1	5110.378	5982.245	1894.271	1032.552	3592.672	-39.83043		
28	27	2	0	0	1768	2	963	20488	0.1	6544	6075.213	2453.851	1118.058	4189.316	107.2		
29	28	2	0	0	2743	3	975	21463	0.1	6666.4	6169.282	2511.99	1138.943	4225.657	107.2		
30	29	2	0	0	3186	4	443	21906	0.1	2940.799	6211.904	1247.715	697.0297	2392.844	-4.001		
31	30	2	0	0	4178	5	992	22898	0.1	6832	6306.688	2575.252	1163.294	4285.918	107.2		
32	31	2	0	0	4443	6	265	23163	0.1	1659.199	6332.265	770.2392	416.0963	1458	-4.001		
33	32	2	0	0	4684	7	241	23404	0.1	1431.989	6355.554	651.0892	394.5237	1277.63	-43.7916		
34	33	2	0	0	5653	8	969	24373	0.1	6494.4	6447.735	2439.352	1218.87	4138.941	0		
35	34	2	0	0	6634	9	981	25354	0.1	6555.183	6541.935	2511.838	1178.36	4099.264	107.2		
36	35	2	0	0	7607	10	973	26327	0.1	6781.6	6634.67	2545.607	1232.236	4313.346	107.2		
37	36	2	0	0	8590	11	983	27310	0.1	6673.6	6728.846	2524.007	1207.971	4200.899	107.2		
38	37	3	0	0	0	0	961	28271	0.1	6479.2	9049.629	2490.284	1201.442	4094.221	107.2		
39	38	3	0	0	983	1	983	29254	0.1	6767.2	9149.586	2516.284	1163.369	4302.293	107.2		
40	39	3	0	0	1963	2	980	30234	0.1	6659.2	9244.377	2495.866	1129.924	4221.691	107.2		
41	40	3	0	0	2913	3	950	31184	0.1	6450.4	9334.909	2472.39	1116.293	4109.845	107.2		
42	41	3	0	0	3892	4	979	32163	0.1	6810.4	9428.529	2564.632	1174.448	4308.237	107.2		
43	42	3	0	0	4878	5	986	33149	0.1	6563.183	9523.786	2410.929	1206.668	4194.391	-51.70206		
44	43	3	0	0	5855	6	977	34126	0.1	6645.593	9617.687	2552.864	1151.308	4159.76	107.2		
45	44	3	0	0	6199	7	344	34470	0.1	2094.398	9651.111	926.9983	572.5004	1779.067	-7.997999		
46	45	3	0	0	7154	8	955	35425	0.1	6666.4	9742.685	2560.893	1187.792	4229.201	107.2		

Figure 22: Agent parameter data.

4.3 Development

4.3.1 Optimize Race Line

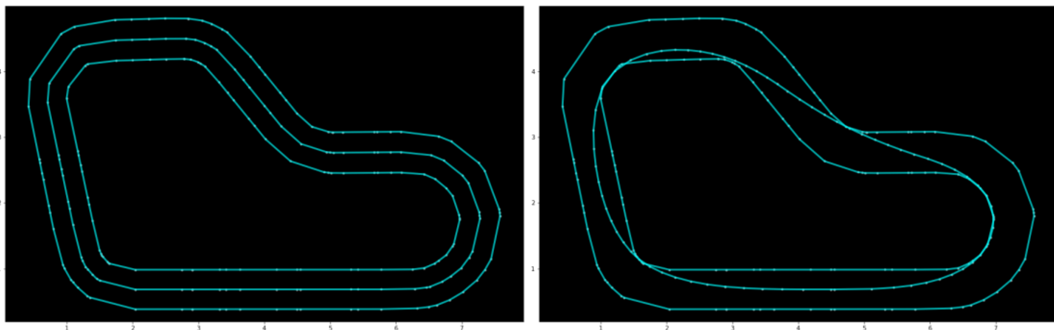
To optimize the reward function to enable the agent to complete the race faster. Remi Coulom introduced the algorithm in his PhD thesis [13]. The purpose is to find the optimal race line path for a certain track and to persuade the agent to follow it. Three sets of coordinates are used to establish a race track: the outside and inner borders, as well as the mid-point. A Python algorithm was developed Using the algorithm mentioned in to generate the track with the

"optimal" path for every track defined using the three coordinate sets [14]. Figure 23 illustrates this technique in action.

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.

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 Track (b) Optimal Race Line

Figure 23: Rebuild the race line [14]

The model was trained using the [14] reward function, which applied the K1999 Race-Line Optimization algorithm. Figure 24 illustrates the reward accumulated throughout training.

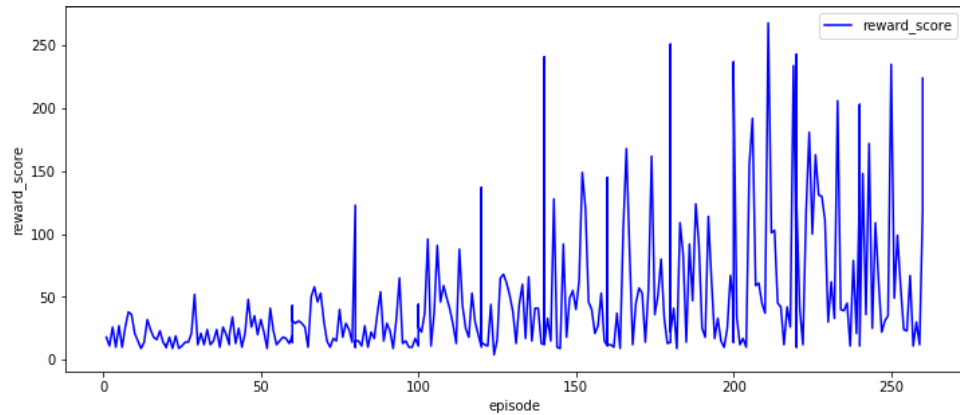


Figure 24: Reward apply K1999 Algorithm

While the rewards in figure 24 appear to be satisfactory, they are not consistent with the work's objective. Training with this reward function significantly increases the probability of overfitting to a track. By implementing the optimal race-line track, the model can operate optimally only on the track on which it was trained. Because the thesis's final aim is to operate the DeepRacer car on an actual track, the model should be capable of navigating an unknown track. The track used to evaluate the automobile may differ from the tracks used to train the model. As a result, it is required to establish a "universal" model.

4.3.2 Universal Model

While training DeepRacer models for a single track produces a reasonable result, it does not produce a robust model that can be utilized across all DeepRacer tracks. Based on [15], a more extended reward function was implemented to train a model on several tracks by cloning the model after each training session. The reward function utilized for this purpose, as developed from [15], is presented below. Figure 25 illustrates the hyperparameters used to train the universal model.

```
1 def reward_function(params):
2     reward = 0.001
3     if params["all_wheels_on_track"]:
4         reward += 1
5     if abs(params["steering_angle"]) < 5:
6         reward += 1
7     reward += ( params["speed"] / 8 )
8 
9     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 25: Hyperparameter apply in the universal model.

Firstly, the model was trained for an hour on the oval track. Figure 26 shows the reward graph, whereas Figure 27 shows the Oval track. The training time was limited to one hour due to the model being cloned and trained on a different track. By cloning, the current network weights would remain constant, allowing the "knowledge" gained during this track's training to be carried over to the next. The reward graph also reveals that the model was unable to complete the

course during training. Additionally, the graph represents the best model depending on the track's completion.

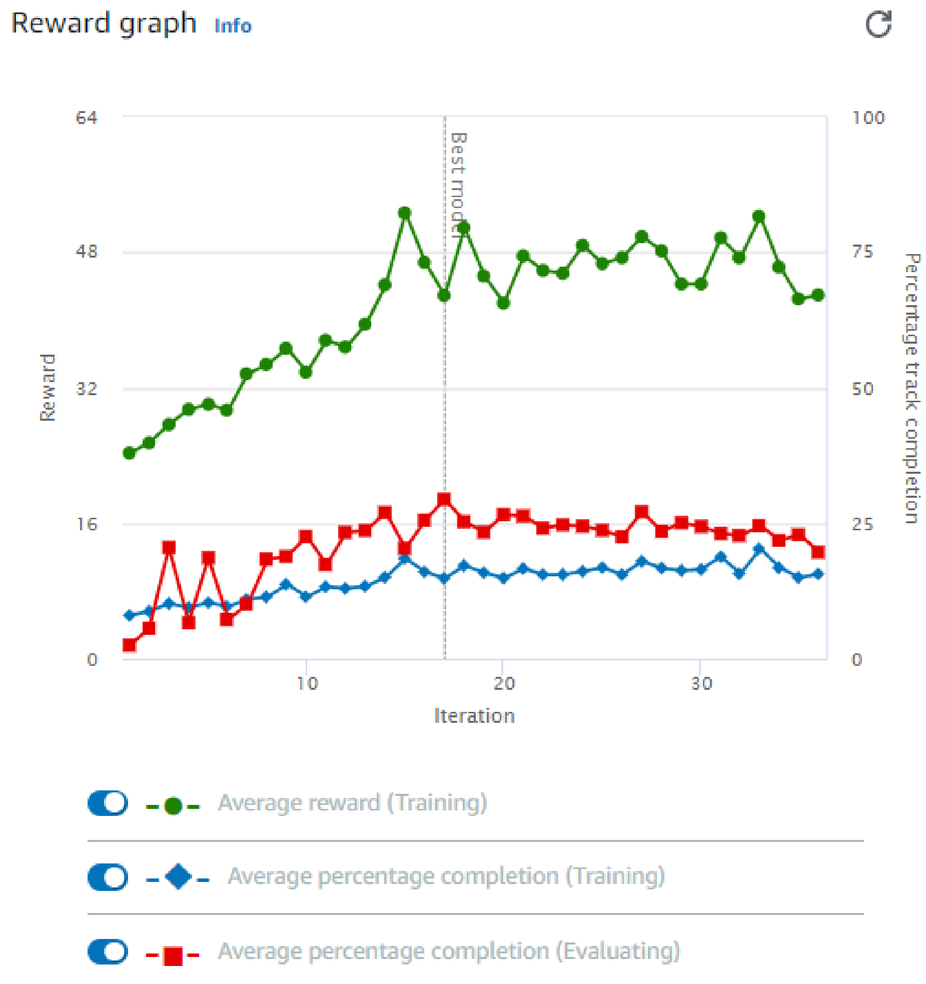


Figure 26: Reward Graph on the Oval Track.

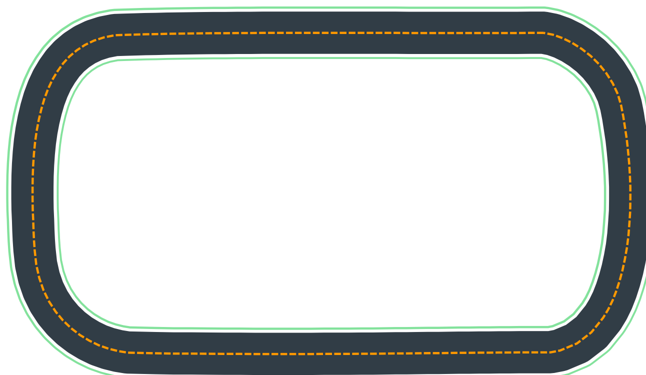


Figure 27: Oval Track.

To train on the Reinvent-base track, the model was duplicated. Since this is a duplicated model, the reward function and hyperparameters were identical. The reward graph and track layout are illustrated in Figures 18 and 19. Both reward and track completion rates improved significantly during this training session.

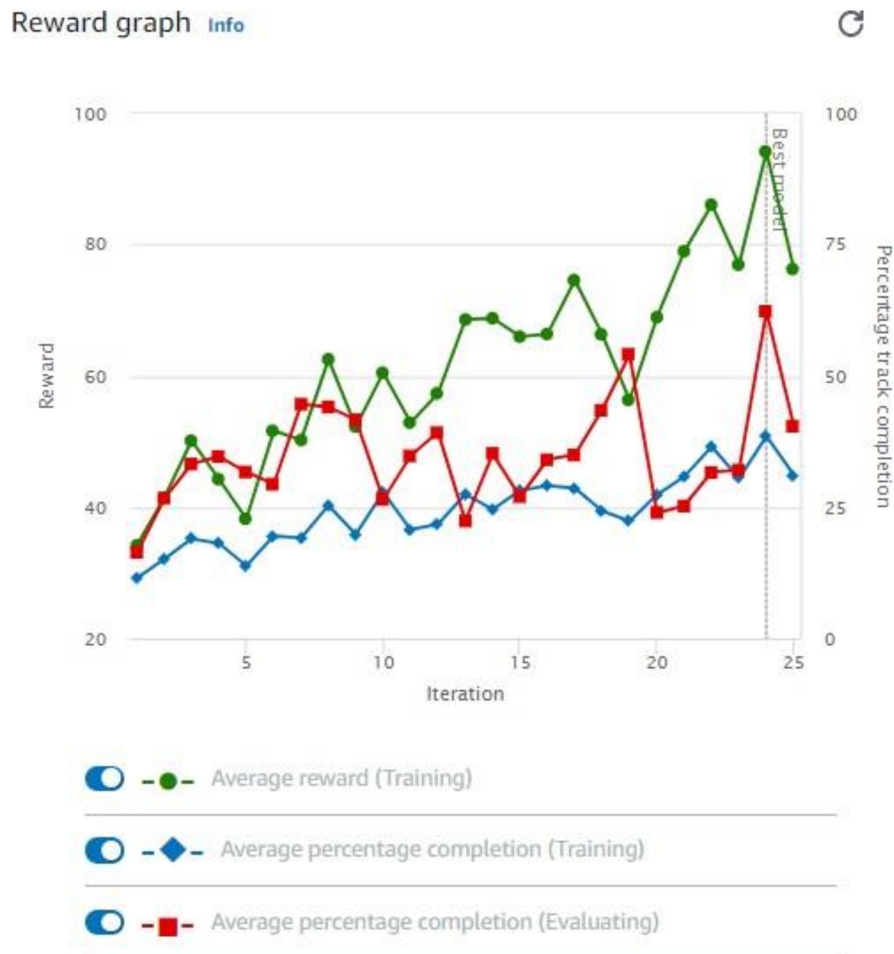


Figure 28: Reward Graph on the Reinvent-base Track

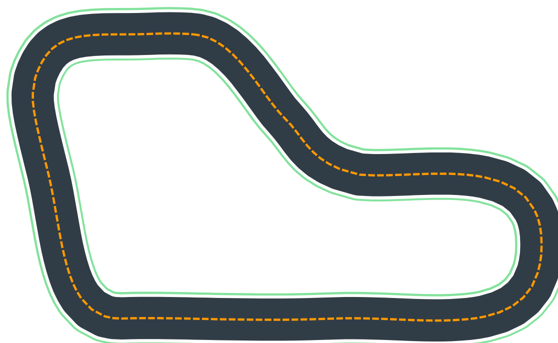


Figure 29: Reinvent-base Track

This model was duplicated to allow for more training on a particularly tough track — the Bowtie. Figure 20 illustrates the track. Due to the bow shape, this track proved to be difficult. The agent did not comprehend the middle curve since it could "see" the track on the other side and desired to take a "shortcut" whenever it met those bends. As a result, the performance rate decreased throughout this training session, as illustrated in figure 21 reward graph. Similarly, the award may have been affected for the same reason. Due to the model's success on the reinvent track, it was able to achieve even greater outcomes.

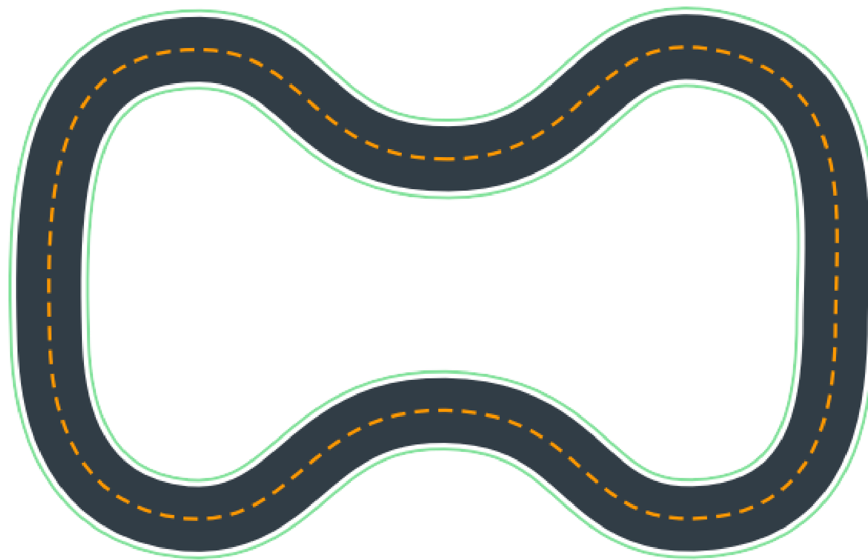


Figure 30: Bowtie Track.

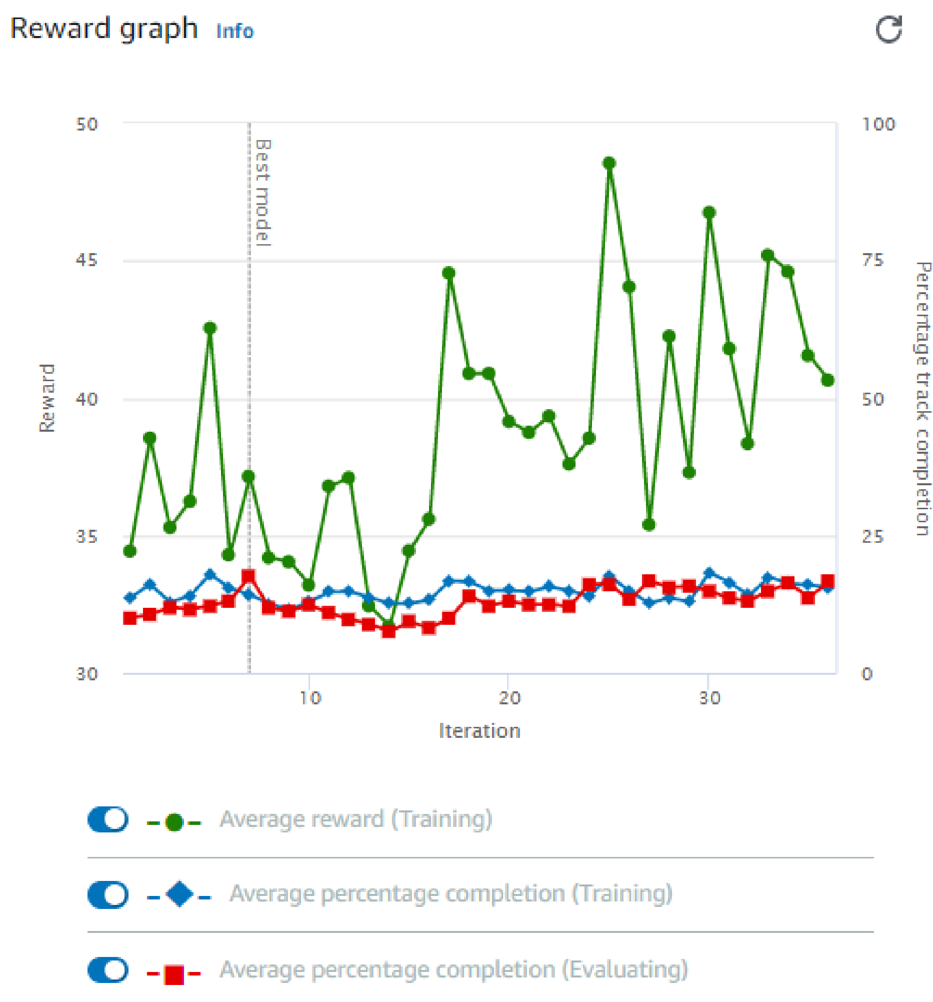


Figure 31: Reward Graph on the Bowtie Track.

5 Conclusion

AWS DeepRacer is an excellent platform for learning about RL and experiencing it in action. It includes simple-to-follow instructions for beginners as well as the ability for an "advanced" user to access the backdoor. The models were trained to run the real car autonomously using a variety of reward functions and state-of-the-art optimization algorithms. However, the development of the track required extreme care and precision, as the training approach could only give a suitable outcome if the track approximated the training track. AWS includes step-by-step instructions for building tracks. With the right track building implements, the car would be capable of successfully

navigating the race track on which it was trained, exactly as it did during the simulated evaluation.

Due to the limited scope, resources, and time available for this research, there was insufficient time to examine everything. Customization of the reward function received less attention. A possible area of investigation is the optimization of the reward function. If the model already had a very good reward function, it would converge more quickly while simultaneously learning a more optimal policy. It is necessary to reward the model for appropriate behavior. After finishing model training and evaluation in simulation, the next stage would be to evaluate DeepRacer in a physical setting.

References

1 Amazon. 2022. AWS DeepRacer: Developer Guide. Amazon Web services

2 Understanding autonomous vehicles: A systematic literature review on capability, impact, planning and policy. Online. JTLU. <https://www.jtlu.org/index.php/jtlu/article/view/1405>. Accessed 6 April 2022

3 Designing autonomous robots | IEEE Journals & Magazine | IEEE Xplore. Online. IEEEXplore. <https://ieeexplore.ieee.org/document/4799448>. Accessed 6 April 2022

4 CSC Infographic Big Data. Online. CSC. <http://assets1.csc.com/insights/downloads/CSC_Infographic_Big_Data.pdf>. Accessed 6 April 2022

5 Lonza, Andrea. 2019. Reinforcement Learning Algorithms with Python. Packt.

6 “Openai documentation: Getting started with gym,”. Online. Gym <https://gym.openai.com/docs/>. Accessed 6 April 2022

7 Deep Reinforcement Learning Course. Online. SIMONINI Thomas <https://simoninithomas.github.io/deep-rl-course/>. Accessed 6 April 2022

8 What Is AWS DeepRacer? - AWS DeepRacer. Online. Amazon Web Services <https://docs.aws.amazon.com/deepracer/latest/developerguide/>. Accessed 6 April 2022

9 Overfitting and Underfitting in Machine Learning – Javatpoint. Online. Javapoint: <https://www.javatpoint.com/overfitting-and-underfitting-in-machine-learning/> Accessed 6 April 2022

10 Ray Goh. October 13, 2020. Using log analysis to drive experiments and win the AWS DeepRacer F1 ProAm Race. AWS Amazon

11 OpenAI. 2018. Proximal Policy Optimization. OpenAI. Online. OpenAI: <https://spinningup.openai.com/en/latest/algorithms/ppo.html/> Accessed 6 April 2022

12 Deepracer Car. Online. Xingtong Li. <https://cse.buffalo.edu/~avereshc/rl_spring20/Xingtong_Li.pdf> Accessed 6 April 2022

13 R. Coulom, “Reinforcement learning using neural networks, with applications to motor control,” Ph.D. dissertation, Institut National Polytechnique de Grenoble- INPG, 2002

14 Discovering Race Lines in DeepRacer Track Geometries. Online. C.D Thompson <https://github.com/cdthompson/deepracer-k1999-race-lines/> Accessed 6 April 2022

15 Aws deepracer experimentation. Online. S. Pletcher. <https://github.com/scottpletcher/deepracer/> Accessed 6 April 2022