Oskar Rönnberg

Reinforcement learning with AWS DeepRacer

Metropolia University of Applied Sciences

Bachelor of Engineering

Programme in Information and Communication Technology

Bachelor’s Thesis

1 June 2021

Abstract

Author:	Oskar Rönnberg
Title:	Reinforced learning with AWS DeepRacer
Number of Pages:	60 pages
Date:	1 June 2021

Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Smart Systems
Instructors:	Erik Pätynen, Senior Lecturer

The main purpose of this project was to take a look at reinforcement learning with AWS DeepRacer by providing good policies to use when training reinforcement learning models. Another goal was to analyse model performance and how to tune the performance of a model to achieve a better model.

AWS DeepRacer is a 1/18th scale autonomous vehicle which is taught to drive by itself with reinforcement learning along various tracks. Currently DeepRacers are trained for three types of objectives: time trials, object avoidance and head-to-head racing. In this project DeepRacer was trained for time trials and object avoidance. DeepRacer was trained to drive within a simulation created by AWS RoboMaker, and its neural network was updated within AWS SageMaker. DeepRacer could either be driven in simulation or on a physical track.

In this project, estimates were created for a required amount of training time for a model. In addition, estimates for the initial training time for a model were created. Moreover, the thesis discusses how significantly agent parameters affect model performance; which approaches work the best in reward functions; how changing hyperparameters affects the model and its performance; how to evaluate model performance from log files; and how to improve the quality of training and model performance by doing log analysis.

The results can be used as general guidelines for model training and improvement in reinforcement learning with AWS DeepRacer. Following the policies recommended in the thesis, better and more stable models can be achieved.

Keywords: Reinforcement learning, neural networks, autonomous driving

Tiivistelmä

Tekijä:	Oskar Rönnberg
Otsikko:	Reinforced learning with AWS DeepRacer
Sivumäärä:	60 sivua
Aika:	1.6.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Smart Systems
Ohjaajat:	Lehtori Erik Pätynen

Opinnäytetyön päätarkoitus oli tarkastella AWS DeepRaceriä vahvistusoppimisen kautta. Tavoitteena oli tarjota hyviä käytäntöjä, joita käytetään, kun harjoitetaan vahvistusoppimisen malleja. Kun analysoidaan mallin suoriutumista ja kun muutetaan mallia, jotta saavutettaisiin parempi mallin suoritus.

AWS DeepRacer on 1/18-suhteessa oleva autonominen ajoneuvo, jota opetetaan ajamaan itsenäisesti eri radoilla vahvistetulla oppimisella. Tällä hetkellä DeepRaceriä harjoitetaan kolmeen eri tehtävään: aikakisoihin, esineiden väistelyyn ja osallistuja-vs-osallistuja kisaan. DeepRacer treenataan AWS RoboMaker -simulaatiossa ja mallin neuroverkko harjoitetaan AWS SageMakerissa. DeepRaceriä voi ajaa joko simulaatiossa tai fyysisellä radalla.

Tämän työn tuloksia ovat: arviot riittävästä harjoitusajasta mallille tai ensimmäisen kerran harjoitussessiolle, agentin parametrien merkittävyys mallin suorituskykyyn. Myös parhaiden suoriutuneiden palkintofunktioiden esittely sekä se, miten hyperparametrien muutto vaikuttaa malliin. Lisäksi tutkitaan, kuinka arvioidaan mallin suoritusta lokeista sekä kuinka parannetaan mallin harjoituksen ja suorituksen laatua lokien analyysillä.

Tämän opinnäytetyön tuloksen tarjoavat yleisen ohjeistuksen mallien harjoittamiseen ja mallien parantamiseen vahvistusoppimisessa AWS DeepRacerilla. Seuraamalla tuloksissa suositeltuja käytäntöjä pitäisi saavuttaa parempi vakaampi malli.

Avainsanat: Vahvistettu oppiminen, neuroverkot, autonominen ajaminen

Contents

List of Abbreviations

Redis:	Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. Redis provides data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes, and streams.
Neural network:	A neural network is a collection of connected nodes that are used to build information models based on a biological neural system. Each node is called an artificial neuron, and it mimics biological neurons in that it receives input and produces an output depending on the node weights.
Tuple:	A tuple is used to store multiple items in a single variable in the Python programming language. Tuple is one of the four built-in data types in Python used for storing data.

1 Introduction

Machine learning has been growing in popularity over the recent years. In the past, machine learning was heavily restricted by the demand in computational power and amount of data available. Development of better algorithms has also helped the field of machine learning to progress. Since machine learning started to flourish, it has quickly become the most popular and successful subfield of AI. Machine learning has multiple applications such as voice and image recognition, and as time passes, it is being applied to further applications.

Compared to traditional programming, machine learning is all about the computer finding its own answers to given problems based on the data it is provided with. Machine learning differs from traditional programming in a sense that instead of giving the computer data and rules from which the computer produces results, the computer is given data and results from which the computer will determine on its own the rules and how it will achieve the desired results. Machine learning is divided into various fields such as supervised learning, unsupervised learning, self-supervised learning and reinforcement learning, which will looked into in this thesis.

This thesis documents an individual project carried out for Metropolia University of Applied Sciences. This thesis will examine reinforcement learning with AWS DeepRacer. The aim of the thesis is to examine three racing types of AWS DeepRacer: time trial, object avoidance and head-to-head racing. The focus of the thesis will be on time trial and object avoidance. The head-to-head racing format will be examined briefly. Various approaches to make a model perform better will be examined. In addition, suggestions are provided for further studies since this thesis can only take a general look at AWS DeepRacer. In general, the goal of this thesis is to give an overview of AWS DeepRacer and its applications and also provide good policies to use in reinforcement learning when building models with AWS DeepRacer.

2 Theoretical background

2.1 Reinforcement learning

Humans have learned through trial and error. Our learning process is based on our own reward mechanisms that provide a certain reward for our actions. The goal of our learning process is through repeated, incentivized repetitions to trigger even more positive responses and disincentivize repetitions of actions triggering negative responses. Through this, we learn how to interact with the world around us and learn how to solve even more complex problems. [1.]

Reinforcement learning is inspired by how humans learn, it is built around the idea of trial and error from interactions with an environment. [1.]

What reinforcement learning really is

Reinforcement learning is one of four types of machine learning: supervised learning, unsupervised learning, self-supervised learning and reinforced learning [2].

Reinforcement learning deals with sequential decision-making trying to reach its desired goals [1]. For example, its desired goal could be learning how to drive itself along a track. In a nutshell reinforcement learning is all about a decision-maker called an agent receiving information concerning its environment and learning to choose actions which in turn will give the agent the highest reward [2].

First, the agent takes an action in the provided environment, virtual or physical. After its action the agent will receive a reward based on the reward function. The reward can either be positive or negative. After that, the agent will reach a new state in its environment and then change its policies(strategy) to gain even more rewards from its actions. The agent’s objective is to maximize the agent’s total reward. The reward amount of an action is the desirability of the action.

This cycle will repeat itself as long as the training persists, as illustrated in Figure 1. [3.]

The agent's goal is to learn an optimal policy in its given environment. Learning is an iterative process of trial and error. At the start, the agent takes random actions to arrive at a new state. Then through iterative process the agent proceeds from the new state to the next state. Through iterations, the agent discovers which actions will lead to maximum long-term rewards. The agent's journey from the initial state to the terminal state is called an episode. [3.]

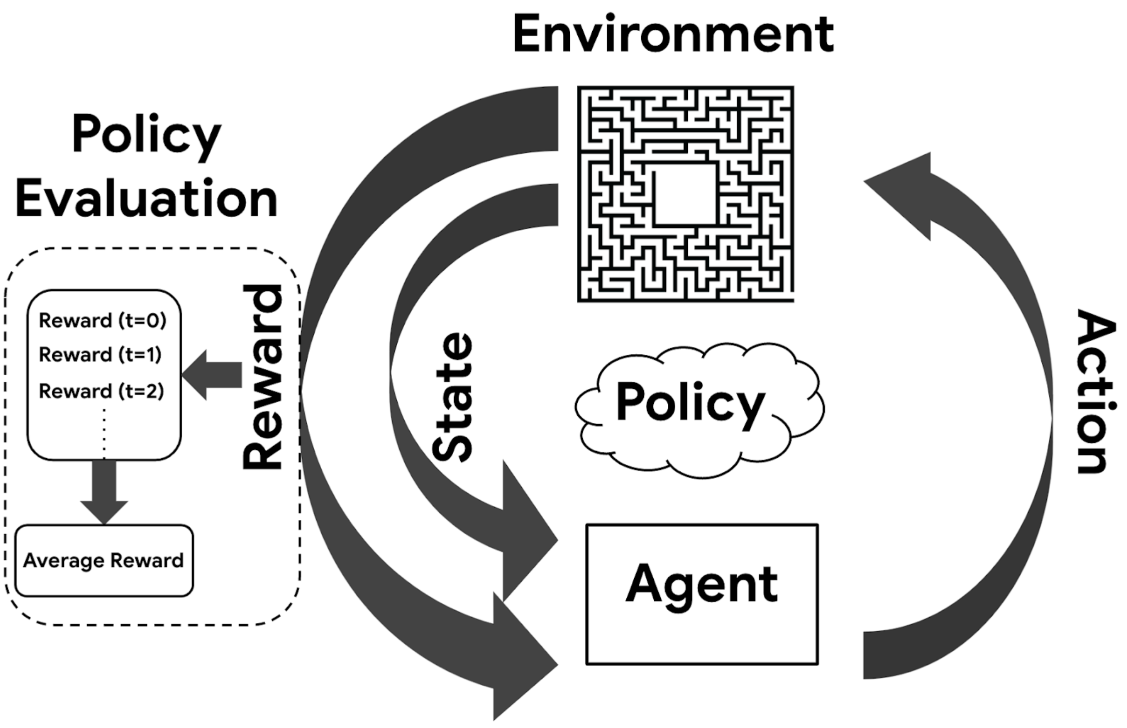


Figure 1. Reinforcement learning in a nutshell. Copied from Google ai blog [4].

AWS DeepRacer

In DeepRacer, the agent represents a neural network the function of which approximates the agent's policy. The environment state is the equivalent of the image from the vehicle's front camera (see Figure 2) and the agent's actions are defined by speed and steering angles.

If the agent stays on-track to finish the race, it will receive positive rewards, if it fails, it will receive negative rewards. An episode starts with the agent being placed somewhere along the track and finishing when the agent either completes a lap around the track, goes off-track or crashes into an object or another vehicle. [3.]

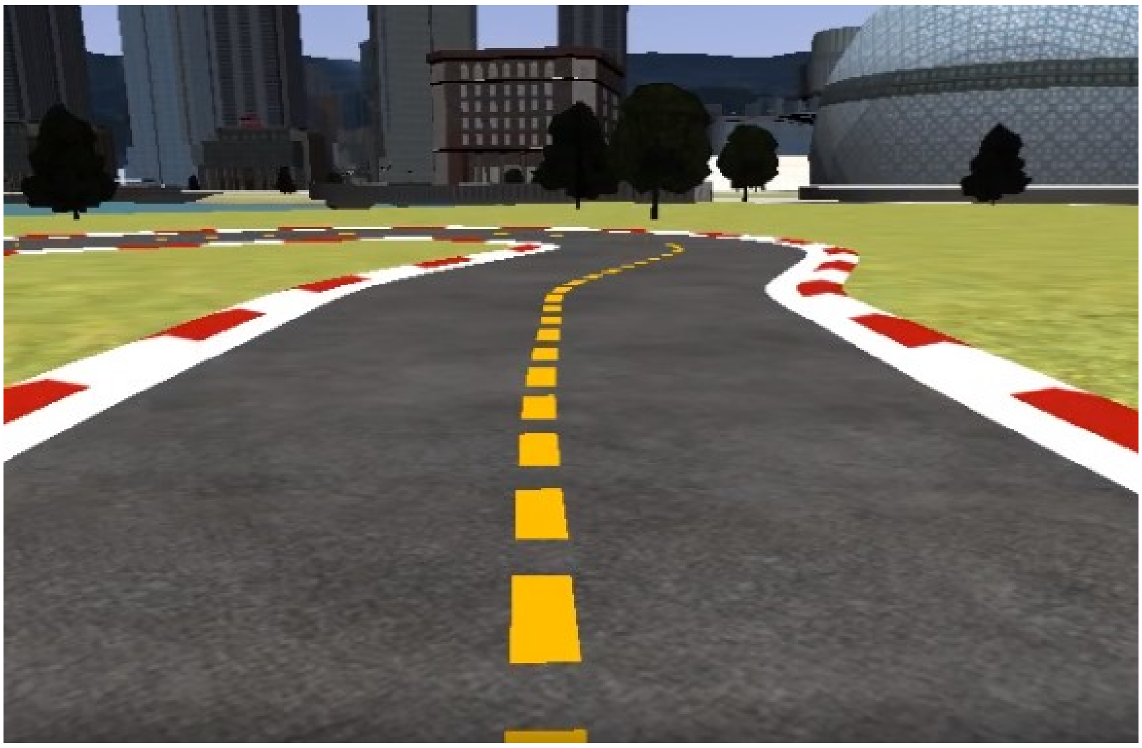


Figure 2. DeepRacer state. Copied from DeepRacer blog [5].

A note from AWS DeepRacer Developer Guide

> Strictly speaking, the environment state refers to everything to the problem. For example, the vehicle's position on the tracks as well as the shape of the tracks. The image fed, though the camera mounted the vehicle's front does not capture the entire environment state. Hence, the environment is deemed partially observed and the input to the agent is referred to as observation rather state. For simplicity, we use state and observation interchangeably throughout this documentation. [3.]

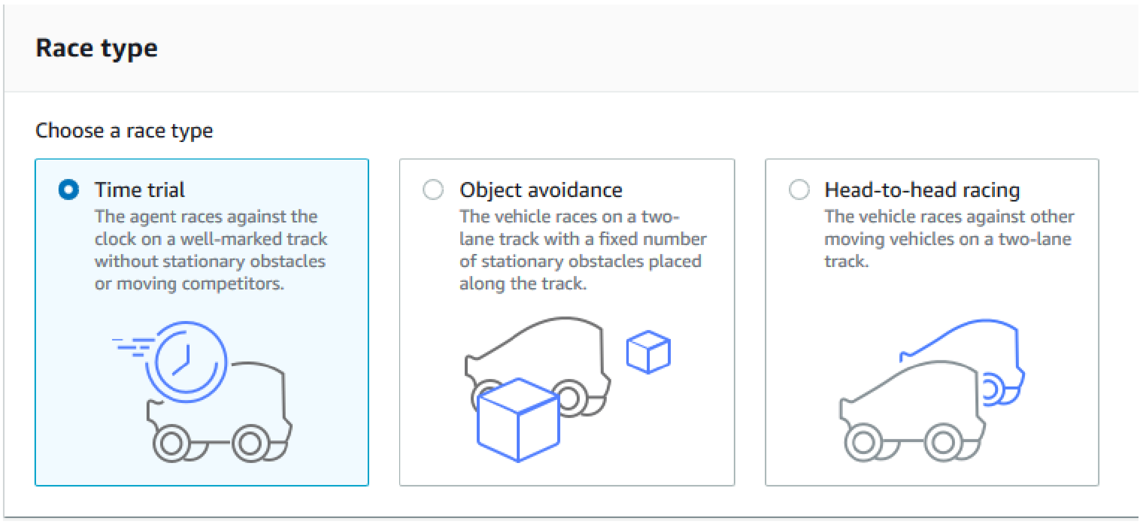
2.2 Model

Model is an essential component of DeepRacer. The model is all the various parameters and settings combined to form an environment in which the agent acts. It has three separate forms: the agent’s state, the agent’s action and the agent’s reward for taking action. Policy is the strategy with which the agent makes its decisions, policy’s input being environment state and output as the action to take. The policy is often represented by a deep neural network and in DeepRacer it is referred to as the reinforcement learning model. Each training session generates one model. Even if the training is stopped before completion, a model is formed. A model is unchangeable, which means that it cannot be modified after creation, but a model can be cloned and then trained with different parameters. [3.]

2.3 Race type

Currently there are three DeepRacer racing categories: time trial, object avoidance and head-to-head racing as shown in Figure 3.

In Time trials, the agent tries to get through the track as fast as possible.
In Object avoidance, the agent tries to get through the track while avoiding objects placed at either static or random locations.
In Head-to-head racing, the agent tries to get through the track while racing against bots with static speed or other agents.


Race type

Choose a race type

(●) Time trial
The agent races against the clock on a well-marked track without stationary obstacles or moving competitors.

( ) Object avoidance
The vehicle races on a two-lane track with a fixed number of stationary obstacles placed along the track.

( ) Head-to-head racing
The vehicle races against other moving vehicles on a two-lane track.

Figure 3. AWS DeepRacer console race type selection. Copied from AWS client console [13].

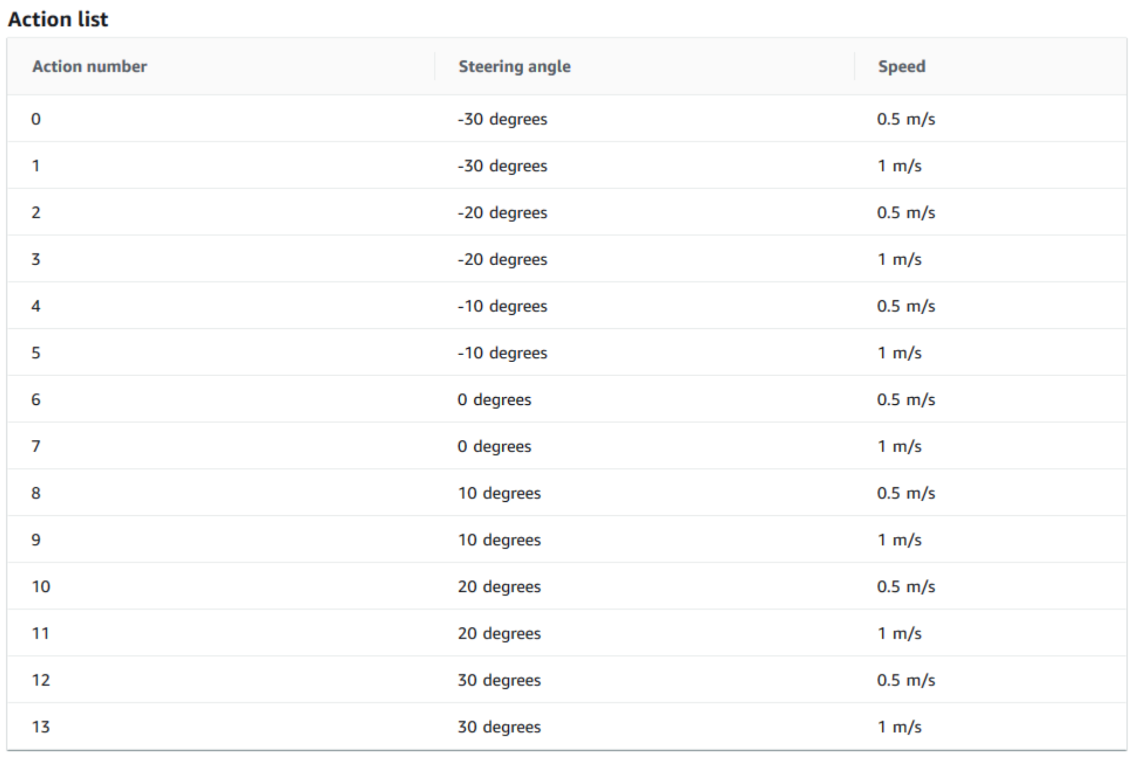
2.4 Action space

Reinforcement learning requires a certain set of valid actions or choices from which an agent can choose, as the agent interacts with its environment. This is called action space. In AWS DeepRacer, action space is either discrete or continuous. [3.]

Discrete action space

Action space that is discrete represents all possible actions an agent can take in each of its states in a finite set. For every different scenario in the track, AWS DeepRacer's neural network responds with a certain speed and turning angle for the agent based on input from its camera(s) and optionally its LiDAR sensor. The choices are in predetermined action numbers, in which the car has a certain steering angle and a throttle combination. [3.]

AWS DeepRacer can turn left, right, accelerate, decelerate and go straight forward. The actions mentioned are combined to form a list of actions for the agent to choose from. For example, an agent could have the action space of 14 actions. The action numbers are, 0 (-30 degrees and 0.5 m/s), 5 (-10 degrees and 1 m/s), 10 (20 degrees and 0.5 m/s) and so forth as shown in Figure 4. [3.]



Action list

Action number	Steering angle	Speed
0	-30 degrees	0.5 m/s
1	-30 degrees	1 m/s
2	-20 degrees	0.5 m/s
3	-20 degrees	1 m/s
4	-10 degrees	0.5 m/s
5	-10 degrees	1 m/s
6	0 degrees	0.5 m/s
7	0 degrees	1 m/s
8	10 degrees	0.5 m/s
9	10 degrees	1 m/s
10	20 degrees	0.5 m/s
11	20 degrees	1 m/s
12	30 degrees	0.5 m/s
13	30 degrees	1 m/s

Figure 4. Discrete action space from an object avoidance agent. Copied from AWS client console [13].

Continuous action space

In a continuous action space, the agent has the liberty to choose from an infinite range of values for each state by itself. Just as in a discrete action space, the car makes its decision based on the input it gets from its camera(s) and possibly the LiDAR sensor. For example, in continuous action space, it is possible to set the range of values to be between -30 to 30 degrees for steering and from 0.5 m/s to 1 m/s with the throttle. What makes continuous action space stand apart from discrete action space is the fact the agent chooses freely values from the range it is given. In addition, it has no predetermined action list as shown in Figure 5. [3.]

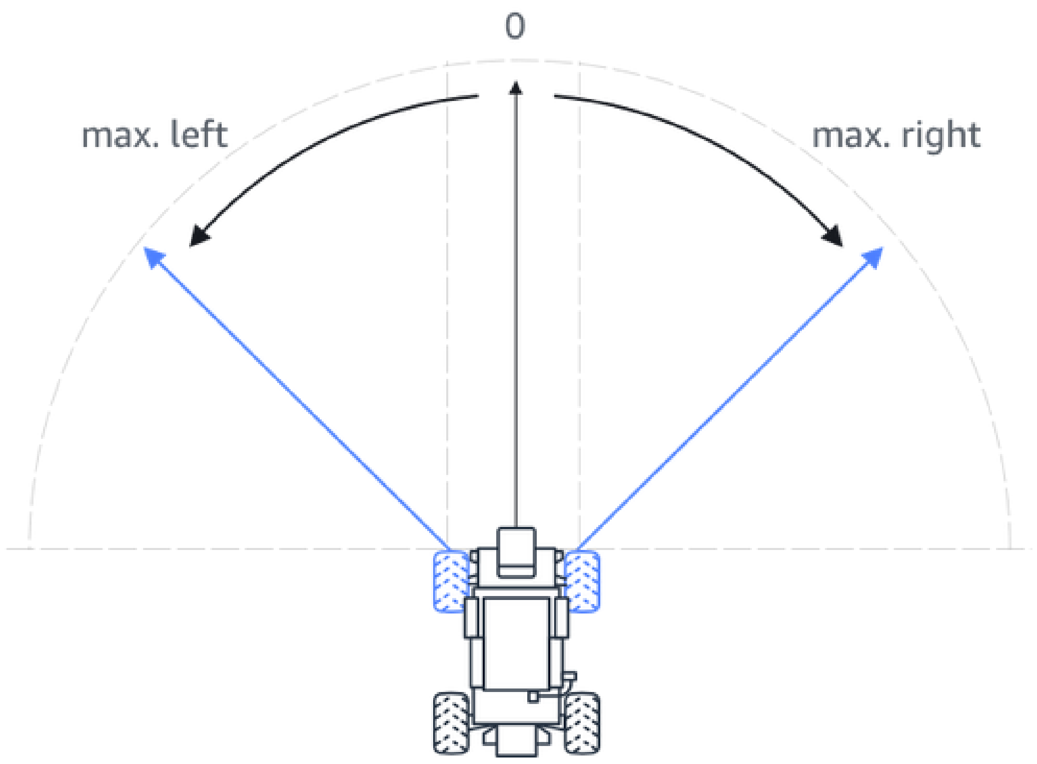


Figure 5. Agent with continuous action space. Copied from AWS client console [13].

Discrete or continuous?

Should one use discrete or continuous action space? Both have their own advantages compared to each other.

The discrete action space gives better comprehension of the agent’s choices since they are limited to a list of actions. It is possible to tailor make action space based on the environment you are working with. For example, if the track the model is trained on only has turns to right, is it possible to make an action list only consist of various turns to right and driving forwards. Using the discrete action space also require less training than the continuous action space since there are fewer actions to choose from and to learn. [3.]

The continuous action space would in turn, give you more options when writing reward functions since you can incentivize different behaviours such as steering or throttle depending on which specific point the agent is at along the track. Having a range of action instead of an action list also gives smoother updates to speed and turning angles, which in turn may benefit the model, especially when the model is used in a physical DeepRacer car. [3.]

2.5 Reward function

Policy

The agent's policy defines how it will select an action in a given state. The policy will choose actions which result in the best possible cumulative reward. It will not choose actions for an immediate higher reward. In a nutshell, the policy always tries to achieve its ultimate objective first, meaning that the policy could choose actions which in its current state are not optimal actions. [1.]

Value function

Value function represents the quality of state in the long-term. It is the cumulative reward that the agent expects to be given in the future state from the current state. The reward measures the agent's immediate performance in the state. The value function measures the agent's performance in the long run. This indicates that a high reward does not necessarily correlate to a high reward function or otherwise. [1.] For example, DeepRacer could dodge an object or another DeepRacer on track and be given a high reward for that action but drive out in the next turn.

Reward

The agent takes an action in each step and the value function tells the agent how good that action was. This is called a reward. As already mentioned, the agent's goal is to maximize the cumulative rewards it receives. [1.]

Rewards can both appear frequently or sparsely. Frequently appearing rewards are called dense rewards and sparsely appearing rewards are called sparse rewards. [1.] For example, DeepRacer staying on-track is a dense reward since the car will either stay on-track for the full track length or drive out of the track. DeepRacer dodging an object is a sparse reward since even with six objects on track they are still within a relatively short distance of the track.

Reward function

The agent learns the value function by exploring the environment. The value function shows the agent which actions are good and which actions are bad. The value function uses the reward function, which is written to score actions. The reward function is written in Python in DeepRacer. Listing 1 is an example of what a default DeepRacer (following the center line function) does.

```
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
```

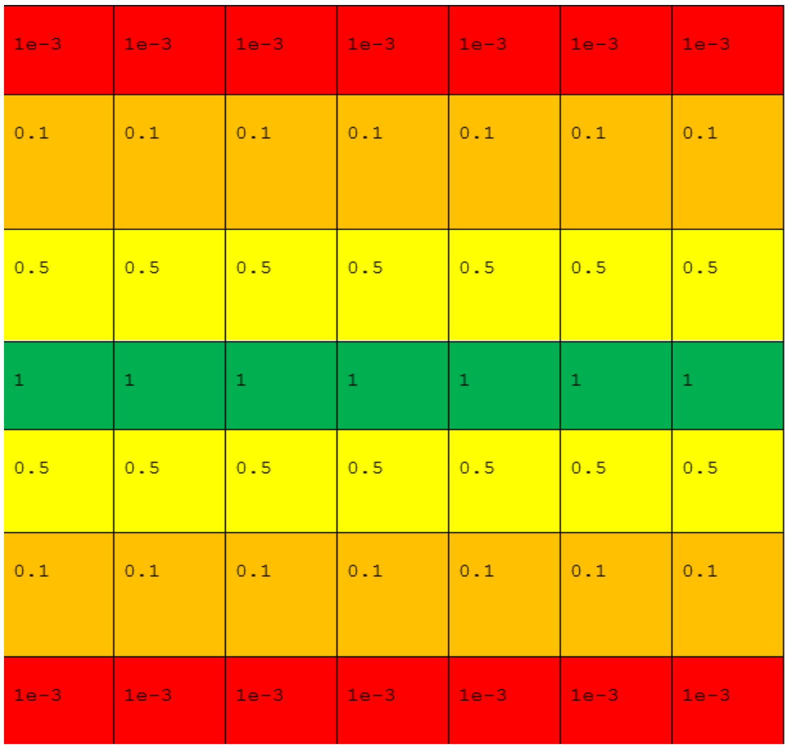
Listing 1. Simple reward function. Copied from AWS client [6].

In the function above the agent is rewarded when it stays inside the track and the reward gets better the closer to the center line the agent is. The agent’s reward is minimal for driving off-course.

At the start, the agent takes random actions to explore the environment but as time passes the agent learns which actions will keep it within the center line and its maximal reward. If the agent keeps taking continuous random actions, it will take longer time to get around the track for a full lap. As the model's policy begins to learn good actions, it will do fewer random actions. However, if the agent only uses already learned actions it will not explore the environment anymore and thus will not learn new actions. This trade-off mentioned above is often called exploration vs exploitation problem in reinforcement learning. [3.]

It is possible to get around the track with surprisingly good results with the default (following the center line function) showed listing 1 [7]. The function seen in Listing 1 has its downsides too, since following the center line is not optimal pathing along the track most of the time. This default function reward function was shown because building a good model does not necessarily require you to have an advanced and complicated reward function, although having a better reward function is always good and rewarding the agent for correct actions is very important. [8.]

Figure 6 shows what the reward function mentioned above will look like in grid form, each column representing a possible state and its reward.



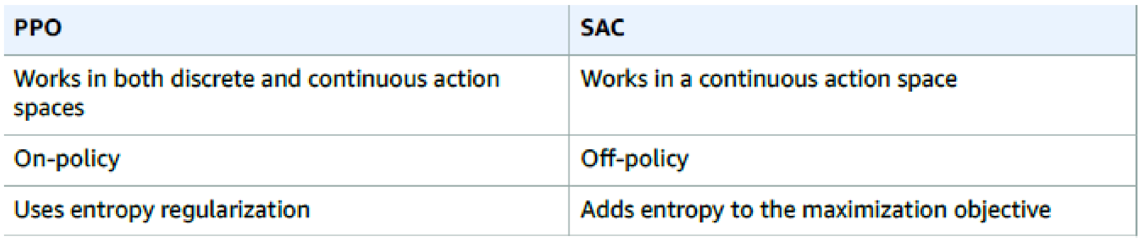
1e-3	1e-3	1e-3	1e-3	1e-3	1e-3	1e-3
0.1	0.1	0.1	0.1	0.1	0.1	0.1
0.5	0.5	0.5	0.5	0.5	0.5	0.5
1	1	1	1	1	1	1
0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.1	0.1	0.1	0.1	0.1	0.1	0.1
1e-3	1e-3	1e-3	1e-3	1e-3	1e-3	1e-3

Figure 6. Reward function grid.

2.6 Training algorithms

AWS DeepRacer has two training algorithms which are Proximal Policy Optimization (PPO) and Soft Actor Critic (SAC). Both algorithms learn policies and value function at the same time, but their strategies vary in three notable ways as shown below in Table 1. [3.]

Table 1. Differences between the two algorithms. Copied from AWS Developer Guide [3].



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 can be seen in the table, both algorithms have their own requirements and approaches concerning how to learn. In this context, it is not necessary to go through the mathematical side of the algorithms; instead, the focus should be on how they are used in DeepRacer and what the advantages are in either of the algorithms. The differences between the algorithms include what has already been described: action space requirement, whether they are on-policy or off-policy and how the algorithms manage entropy.

Entropy

Entropy is the measure of uncertainty in the policy decision-making, it can be thought of as the measure of confidence for the policy to choose its actions. Low entropy means that the policy is very confident in its decision-making; high entropy leads to the policy finding trouble at choosing which action to take. [3.] Generally, when training a model, you have high entropy at the start of the training, and then as the training progresses, the entropy should lower over time since the model it has explored its environment and learned its value function.

Exploration or exploitation?

Should you either explore or exploit action space? This is one of the biggest questions in reinforcement learning. To achieve a higher cumulative reward, an algorithm should exploit already learned information, but it should also explore for more information, so it can be used for finding an optimal policy. As the training progresses over multiple iterations within an environment, the policy learns and begins to make certain decisions for a given state. However, this can lead to problems if the policy does not explore the environment enough. Then the policy will only use already learned information and it will not explore for more information within the environment that could lead to an optimal policy. [3.]

PPO

As shown above in the table 1, PPO uses on-policy learning, meaning that PPO learns its value function from the current policy exploring its environment. You could say that it learns from the same data that it generates. On-policy algorithms need more data for training most of the time, but they are in return more stable. Over time, the policy becomes less random since updates on weights encourage exploiting already found rewards. [1; 3; 9.]

The PPO algorithm uses entropy regularization which will prevent the agent from converging at local maxima. PPO encourages exploration by that. Entropy regularization makes PPO less efficient with data. [3.]

SAC

SAC uses off-policy learning. Off-policy learning consists of two policies: behaviour policy and target policy. The behaviour policy interacts with the environment and collects data for the target policy to improve itself. The target policy can learn from previous policies. [1; 3.] Off-policy algorithms are less stable most of the time, but they require less data for training. [3.]

SAC uses entropy maximization. It encourages wider exploration while also avoiding convergence at bad local maxima. SAC does this by preferring the agent to choose actions with higher entropy. SAC also has the unique advantage giving up on policies that choose inflexible behaviour which is one of the reasons why SAC seems to be more data efficient. [3; 10.]

2.7 Hyperparameters

One of the ways to improve model performance is to have more efficient training. To obtain a stable, robust model means that the training must give your agent evenly distributed sampling over its own action space. To achieve this, the model should have a decently balanced mixture of exploration and exploitation. A lot of variables affect this outcome such as: number of episodes between each training, batch size, learning rate and entropy. [3.] Tuning hyperparameters can prove to be valuable, but they will not turn a bad model into a good model, and you can ruin a good model with too much hyperparameter tuning. [8.] What these various hyperparameters are is explained below.

Data point

Data point or experience is a tuple of (s, a, r, s’): s for state, a for action, r for reward and s’ for new state [3].

Episode

Episode is a period in which the agent starts at its given starting point and either drives round the track to that same exact point or drives off-track. Episode lengths can and will vary from each other. An episode is a collection of data points. [3.]

Experience buffer

Experience buffer has a certain number of ordered data points which are collected over a fixed period of episodes of various lengths over training. In DeepRacer, it correlates to images taken by the vehicle’s camera and the photos taken serve as the source from which the input data is drawn. [3.]

Batch

Batch is a list of ordered data points. They represent a portion of simulation over a certain amount of time, used to update the weights of the policy network. The batch is a subset of the experience buffer. [3.]

Training data

Training data is a set of batches sampled in randomized order from the experience buffer. The training data is used for updating policy network weights. [3.]

Gradient descent batch size

Gradient descent batch size is the number of recent agent data points sampled at random from an experience buffer and they are used to update weights in the neural network. There is inherent correlation in the input data. The correlation is reduced by random sampling. Larger batch sizes lead to more stable and smoother updates to the weights, but the training time of the model probably is longer and slower. [3.]

Epoch

The number of epochs is how many times the training data is passed through in order to update the weights of the neural network during its gradient descent. The training data is a certain amount of random samples from the experience buffer. Using a larger amount of epochs will lead to more stable training which in turn will be slower. [3.]

Learning rate

The learning rate controls at which rate the model learns. It does this by regulating how much gradient descent (or ascent) changes the values of neural network weights. A larger learning rate leads to faster training since the weights are updated more frequently, but if the learning rate is too large, the model might not be able to converge at all. [3.]

Entropy

Entropy is the factor of uncertainty within the policy allocation. The model with higher entropy has more randomness in its actions than a model with low entropy. Higher entropy leads to better and more thorough exploration of action space. [3.]

Discount factor

Specifies how far ahead into the future states rewards the agent will estimate with reward function when considering taking an action. A larger value leads to more steps being taken into consideration when making a move, but the training will be slower because of it. [3.]

Loss type

Loss type is an objective function used to update the weights of the neural network. The goal of loss type is to achieve incremental changes in the agent’s policy, so it converts from random actions to policy influenced actions over time. The loss type in DeepRacer is either Huber loss or Mean squared error loss. When the weigh updates are small, both behave in the same way. Differences occur when the changes to the weights are larger. The mean squared error starts making larger increments compared to the Huber loss. [3.]

Episodes between each policy update

The size of the experience buffer orders how many data points there are within an experience buffer. Larger buffers lead to slower but more stable updates. Especially more complex solutions require more data points since there is more to learn. Less complicated problems require less. [3.]

2.8 Overfitting and underfitting

In reinforcement learning, it is very important to learn how to spot overfitting or underfitting. Overfitting and underfitting will both heavily influence how your model behaves and performs if you do not take actions in order to counter them or at least lessen their impact on the model’s performance.

Overfitting

When the model starts to overfit, it grows too accustomed to its training environment. At some point, the model starts to perfectly perform within its environment at the cost of its performance dropping massively on any other environment. The model grows incapable of not performing outside of its training environment. A simple way to recognize overfitting with a model is to see if the model can perform relatively well on other tracks. [7; 11.]

Underfitting

Before the model has explored all its action space and environment, it will underfit. The underfitting model is easily spotted when the agent cannot even finish one lap in within a batch or even in the entire training session. [7, 11]

2.9 Convergence

Convergence means that the model is performing at its best. It is the optimal spot the model should be trained for. After convergence, it is possible that the model will slowly start to overfit and its performance within any other environment except its training environment will start to fall. As training progresses, entropy should decrease according to time since the agent is exploring its action space. When rewards and average progress start to even out, entropy too should even out because entropy can be used to spot convergence. When the rewards, average progression and entropy of the model start to even out, the model has converged (see Figure 7). [12.]

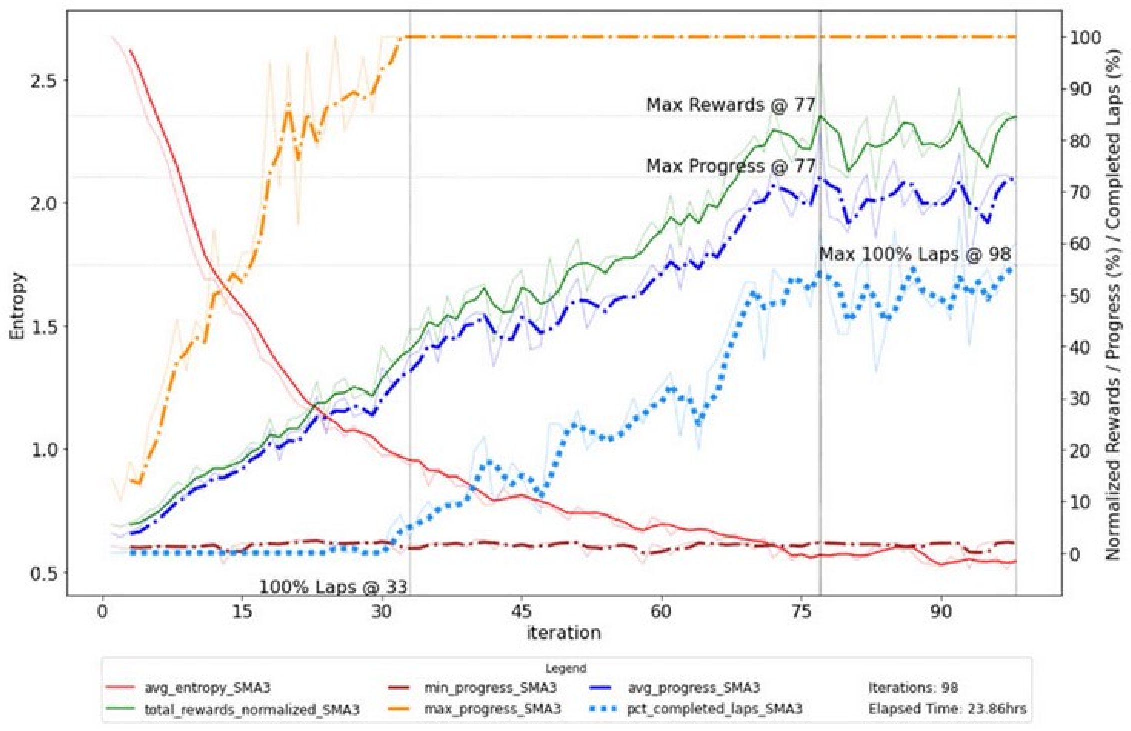


Figure 7. The convergence of a good model. It is important to notice the three separate peaks within the end of the training session. Copied from blog post [12].

2.10 Model stability and universality

In DeepRacer, you want your models to be as stable as they can. Stability leads to the model's performing better overall without having any significant drops in performance. Generally, if you do not have just one goal in mind, your models should be as universal as possible. If your models are universal, you can use them on multiple tracks and you can clone them for further training for other purposes or do something else with them. Models that are not universal have very limited usage since they probably have already overfitted to a particular training track. [7; 8.]

Stability

In generally, you want your models to be as stable as they can. The stability of a model is the key for thorough solid performance. The behaviour of a stable model is also at least relatively predictable while unstable models are very hard to predict. [7; 8; 12.]

Universality

For a model to be universal, it needs to be able to perform in multiple different environments even if it is only being trained in one environment or sometimes a few. In the universal model, you want the model to learn general rules and principles behind actions which the model can apply on various tracks. For a model to be universal, it should not preferably overfitting at all. [7; 12.]

2.11 Simulated-to-real performance gap

Since a simulated environment cannot capture the real world entirely with all of its aspects, the models trained in simulations may not work so well within real world environments. These inconsistencies are often called by the following term: simulated-to-real performance gap or sim2real performance gap.

In DeepRacer, actions have been taken to mimize this performance gap. In the simulation, the agent takes ten actions per second, and in DeepRacer, the vehicle runs at ten states per second. The starting position of the vehicle is always randomized so that the agent learns all parts of the track equally. [3.]

3 Methods

3.1 AWS DeepRacer client and service architecture

The architecture of AWS DeepRacer or in short DeepRacer, is built upon Amazon SageMaker, Amazon RoboMaker and other services such as Amazon S3. (See Figure 8.) [3.]

SageMaker is an Amazon AWS machine learning platform. In general, SageMaker is used to train various machine learning models such as DeepRacer. Amazon RoboMaker is a cloudbased service for developing, testing and deploying various robotic products. In DeepRacer, RoboMaker is used to create the agent and its environment. S3 is used for cloud-based storage capacity. DeepRacer also stores its models in S3. An in-memory database called Redis is used as an experience buffer for selecting training data to form the model’s policy network. [3.]

RoboMaker creates a simulated environment within the DeepRacer framework in which the agent drives on a chosen track. The model’s policy decides the agent’s actions. It is trained in SageMaker in the training sessions. Each run is represented by an episode. The training track is divided into a fixed number of steps. This is done in every episode. One step equals one data point mentioned in section 2.7 about hyperparameters. All the data points are then stored in Redis as an experience buffer. The experience buffer is then randomly drawn from by SageMaker in batches and fed to the neural network as the input data for updating the weights. After the model has been updated, it is stored to S3 so SageMaker can use the updated model for more training. This cycle only ends when training time stops. [3.]

In the beginning of the training session, SageMaker initializes the experience buffer with random actions. The amount of randomness in the agent’s actions should lessen as training time passes. [3.]

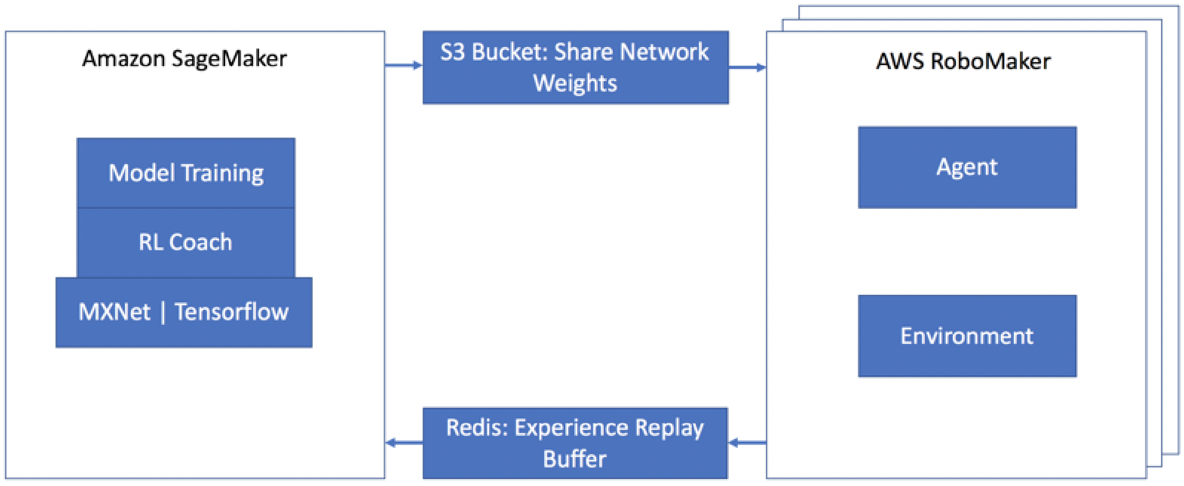


Figure 8. The AWS DeepRacer service architecture. Copied from AWS Developer guide [3].

3.2 AWS vehicle – DeepRacer

AWS DeepRacer also known as DeepRacer is a 1/18th scale model car. DeepRacer has a mounted camera, a LiDAR sensor, an on-board compute module and WiFi-connection. In order to drive itself within the track, DeepRacer makes decisions with its compute module. It can also be driven manually. DeepRacer has its own dedicated batteries for the computing and driving, respectively known as compute battery and driving battery. Figure 9 shows all the DeepRacer parts separately. Further details can be seen in Tables 2 and 3. [3.]

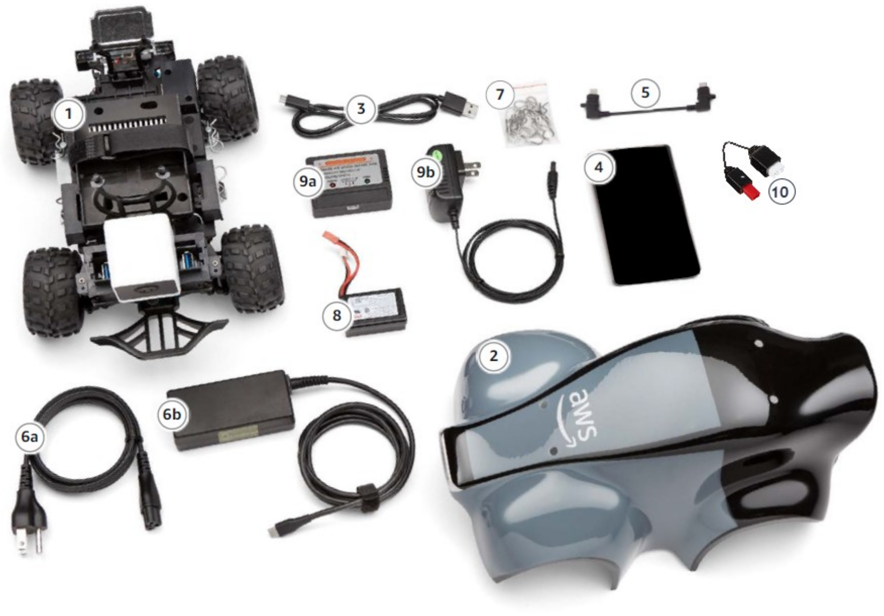
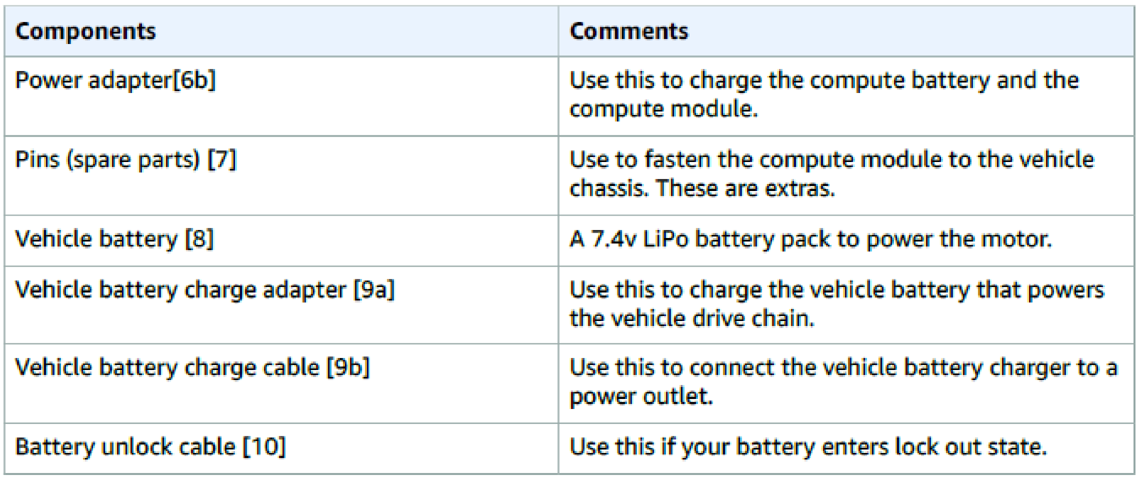


Figure 9. AWS DeepRacer vehicle in parts. Copied from AWS Developer guide [3].

Table 2. AWS DeepRacer vehicle parts 1. Copied from AWS developer guide [3].

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 3. AWS DeepRacer vehicle parts 2. Copied from AWS developer guide [3].



Components	Comments
Power adapter[6b]	Use this to charge the compute battery and the compute module.
Pins (spare parts) [7]	Use to fasten the compute module to the vehicle chassis. These are extras.
Vehicle battery [8]	A 7.4v LiPo battery pack to power the motor.
Vehicle battery charge adapter [9a]	Use this to charge the vehicle battery that powers the vehicle drive chain.
Vehicle battery charge cable [9b]	Use this to connect the vehicle battery charger to a power outlet.
Battery unlock cable [10]	Use this if your battery enters lock out state.

Physical DeepRacer

As mentioned before in section 2.11 simulated-to-real performance gap, there are differences between the physical car and the simulated car. This does not only include differences between performances. The physical DeepRacer needs its batteries to be charged, its compute module tested and physical parts calibrated. It also needs to be connected to a WiFi-network. When all this is done, you can upload a model into the car. [3.] This thesis will not go into details about how all of this is done since, only a simulated environment was used in the project.

Physical track

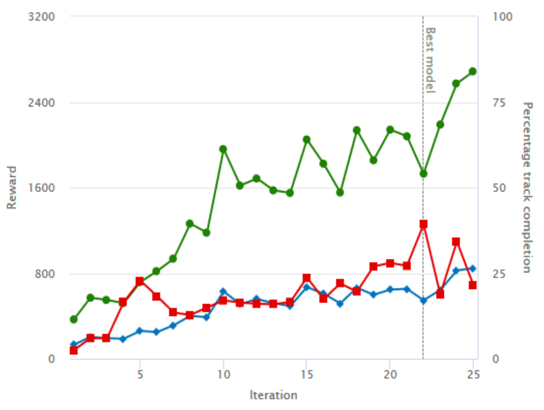
To drive DeepRacer autonomously and test your models in a physical DeepRacer vehicle, you will need to build a physical track. The physical track should resemble the simulated track as much as possible. There are certain requirements concerning the dimensions of the track and materials used. [3.] Since physical tracks were not used at all in this project, they are just mentioned here briefly.

3.3 Jupyter lab

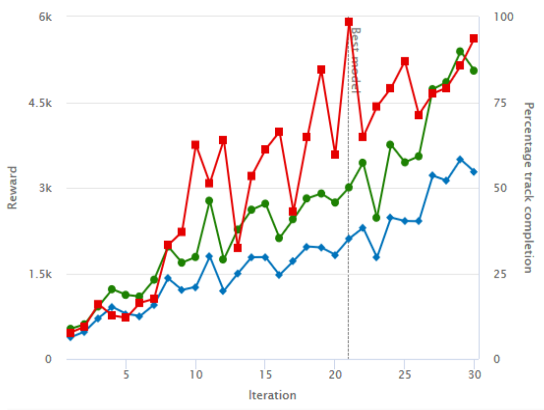
JupyterLab is a web-based development environment for Jupyter notebooks, code and data. It is flexible with a multitude of configuration and arrangement options. The Jupyter lab has many applications in data science, scientific computing and machine learning. The Jupyter lab was used to run log analysis for Sagemaker and Robomaker logs.

3.4 Amazon SageMaker and RoboMaker logs

Training a DeepRacer model can be arduous. It possible that the changes made are not that great after all. Knowing how to improve a model can be hard. Achieving a good model or improving the model can be difficult. Relying only on the default graph and evaluation results from DeepRacer do not tell you that much. You will know a bad mode from a good model, but it does not show how to improve your model (see Figure 10). Can you tell the difference in the models with these pictures? Yes, but only in a very broad way. Of course, you could do multiple training session and change the settings little by little and see the changes in the reward graphs but that is not very efficient. For example, there is always the possibility that the vehicle could over or understeer in a certain part of the track and you do not know it from the graph. To save money and time, it is recommended to do log analysis. [8.]



Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:04.567	19%	Off track
2	00:04.785	17%	Off track
3	00:04.323	17%	Off track

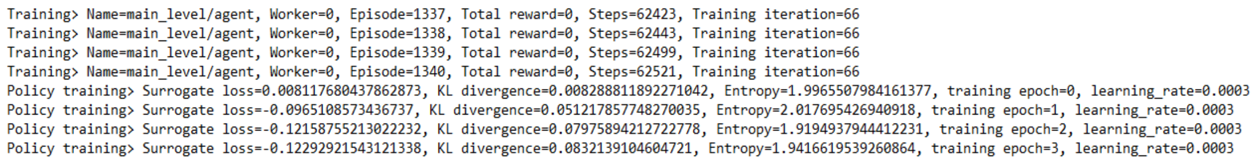


Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:16.077	100%	Lap complete
2	00:17.163	100%	Lap complete
3	00:16.026	100%	Lap complete

Figure 10. Two different models trained on the same track for 2 hours.

To help you understand the training process, how your model learns and interacts within its environment, there are logs for analysis. Within a log file for a model, there are logs for Amazon SageMaker and RoboMaker, metrics a in JSON file and SIMtrace in CVS files. After each training session is over, you can download logs for your model from the AWS client. Only SageMaker and RoboMaker logs are analysed here.

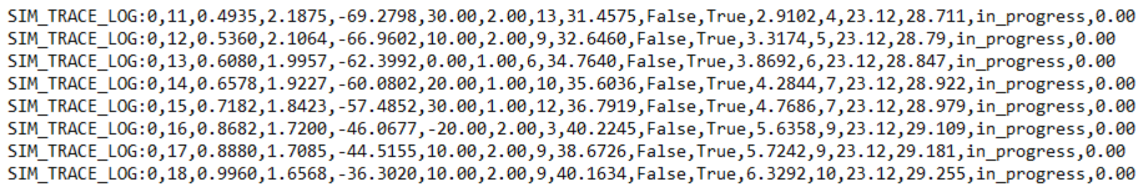
SageMaker logs contain data concerning the underlying neural network, its loss function and weights and their updates (see Figure 11). The RoboMaker logs have inside them the entire simulated training session. Each step has its own log line with various parameters such as iteration, episode, steps, start_at, progress, time, new_reward, speed, reward, time_if_complete, reward_if_complete, quintile and complete (see Figure 12).



```
Training> Name=main_level/agent, Worker=0, Episode=1337, Total reward=0, Steps=62423, Training iteration=66
Training> Name=main_level/agent, Worker=0, Episode=1338, Total reward=0, Steps=62443, Training iteration=66
Training> Name=main_level/agent, Worker=0, Episode=1339, Total reward=0, Steps=62499, Training iteration=66
Training> Name=main_level/agent, Worker=0, Episode=1340, Total reward=0, Steps=62521, Training iteration=66
Policy training> Surrogate loss=0.008117680437862873, KL divergence=0.008288811892271042, Entropy=1.9965507984161377, training epoch=0, learning_rate=0.0003
Policy training> Surrogate loss=-0.0965108573436737, KL divergence=0.051217857748270035, Entropy=2.017695426940918, training epoch=1, learning_rate=0.0003
Policy training> Surrogate loss=-0.12158755213022232, KL divergence=0.07975894212722778, Entropy=1.9194937944412231, training epoch=2, learning_rate=0.0003
Policy training> Surrogate loss=-0.12292921543121338, KL divergence=0.0832139104604721, Entropy=1.9416619539260864, training epoch=3, learning_rate=0.0003
```

Figure 11. SageMaker log file.

As mentioned before, SageMaker logs contain the neural network updates and RoboMaker logs contain the simulated training session actions and results.



```
SIM_TRACE_LOG:0,11,0.4935,2.1875,-69.2798,30.00,2.00,13,31.4575,False,True,2.9102,4,23.12,28.711,in_progress,0.00
SIM_TRACE_LOG:0,12,0.5360,2.1064,-66.9602,10.00,2.00,9,32.6460,False,True,3.3174,5,23.12,28.79,in_progress,0.00
SIM_TRACE_LOG:0,13,0.6080,1.9957,-62.3992,0.00,1.00,6,34.7640,False,True,3.8692,6,23.12,28.847,in_progress,0.00
SIM_TRACE_LOG:0,14,0.6578,1.9227,-60.0802,20.00,1.00,10,35.6036,False,True,4.2844,7,23.12,28.922,in_progress,0.00
SIM_TRACE_LOG:0,15,0.7182,1.8423,-57.4852,30.00,1.00,12,36.7919,False,True,4.7686,7,23.12,28.979,in_progress,0.00
SIM_TRACE_LOG:0,16,0.8682,1.7200,-46.0677,-20.00,2.00,3,40.2245,False,True,5.6358,9,23.12,29.109,in_progress,0.00
SIM_TRACE_LOG:0,17,0.8880,1.7085,-44.5155,10.00,2.00,9,38.6726,False,True,5.7242,9,23.12,29.181,in_progress,0.00
SIM_TRACE_LOG:0,18,0.9960,1.6568,-36.3020,10.00,2.00,9,40.1634,False,True,6.3292,10,23.12,29.255,in_progress,0.00
```

Figure 12. RoboMaker log file.

4 Results

The results are divided into 3 racing types: time trial, object avoidance and head-to-head racing. In time trial and object avoidance categories there are recommended amounts of training for each specific trial whatever training on a simpler track or a more complicated one. Various recommendations concerning agent parameters. How changing agent parameters affect the model. Then showcasing the best model from both categories. After this has been discussed, there are insights concerning hyperparameters and training. Finally, what should be looked at when doing log analysis.

4.1 Time trials

Recommended amount of training for simpler tracks

In time trial, for simpler tracks such as The 2019 DeepRacer Championship Cup, re:Invent 2018, AWS Summit Raceway and Oval Track (see Figure 13), the recommend training time is two hours with default hyperparameters. . Modified hyperparameters could be tried out but it is important to keep in mind that increasing or decreasing default hyperparameters will either increase or decrease the required amount of training. In general, for simpler tracks in time trial, you will know within two hours of training if your model is good or not and whether you should train your model more or just analyse the logs and move on.

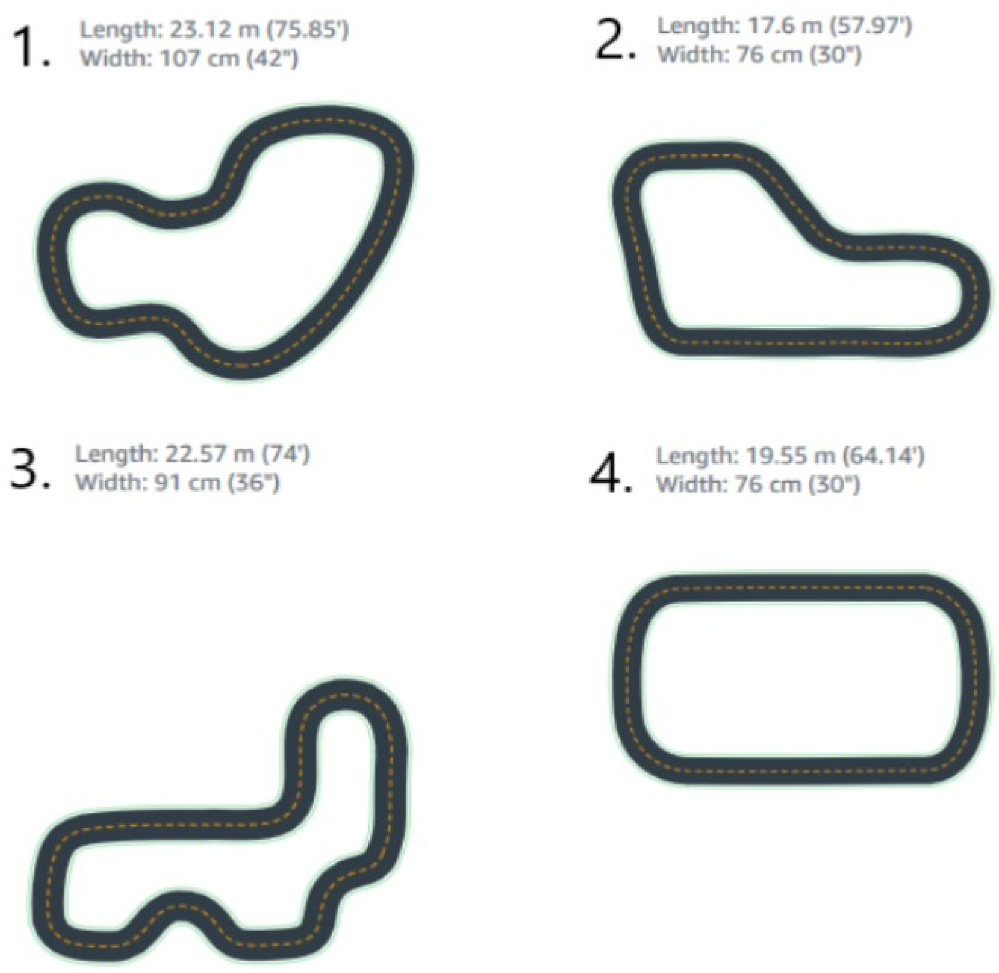
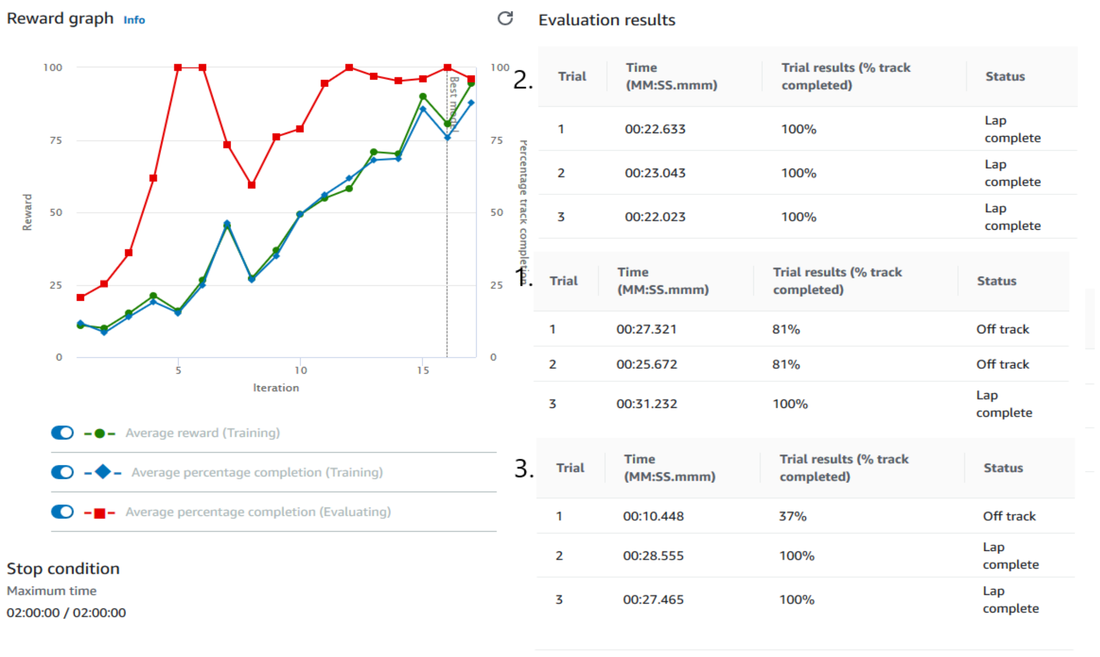


Figure 13. Racing tracks: 1. The 2019 DeepRacer Championship Cup, 2. re:Invent 2018, 3. AWS Summit Raceway and 4. Oval Track. Copied from AWS Client [13].

Figures 14, 15 and 16 show the results of the models trained in these tracks. In this context we can look at the reward function and evaluation results provided within the AWS DeepRacer console, so as to determine whether a model is good and whether the model has been trained for long enough.


Reward graph
Info
Reward
Iteration
Percentage track completion
Best model
100
75
50
25
0
5
10
15
Average reward (Training)
Average percentage completion (Training)
Average percentage completion (Evaluating)

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

Evaluation results

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

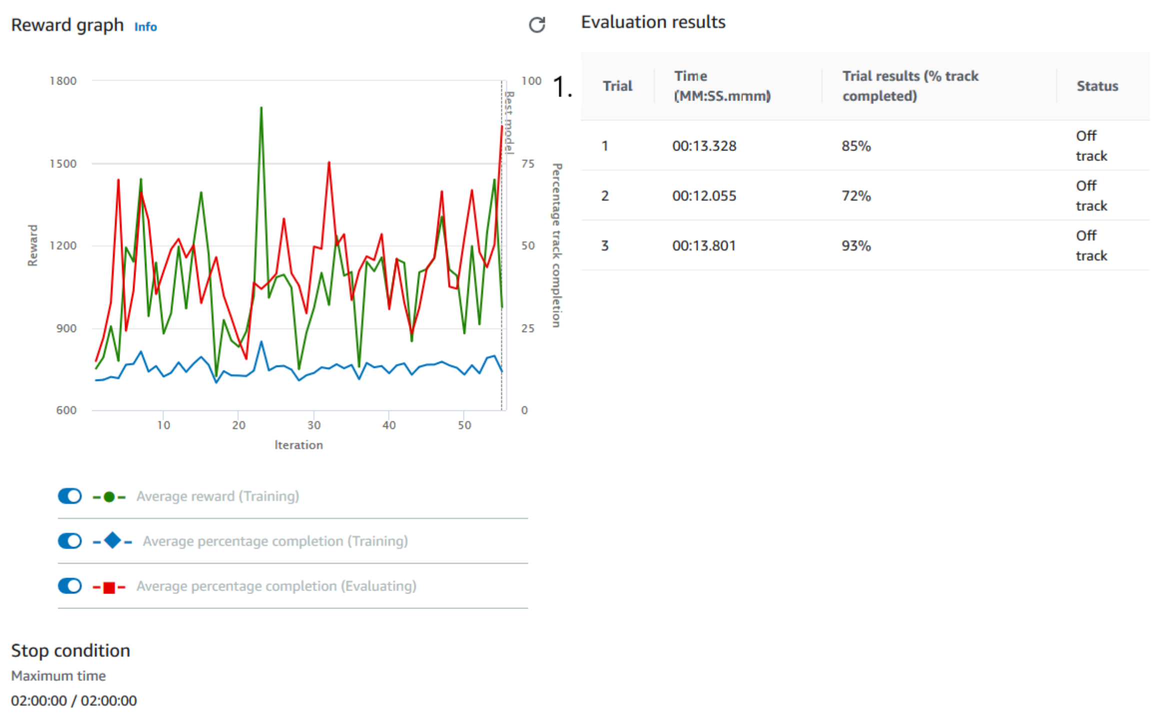
1.

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 14. One of the earliest models. (Cf. Figure 13, track 2.)

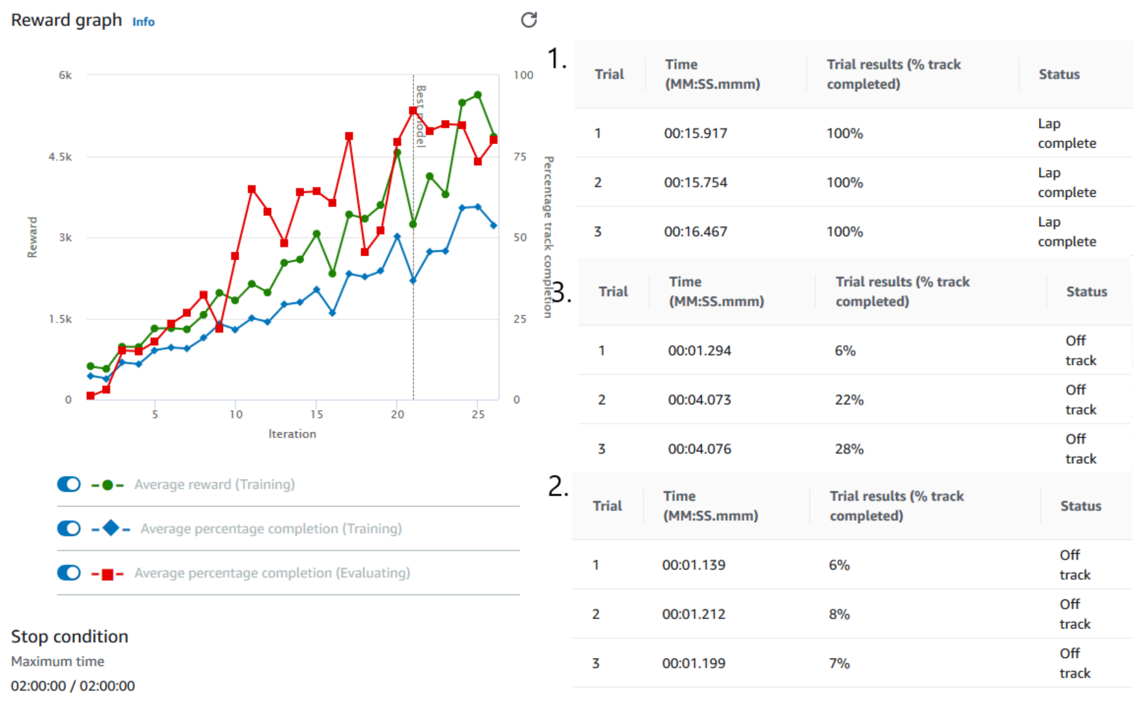

Reward graph Info
Reward
Percentage track completion
Iteration
Best model
1.
Average reward (Training)
Average percentage completion (Training)
Average percentage completion (Evaluating)

Evaluation results

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

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

Figure 15. A model testing to increase the entropy hyperparameter. A highly unstable model.


Reward graph Info
Reward
Percentage track completion
Iteration
Best model
Average reward (Training)
Average percentage completion (Training)
Average percentage completion (Evaluating)

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

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

Figure 16. A later model in Time Trial trained at the 2019 DeepRacer Championship Cup

Recommended amount of training for more complex tracks

For more complex tracks, it will probably take significantly more training time than for more simple tracks. The recommended training time for a model for initial training time is three to four hours. Then it is important do the log analysis to see if the model shows promise. It is also important to evaluate the quality of the training for the model (see Figure 17). In Figure 17, after three hours of training, we can see the relatively steady rise of rewards and progress per iteration. Most of the progress is from zero to thirty percentage. The model has not completed a lap yet, but the model is quite steadily rising in performance per iteration, so the model is worth training for more. In general, it is wiser to train your models in smaller training time sessions since you will have more control over your training periods. Figure 18 shows the Lars Loop track in which the model in figure 17 was trained in.

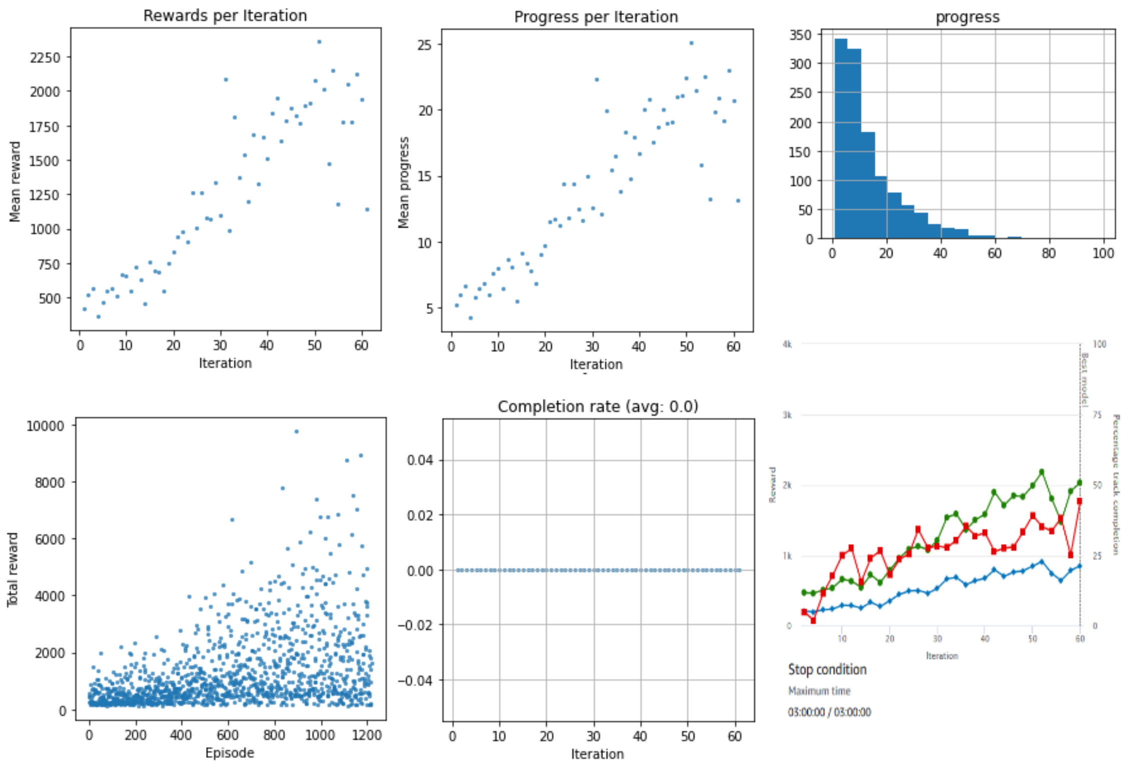


Figure 17. The training progress of the model displayed in graphs.

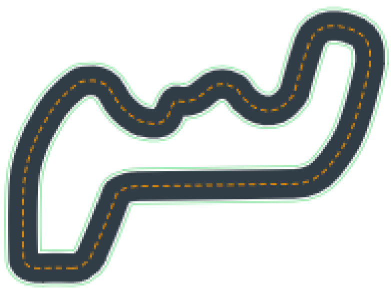


Figure 18. The Lars Loop track. Copied from AWS client [13].

Agent parameters

In Time Trials, you will be fine with just the regular camera. Since the track is empty, you do not need the stereo camera for depth of view or the LiDAR for sensing objects.

As mentioned already in section 2.4, both the continuous and discrete actions space have their own benefits. Most of the models created during this project were discrete models. At the start of the testing period of the models, the discrete models did a little bit better than the continuous models, but it could just be because of the limited training time. As already mentioned in section 2.4, continuous models likely train slower than discrete models. There is not enough data to show whether the discrete or continuous action space should be preferred.

Speed is the most critical factor in agent parameters. The model's speed has the most significant impact on the performance of the model. More speed equals fewer mistakes the model can make in difficult turns. In addition, training will take longer since the model is going off-track more often during the starting periods of training and in sharp turns the agent can start to drift or even spin if the steering angle is too large. More speed makes the models less stable. The testing did not give any conclusive evidence of where the sweet spot for the agent's speed would be, but as figures 19, 20 and 21 show, there is significant difference between model A with maximum speed of 2 m/s and model B with maximum speed of 3 m/s. Other than the difference in speeds, the models share the same action space.

Between Model A and B there are significant differences in rewards per iteration, times per iteration and progress per iteration. Model A's and B's total reward also shows a clear difference. Model A has more evenly split reward across all the episodes than model B. Meaning that as can been seen from figures 19 and 20 total rewards to episodes graph, model A clearly has more balanced growth over the iterations than model B. Model A also has around five times better average completion rate compared to model B. Model A's completion rate also grows as iterations go by significantly more than model B's.

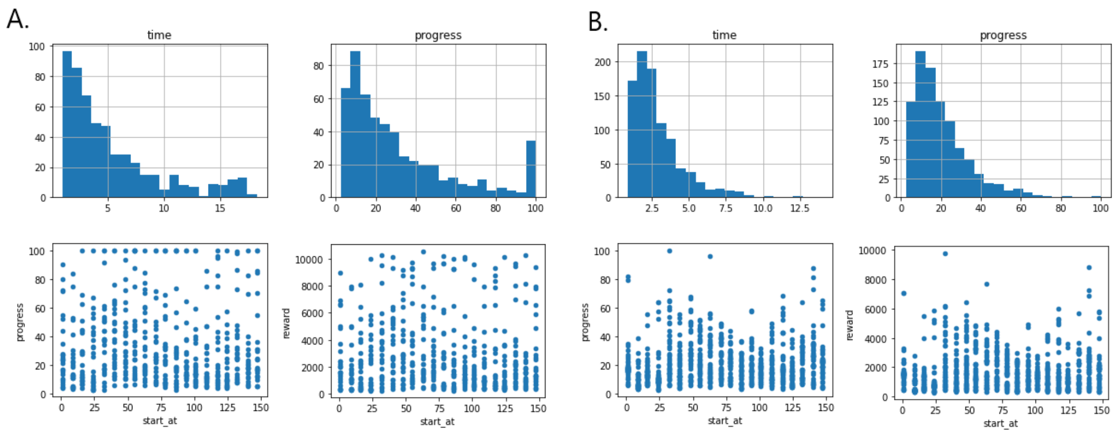


Figure 19. Progress of models A and B over a training period.

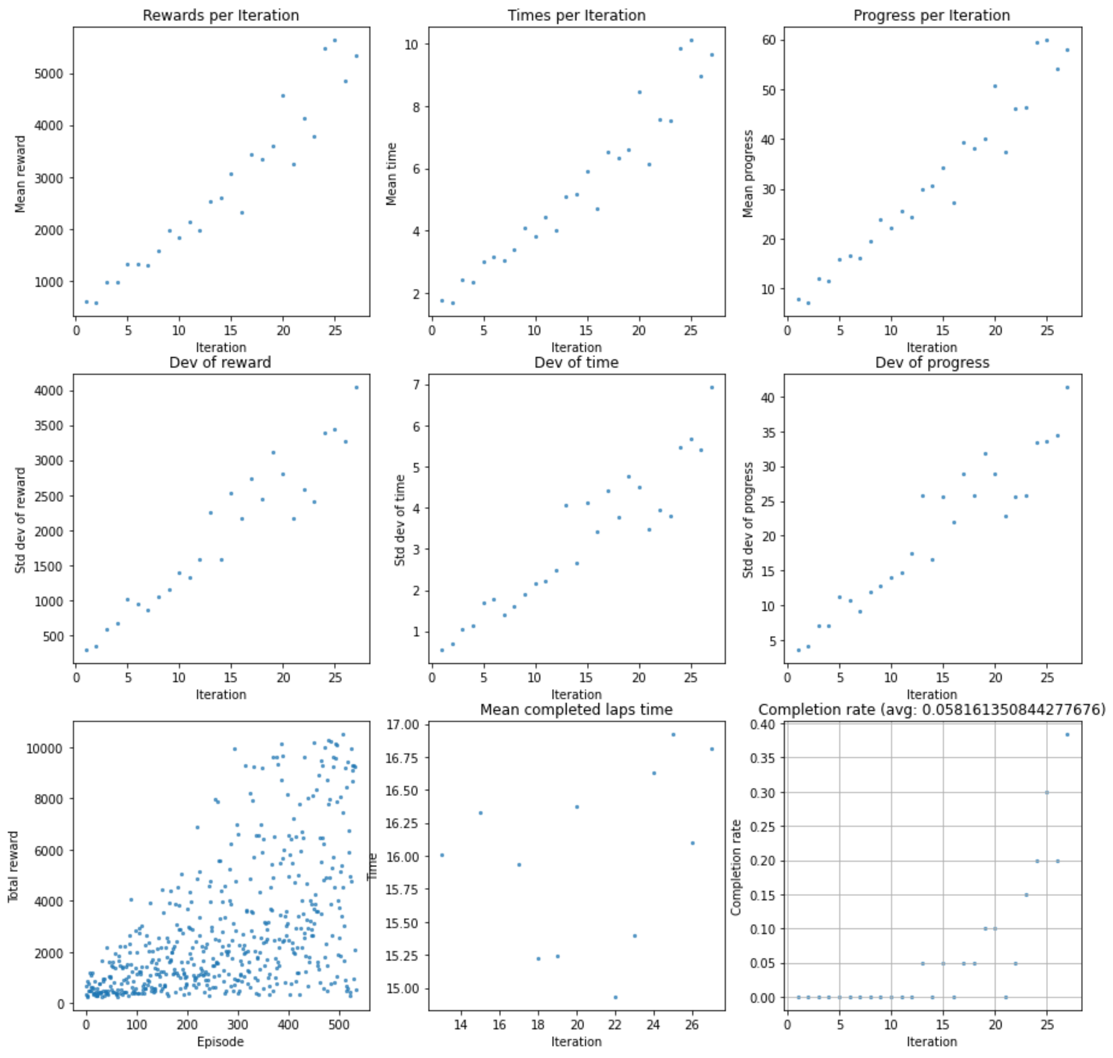


Figure 20. Model A

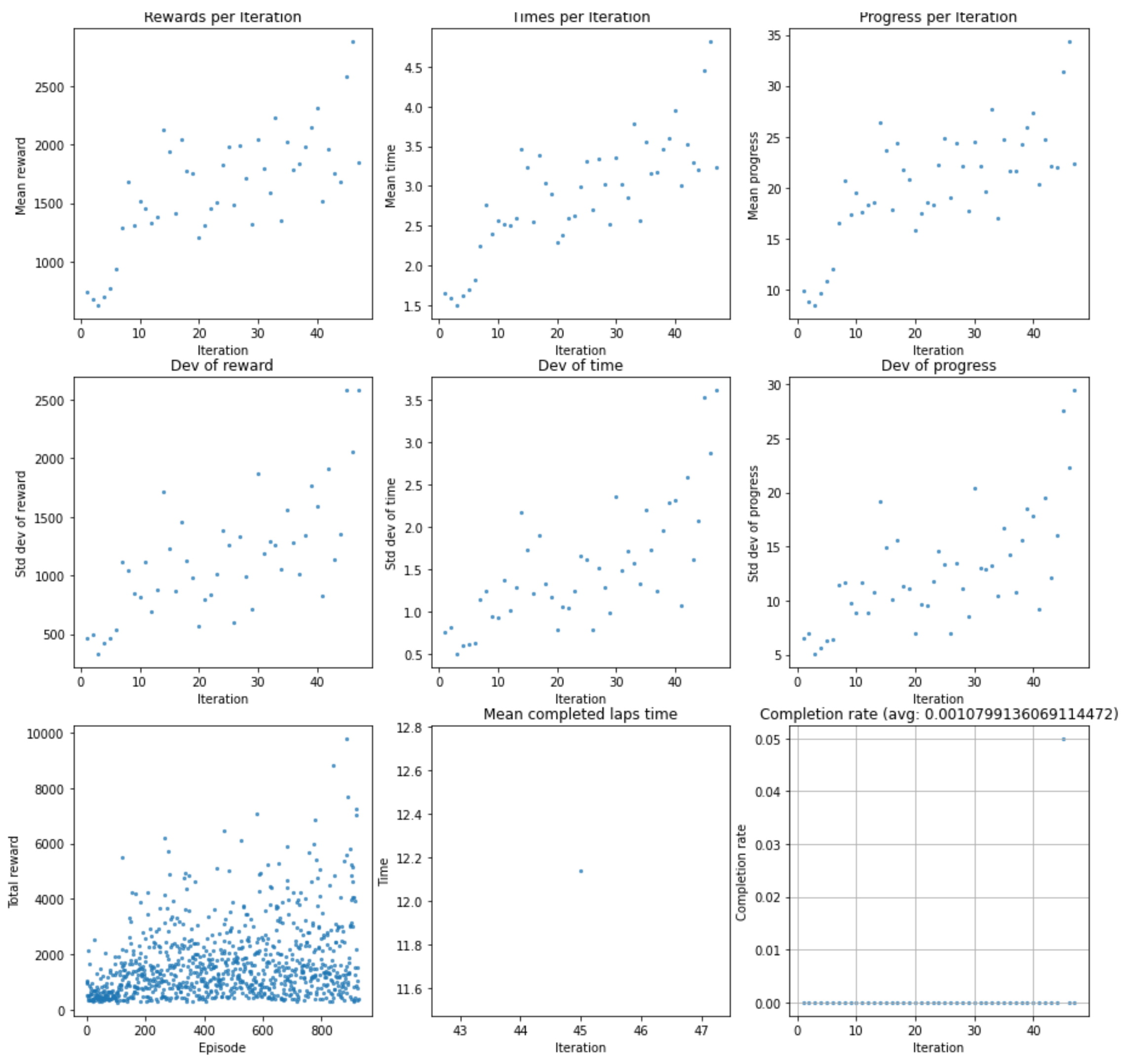


Figure 21. Model B

Best reward function

When you begin to examine DeepRacer and build your first model, you will probably use the default follow the center line function showed in Listing 1. The first model done in this project used a reward function fairly similar to the one showed in Listing 1. During this project various reward functions were tried. Most of the reward functions tried were quite good, some bad, but the best model was trained with a reward function originating from a AWS DeepRacer re:Invent 2019 youtube video. In the video one of the reward functions of the models showcased was simple but yet it seemed quite effective and it felt interesting to try out. [7] After deciding to give the reward function a try, the reward function seemed to work the best out of all the reward functions tried during this project, thus it is showed here. It is a simple reward function, where you heavily reward the agent for the amount of progress it has made divided with the steps it has taken to incentivize fewer steps being taken in a lap. Since time trial is all about speed, you should reward the agent for its speed too. Minimal reward if conditions are not met.

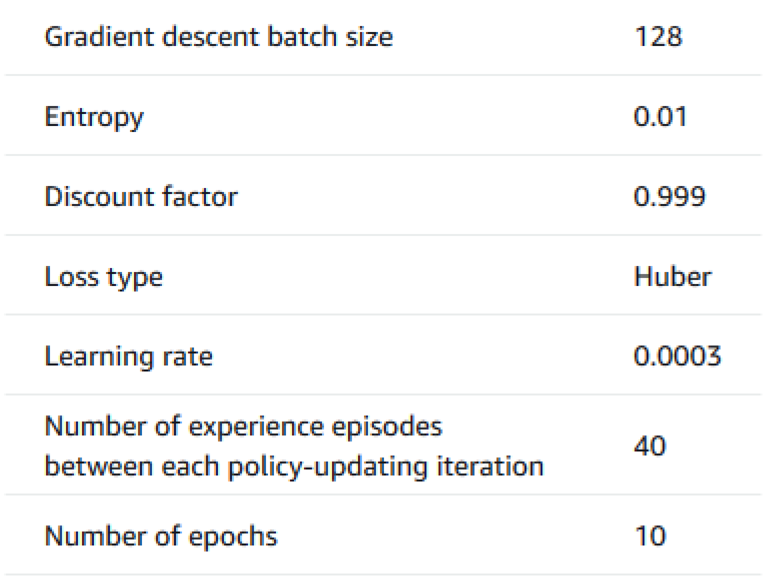
```
 def reward_function(params):

    if params['all_wheels_on_track'] and params['steps'] > 0:
        reward =
((params['progress'])/params['steps'*100)+(params['speed']**2)
    else:
        reward = 0.01

    return float(reward)
```

Listing 2. The Time Trial reward function

Best model

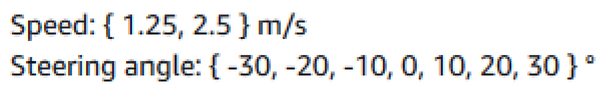
The model trained for the LarsLoop track is probably the best time trial model done in this project. The model has not converged. It would need more training to converge. However, the model is the best out of all the time trial models, because the model has proven to be able to learn more after multiple training sessions and it has the potential to reach a state where you could possibly use the model in an AWS community race.

Probably a batch size of 64 would have been fine but desiring a little more stable and smoother updates, the batch size was increased to 128. The default learning rate and episodes between each policy update was raised to 40 experience episodes from 20. The model would have probably been just fine with 20 experience episodes between policy-updates. The decision to increase experience episodes to 40 was made in order for the agent to surely learn all the actions needed. Figures 22 and 23 show the model’s hyperparameters and action space.



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	40
Number of epochs	10

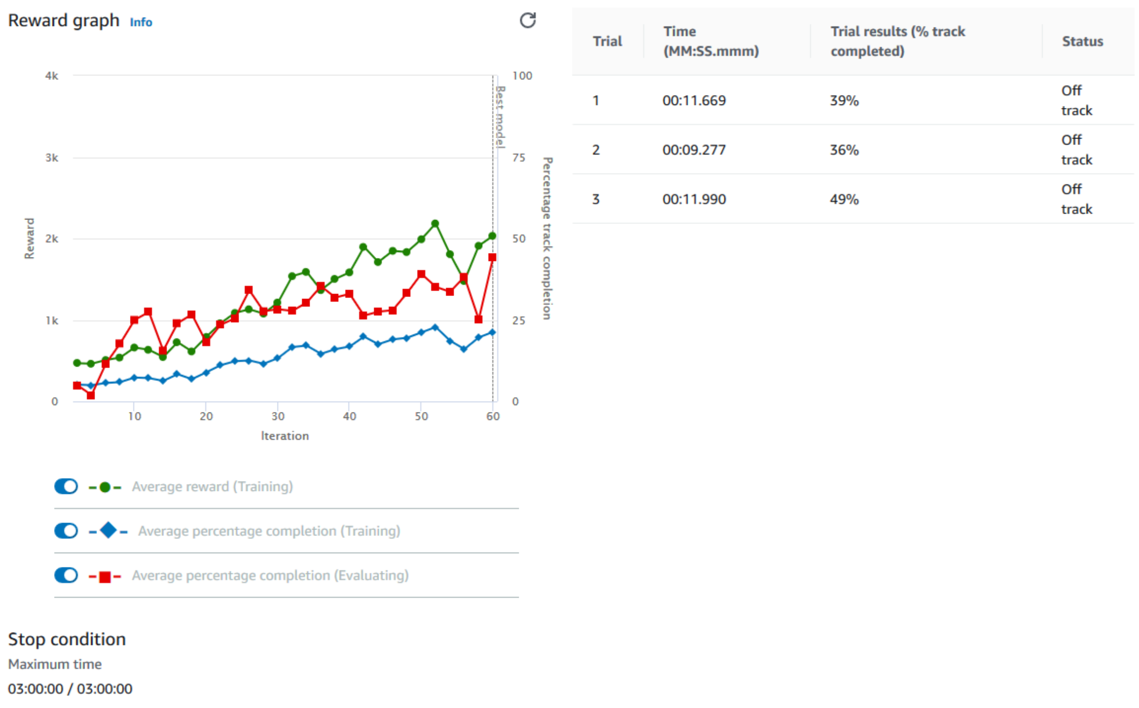
Figure 22. Model hyperparameters.



Speed: { 1.25, 2.5 } m/s
Steering angle: { -30, -20, -10, 0, 10, 20, 30 } °

Figure 23. Model action space.

Next, it is important to look at model performance graphs after initial training of three hours. The graphs are shown in Figures 24 and 25. The model cannot complete even a single lap after the first training session in evaluation, but its progress, reward and total reward grow so nicely intact it is worth training for more. In Figure 25, it can be seen that the model is still within its early learning period since most of the episodes have a track completion rate of 0 to 20 percent. The model clearly needs to be trained for more.


Reward graph Info
Reward
Percentage track completion
Best model
Iteration
Average reward (Training)
Average percentage completion (Training)
Average percentage completion (Evaluating)

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:11.669	39%	Off track
2	00:09.277	36%	Off track
3	00:11.990	49%	Off track

Stop condition
Maximum time
03:00:00 / 03:00:00

Figure 24. Model reward graph and evaluation results. Training session 1.

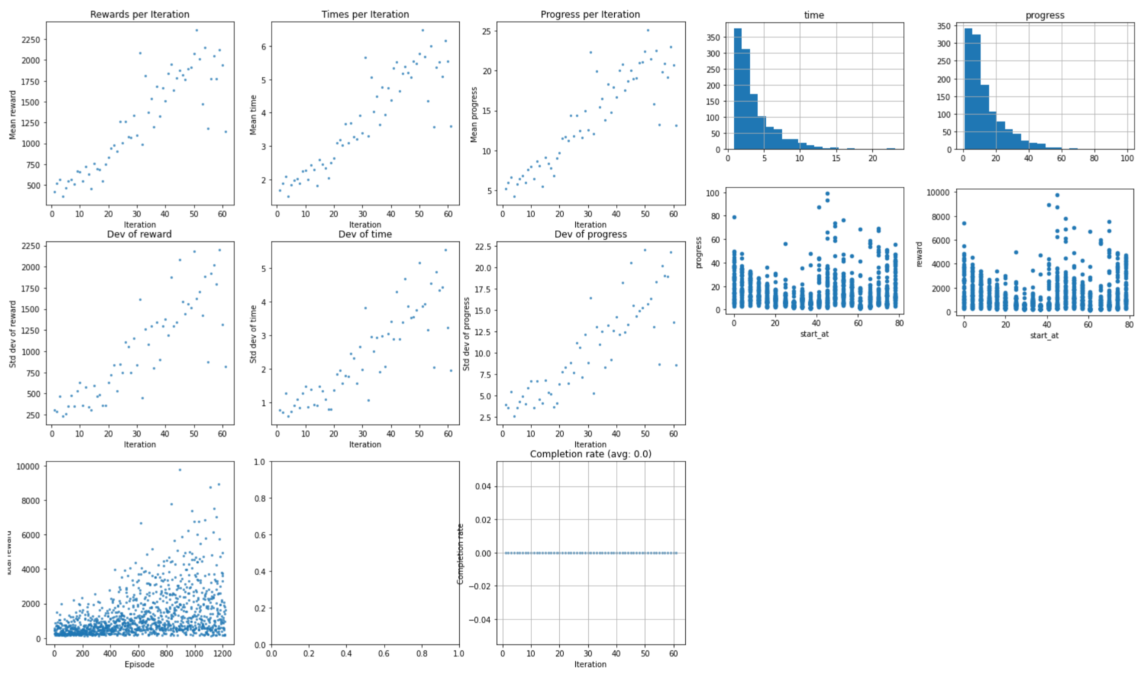
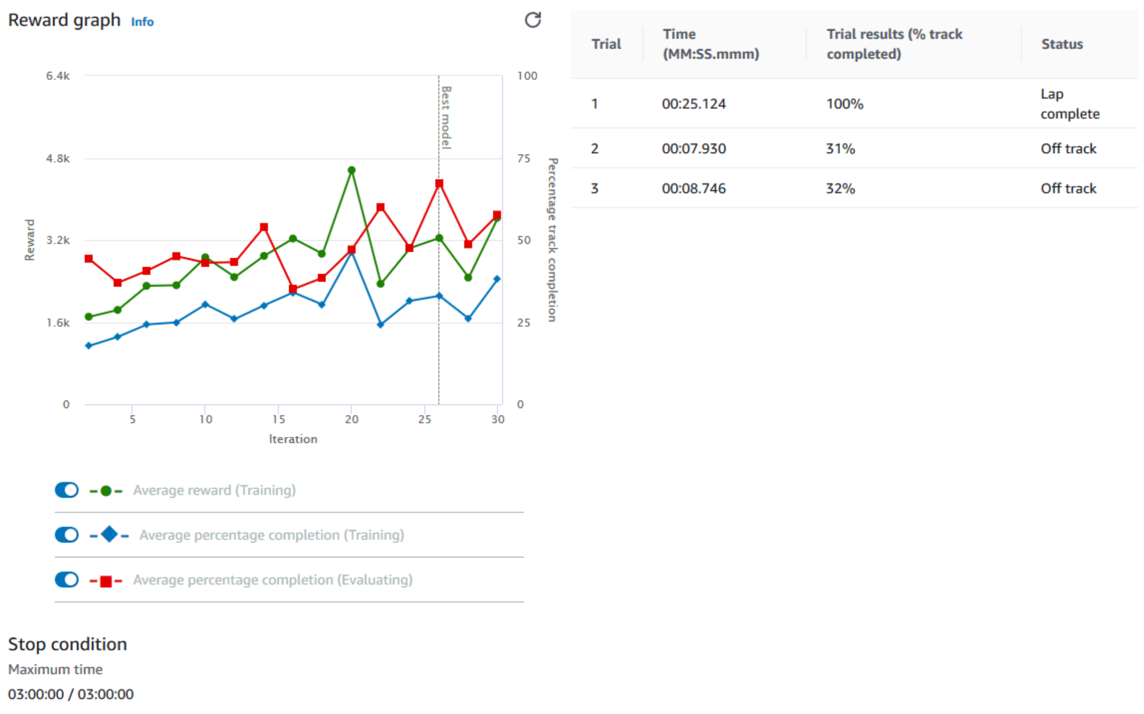


Figure 25. Model training performance graphs. Training session 1.

After three more hours of training with same hyperparameters, the model shows improvement yet again. In evaluation, the model completes one lap round the track (see Figure 26). It is also important to look at Figure 27 for more details about the training session.


Reward graph Info
6.4k
4.8k
3.2k
1.6k
0
Reward
100
75
50
25
0
Percentage track completion
Best model
5
10
15
20
25
30
Iteration
Average reward (Training)
Average percentage completion (Training)
Average percentage completion (Evaluating)

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:25.124	100%	Lap complete
2	00:07.930	31%	Off track
3	00:08.746	32%	Off track

Stop condition
Maximum time
03:00:00 / 03:00:00

Figure 26. Model reward graph and evaluation results. Training session 2.

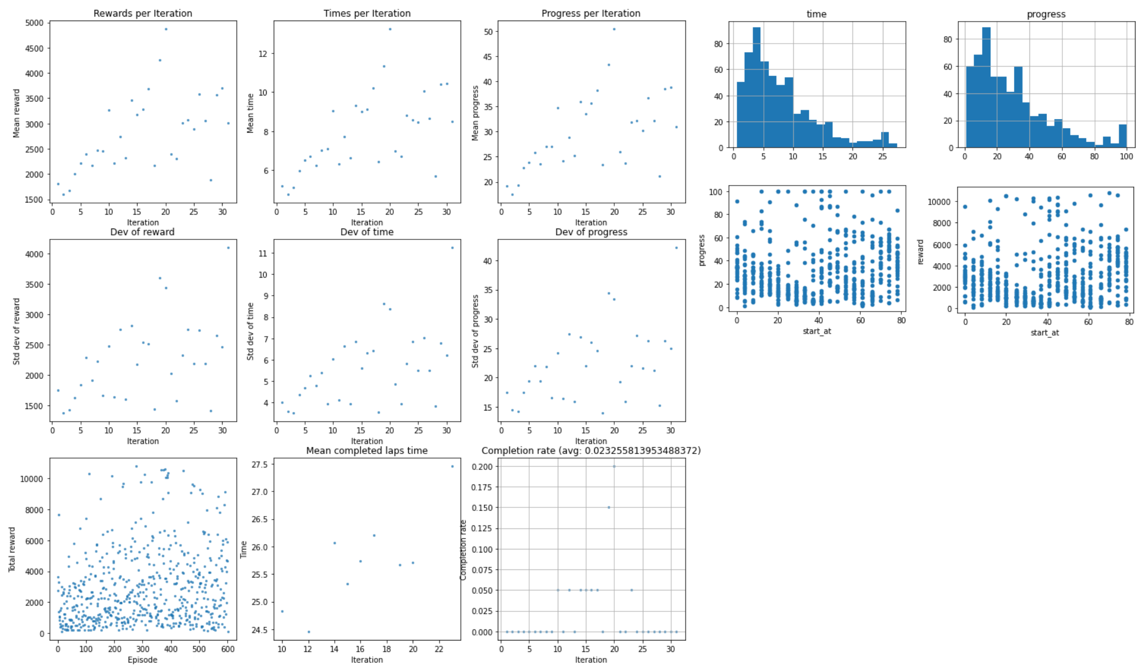
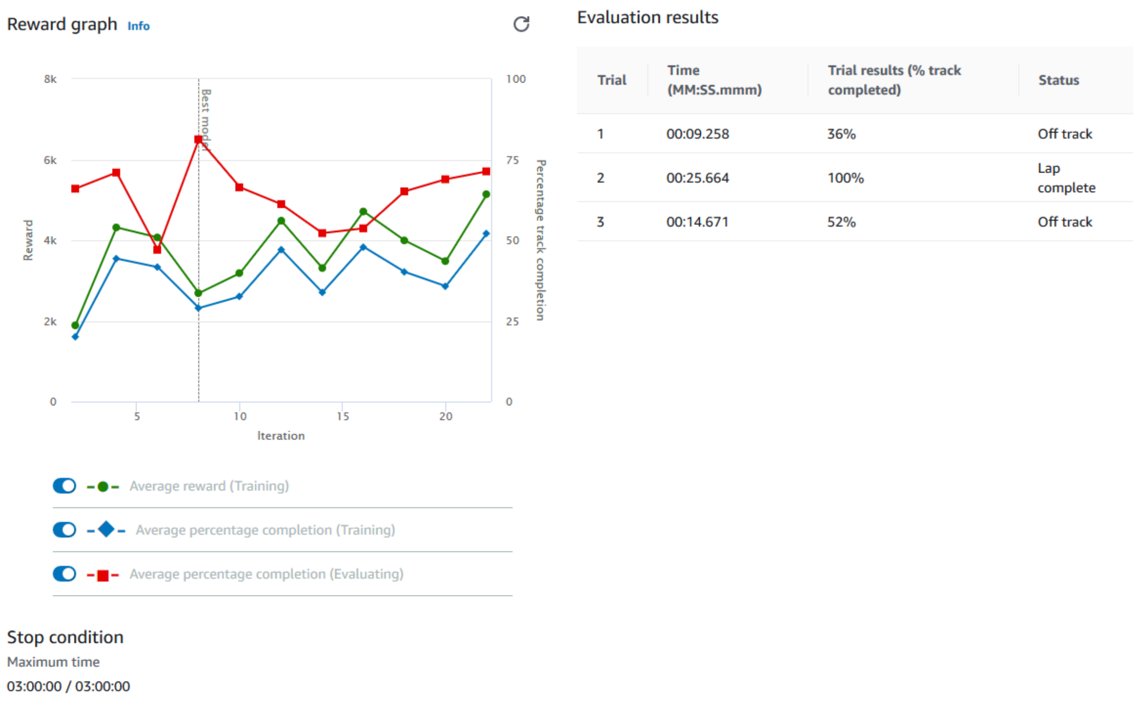


Figure 27. Model training performance graphs. Training session 2.

Figure 27 shows how the model now has more evenly split episode progress and that the model's average completion rate has grown to 2.3 %. Progress and rewards per iteration continue to grow, but both start to scatter in the plot as iterations pass. To counter the increasing scattering within rewards and progress, learning rate for the next training session should be lowered. The model can still learn more and has not converged.

For the third training session, the learning rate is lowered from 0.0003 to 0.0001. Other hyperparameters stay the same. In evaluation, the agent completes yet again only one lap (see Figure 28).


Reward graph
Info
8k
6k
4k
2k
0
Reward
100
75
50
25
0
Percentage track completion
Best model
5
10
15
20
Iteration
Average reward (Training)
Average percentage completion (Training)
Average percentage completion (Evaluating)

Stop condition
Maximum time
03:00:00 / 03:00:00

Evaluation results

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:09.258	36%	Off track
2	00:25.664	100%	Lap complete
3	00:14.671	52%	Off track

Figure 28. Model reward graph and evaluation results. Training session 3.

Figure 29 shows that rewards, times and progress per iteration looks little bit less scattered and they do increase over the training session, probably lowering the learning rate even more would probably have been fine, maybe to 0.00008 or 0.00007. The average completion rate looks a lot better than in the second training session. Over the iterations, the completion rate grows steadily up reaching a 25 percent mark within iteration 21 reaching its local maxima. Before the training ends, the completion rate rapidly falls to 0 %. This is because the model reaches local minima just at the end of the training. Considering that rewards, progress and total reward increase over the training time, it would be fair to say the model still can be trained for more and it has not converged. The model has not been trained for the fourth training session, but the graphs of this iteration should show enough evidence that the model is still within its learning period and should be trained for at least one or more sessions until it converges.

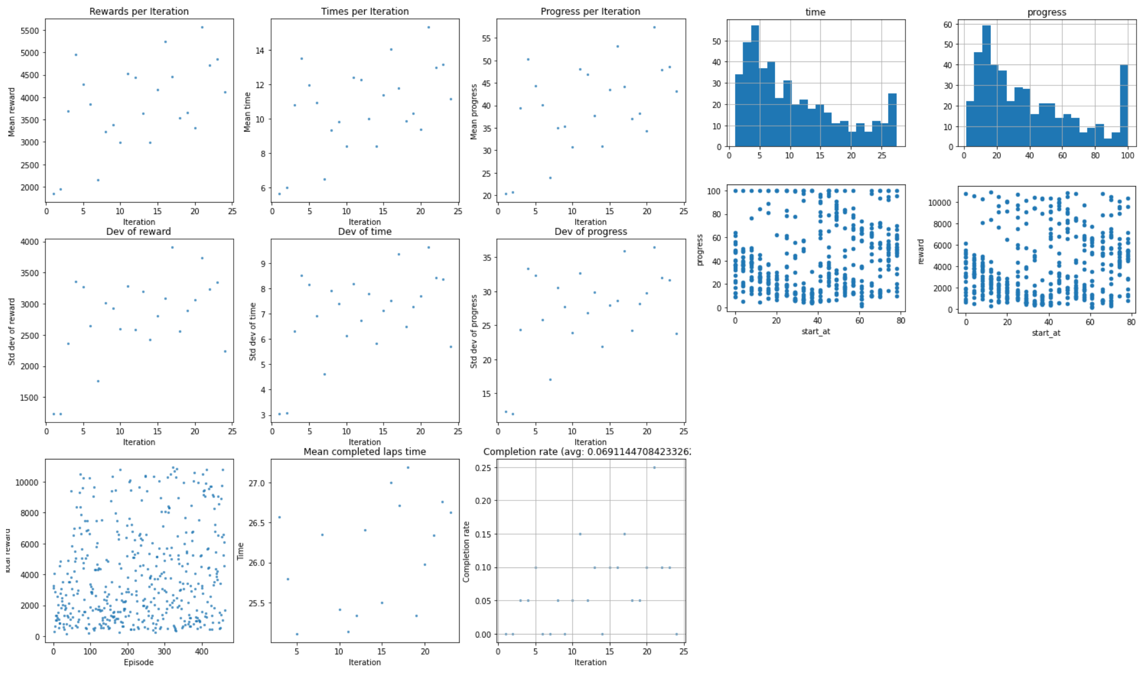


Figure 29. Model training performance graphs. Training session 3.

Figure 30 demonstrates the fastest lap the model had in training session 3. Figure 31 demonstrates the optimal racing line for the Lars Loop.

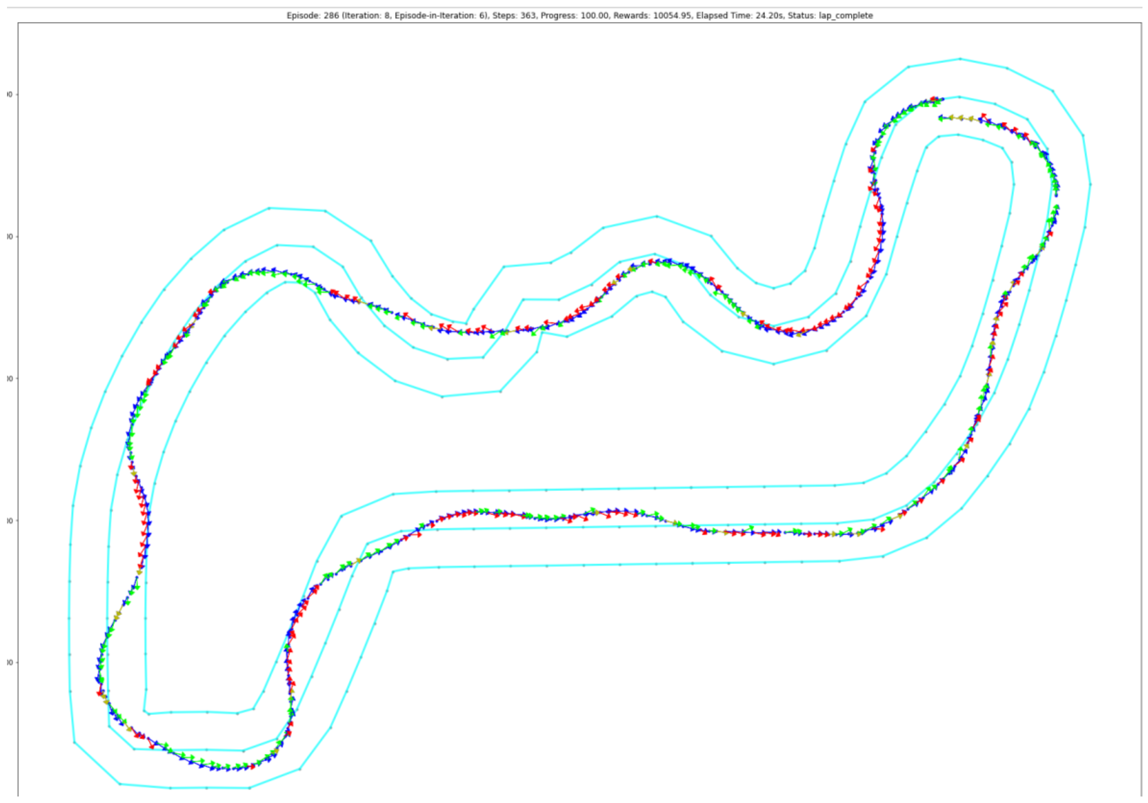


Figure 30. The fastest lap in the third training session of the model. 24.2 seconds.

The optimal racing line calculated by an algorithm for the LarsLoop track can be seen in Figure 31. There is still considerable room for improvement especially concerning the racing line the vehicle takes. The vehicle is clearly oversteering in more than one turn round the track. This can be fixed either by adjustments in the reward function or by simply training the model for more since the fewer steps the models takes to complete the lap, more rewards the model will receive. It is probably faster to get results from the first option, but the second option is probably better in the long run since the model learns an optimal policy on its own without outside interference.

```
These should be the same:  ((83, 2), (83, 2))
Original centerline length: 40.91
New race line length: 35.02
```

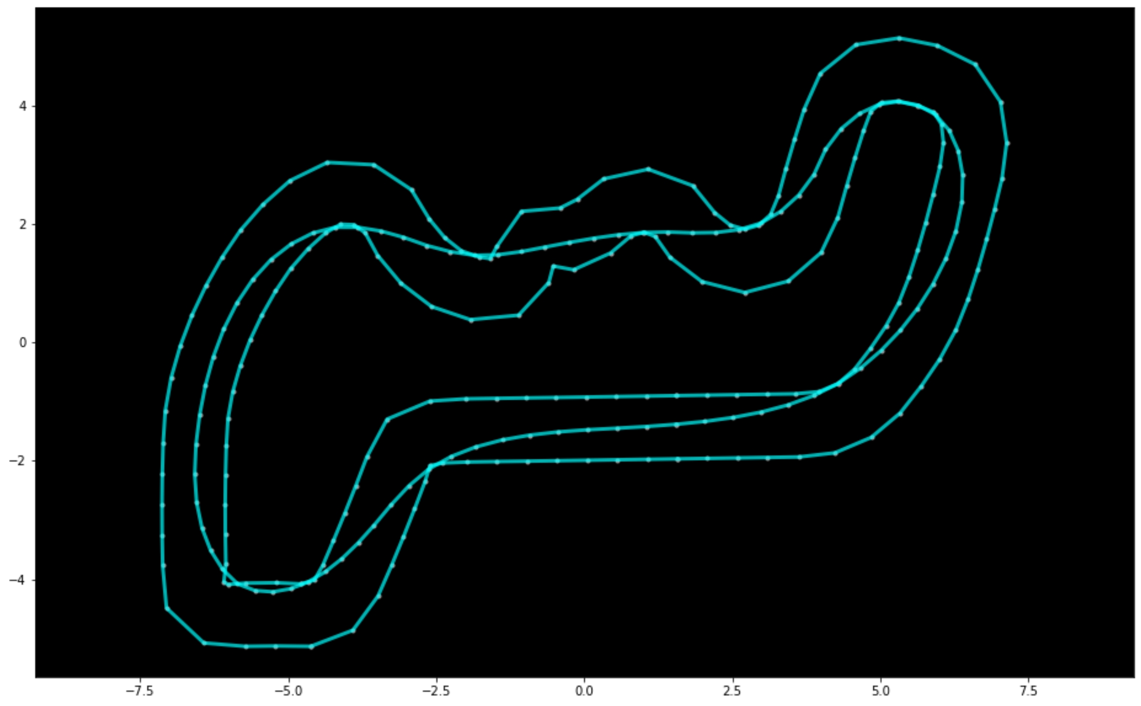
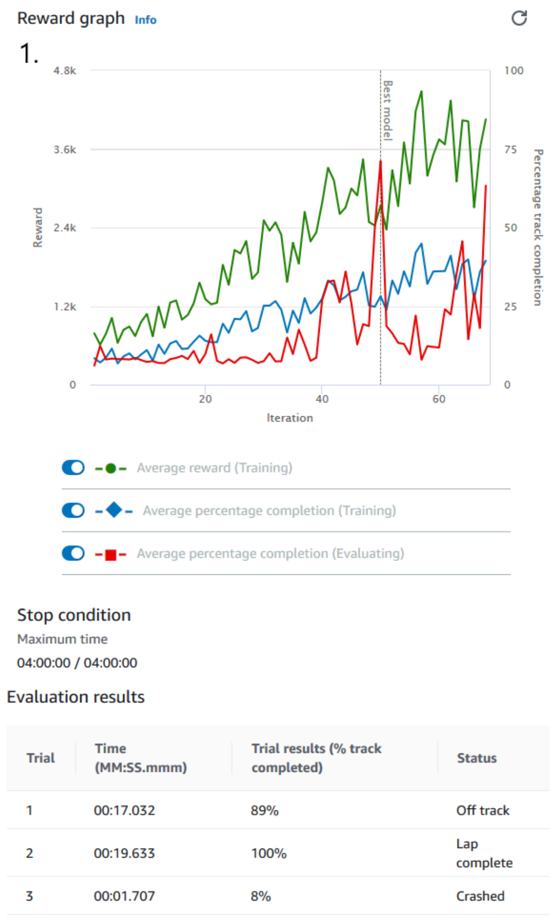


Figure 31. Optimized racing line on the Lars Loop track.

4.2 Object avoidance

Recommended amount of training for simpler tracks

For object avoidance, you can choose between from one to six objects either randomly or statically placed. There is a huge difference between having 1 or 6 objects. There is even a larger difference between randomly and statically placed objects. Statically placed objects can be passed without the model learning that there is actually an object there, the same cannot be said concerning randomly placed objects. Models, which have statically placed objects, train faster than randomly placed objects. Models 1 and 2 in figure 32 are not compatible with each other since model 1 has maximum speed of 2 m/s and model 2 has maximum speed of 1 m/s. In addition, there are minor differences between their reward functions. The aim here is to prove that around 4 hours should be enough for the initial training time for object avoidance models. For additional information see Figures 33 and 34.


Reward graph Info
1.
4.8k
3.6k
2.4k
1.2k
0
Reward
20
40
60
Iteration
Best model
100
75
50
25
0
Percentage track completion
Average reward (Training)
Average percentage completion (Training)
Average percentage completion (Evaluating)

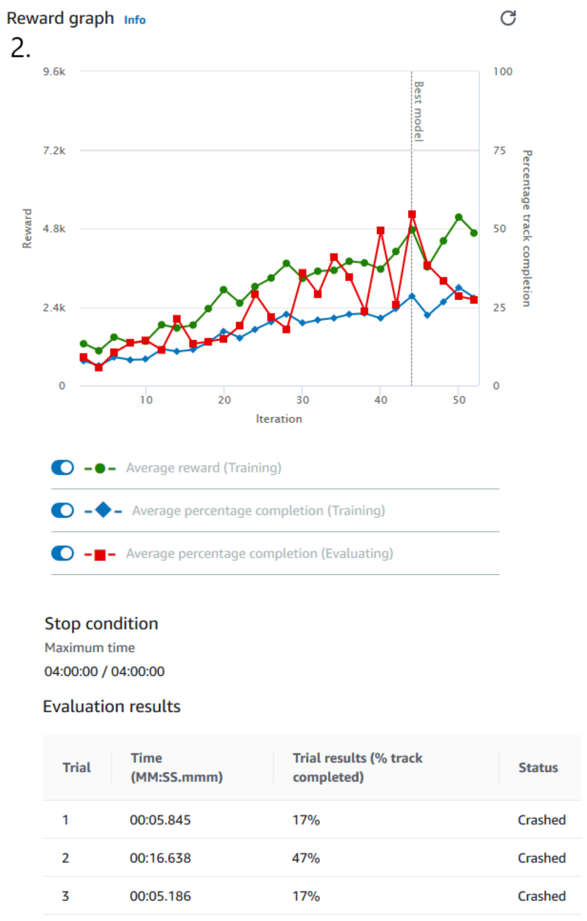
Stop condition

Maximum time

04:00:00 / 04:00:00

Evaluation results

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:17.032	89%	Off track
2	00:19.633	100%	Lap complete
3	00:01.707	8%	Crashed


Reward graph Info
2.
9.6k
7.2k
4.8k
2.4k
0
Reward
10
20
30
40
50
Iteration
Best model
100
75
50
25
0
Percentage track completion
Average reward (Training)
Average percentage completion (Training)
Average percentage completion (Evaluating)

Stop condition

Maximum time

04:00:00 / 04:00:00

Evaluation results

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:05.845	17%	Crashed
2	00:16.638	47%	Crashed
3	00:05.186	17%	Crashed

Figure 32. Model 1 has stationary objects and Model 2 has randomly placed objects. Both have six objects on track.

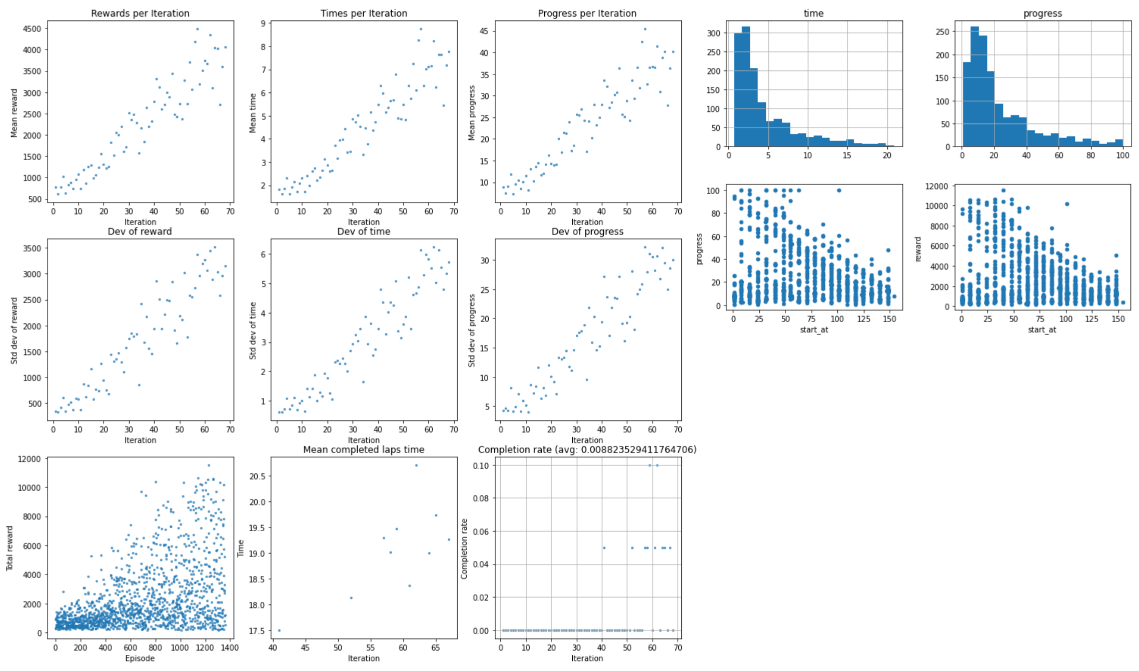


Figure 33. Model 1 performance graph.

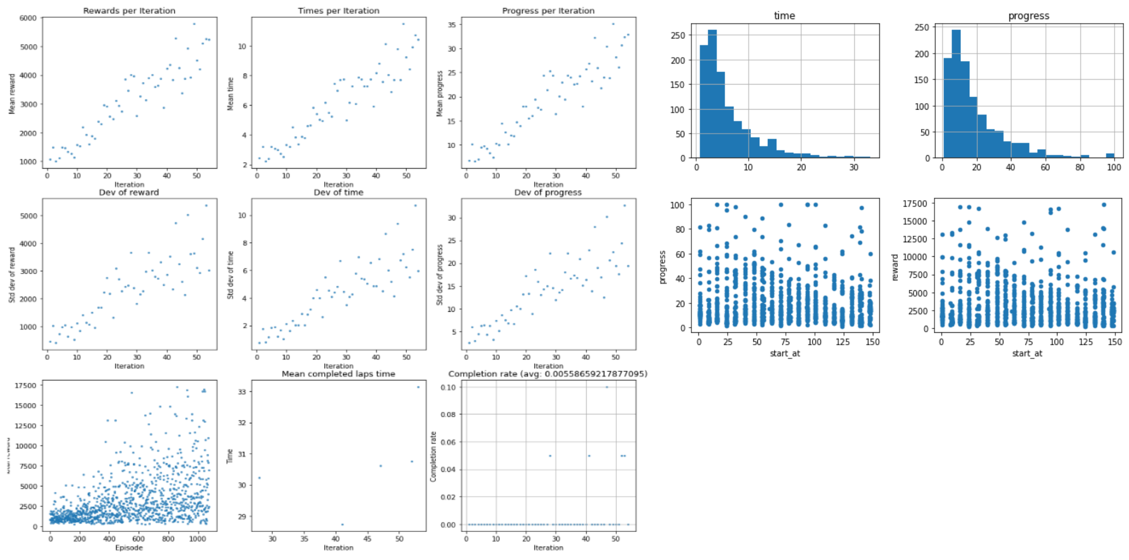


Figure 34. Model 2 performance graph.

In figures 33 and 34 we can clearly see that both models can still be trained for more. In both figures progress steadily grows. The total reward grows as the training continues.

Recommended amount of training for more complex tracks

None of the object avoidance models were trained on more complicated tracks. All the models were trained on the 2019 DeepRacer Championship Cup track. It can be concluded that training will take more time than on simpler tracks.

Agent parameters

The agent should have either a mono camera with a LiDAR sensor or a stereo camera with a LiDAR sensor. Mono camera can learn over time how to avoid statically placed objects, since the vehicle will just learn to pass the stationary object in order to gain more rewards. Randomly placed objects require you to use a stereo camera or LiDAR or both in order to avoid them reliably. It could be possible to effectively avoid randomly placed objects with a mono camera and a LiDAR sensor.

All the object avoidance models had a discrete action space with differences being between action lists. The recommendation is to use the discrete action space for faster training and convergence.

Speed is again a limiting factor. The more speed you have, the more challenging the training is going to be, as was shown in Figures 19, 20 and 21 in section 4.1.

Best reward function

Listing 3 illustrates the reward function with the best results.

```
import math
def reward_function(params):
    objects_distance = params['objects_distance']
    _, next_object_index = params['closest_objects']
    objects_left_of_center = params['objects_left_of_center']
    is_left_of_center = params['is_left_of_center']
    objects_location = params['objects_location']
    # Initial reward
    reward = 1e-3
    # Reward if the agent stays inside the two borders of the track and gets
rewarded more for better progress/steps ratio
    if params['all_wheels_on_track'] and params['steps'] > 0:
        reward_lane = 1 + (params['progress']/params['steps']*100)
    else:
        reward_lane = 1e-3

    # Penalize if car is too close to the next object
    reward_avoid = 1.0

    # Distance to the next object
    next_object_loc = objects_location[next_object_index]
    distance_closest_object = math.sqrt((params['x'] - next_object_loc[0])**2
+ (params['y'] -
next_object_loc[1])**2)
    # Decide if the agent and the next object is on the same lane
    is_same_lane = objects_left_of_center[next_object_index] ==
is_left_of_center

    if is_same_lane:
        if 0.5 <= distance_closest_object < 0.8:
            reward_avoid *= 0.5
        elif 0.3 <= distance_closest_object < 0.5:
            reward_avoid *= 0.2
        elif distance_closest_object < 0.3:
            reward_avoid = 1e-3 # Likely crashed
#I decided to reward more for avoiding than driving so that the agent would
prioritize the avoidance and preferably still stay inside the track
    reward += 1.5 * reward_lane + 3.0 * reward_avoid

    return reward
```

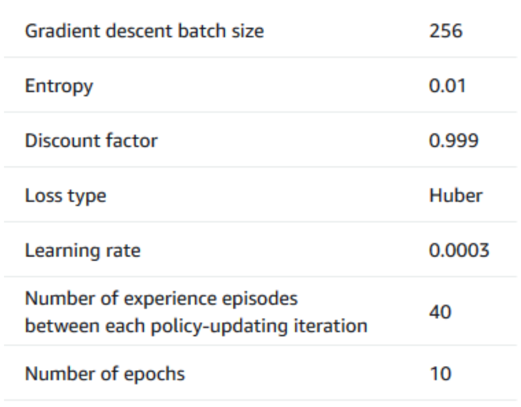
Listing 3. Object avoidance reward function.

Listing 3 shows the reward function. First, it is important to check if the agent is inside the two borders and if steps are larger than 0. The reward is minimal if the agent is not inside the two borders. It is necessary to give the initial reward for avoidance and then calculate the distance toward the next object using pythagoras. Then use a Boolean value to check if the agent and the object are on the same lane. If they are on the same lane, depending on the distance between the object and the agent, it is possible to decrease the reward_avoid by either multiplying the reward_avoid with 0.5 or 0.2 or if reward_avoid is very close to the object, it can be given just a flat minimal reward. After this, it is necessary to add both rewards together for a final reward.

Note: While writing this thesis, it has become quite clear that weights are unbalanced since the reward function rewards heavily for progress, but reward for avoidance is too small, the initial reward being 1. This imbalance should have been taken into account, probably reward ratios not being ideal has impacted the model performance.

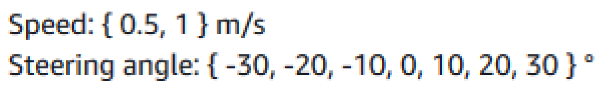
Best model

In object avoidance, there was a lot of problems with the stability and learning efficiency of the models. Most of the problems come with speed of the models and probably the fact that there is significant room for improvement for skills when tuning reward functions, hyperparameters and such. When you have six randomly placed objects, the model has to learn, how to detect an object and dodge it. After trying out various models with higher speeds, it became quite clear that currently is it not possible to build a model with higher top speeds than 1 m/s while trying to avoid six randomly placed objects. That is why the model's top speed is 1 m/s and the model's initial training time is four hours. Figures 35 and 36 show the initial hyperparameters and action space.



Gradient descent batch size	256
Entropy	0.01
Discount factor	0.999
Loss type	Huber
Learning rate	0.0003
Number of experience episodes between each policy-updating iteration	40
Number of epochs	10

Figure 35. Model hyperparameters.

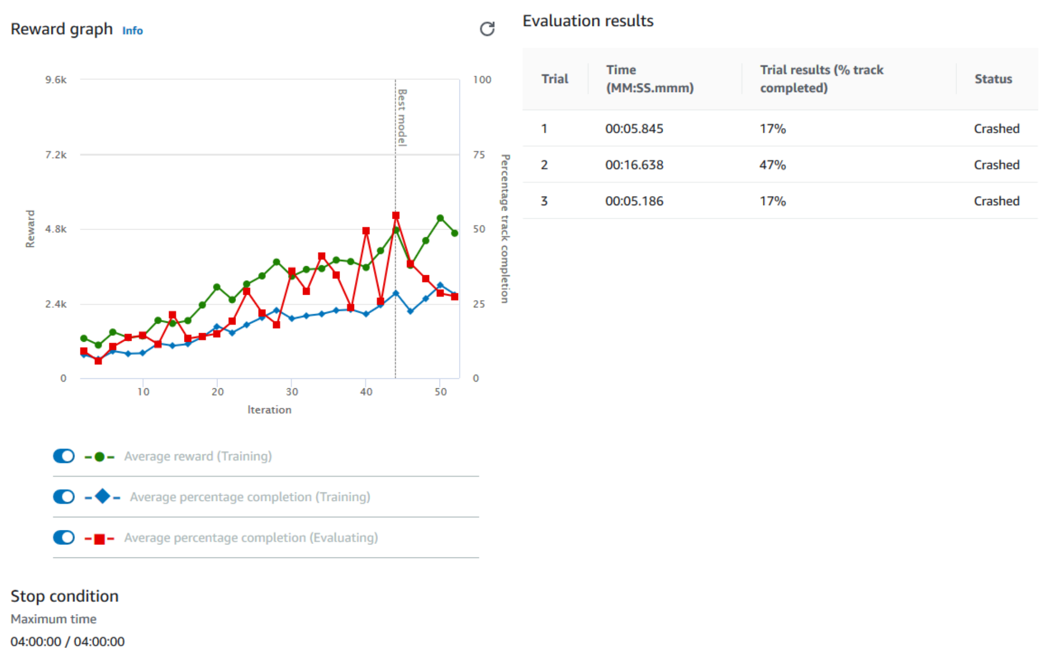


Speed: { 0.5, 1 } m/s
Steering angle: { -30, -20, -10, 0, 10, 20, 30 } °

Figure 36. Model action space

Batch size of 256 was chosen since the previous models had had problems with the stability of the training and increasing the batch size would give more stable updates. The experience episodes between policy-updates of 40 were chosen from the default 20 because the model needed more time to learn its policy since the model needs to be able to avoid six randomly placed objects while still staying on-track.

The model has not yet converged, but the results the model has shown look promising. The model was trained for 4 hours initial training time. A model reward graph and evaluation can be seen in Figure 37. Even though the model cannot complete even a single lap in the evaluation, the reward graph shows promise, but to know better, logs need to be taken a look at. As can be seen from the graphs in Figure 38, the model learns at a good rate and it is already able to complete a few laps within the training session, which is good.


Reward graph
Info
9.6k
7.2k
4.8k
2.4k
0
Reward
100
75
50
25
0
Percentage track completion
Best model
10
20
30
40
50
Iteration
Average reward (Training)
Average percentage completion (Training)
Average percentage completion (Evaluating)

Evaluation results

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:05.845	17%	Crashed
2	00:16.638	47%	Crashed
3	00:05.186	17%	Crashed

Stop condition
Maximum time
04:00:00 / 04:00:00

Figure 37. Model reward graph and evaluation results. Training session 1.

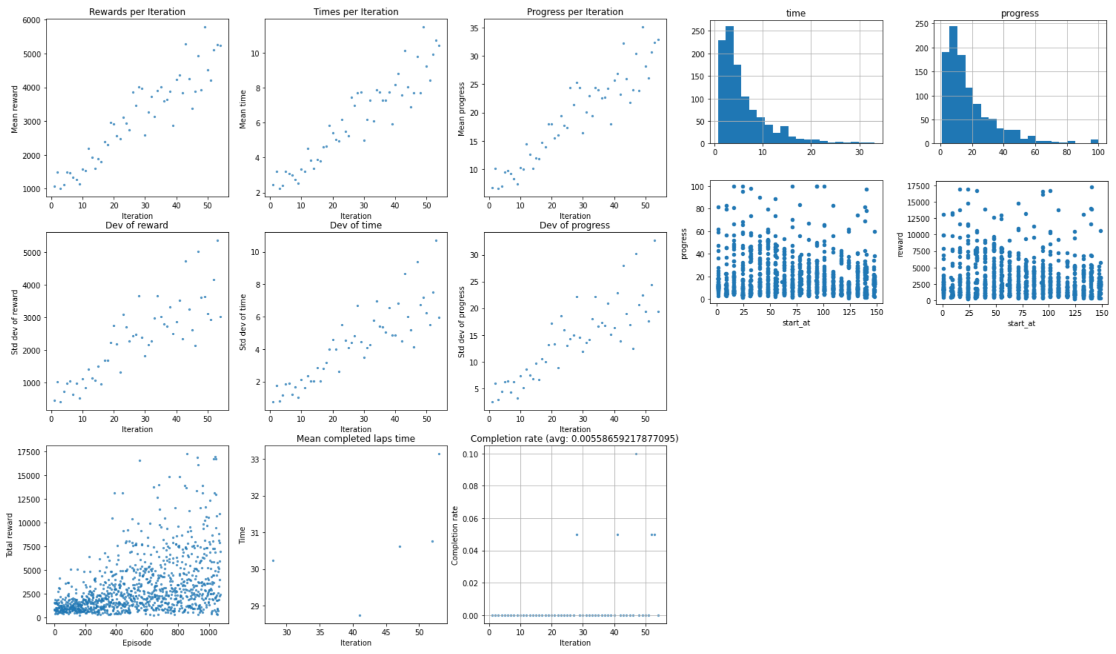
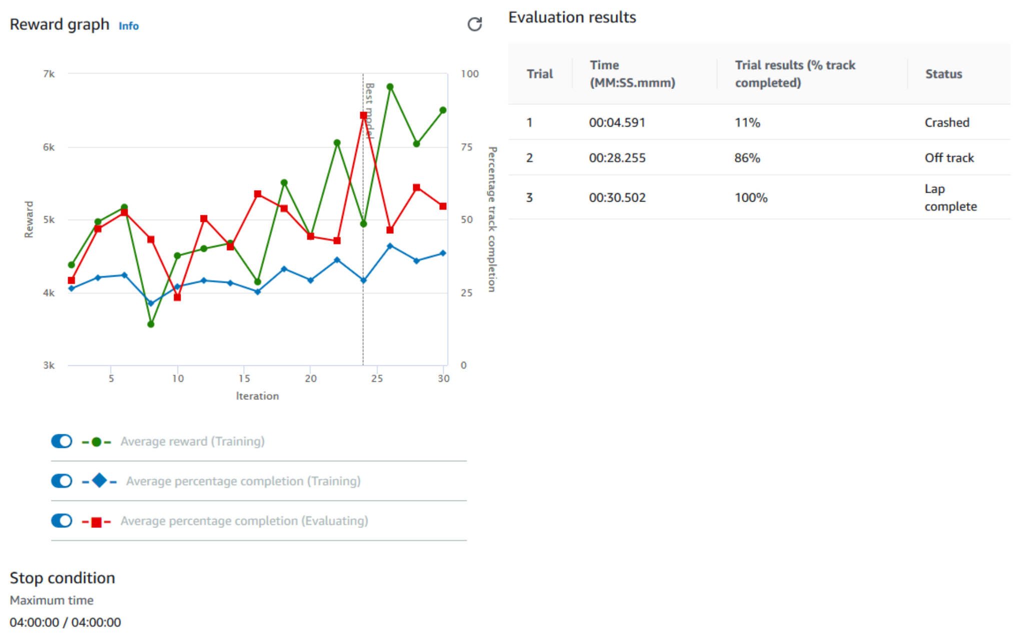


Figure 38. Model training performance graphs. Training session 1.

As can be seen from Figures 37 and 38, progress, rewards, total reward and average progression increase nicely by iteration. The next step is to clone the model and training it for 4 more hours.

After training the model for 4 more hours, the model completed one lap in evaluation (see Figure 39).


Reward graph
Info
Reward
7k
6k
5k
4k
3k
Percentage track completion
100
75
50
25
0
Best model
Iteration
5
10
15
20
25
30
-●- Average reward (Training)
-◆- Average percentage completion (Training)
-■- Average percentage completion (Evaluating)

Evaluation results

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:04.591	11%	Crashed
2	00:28.255	86%	Off track
3	00:30.502	100%	Lap complete

Stop condition
Maximum time
04:00:00 / 04:00:00

Figure 39. Model reward graph and evaluation results. Training session 2.

The learning rate of this training session was reduced from 0.0003 to 0.0001 to keep progress made by iterations tighter. In hindsight, lowering the learning rate by 0.00005 or 0.0001 not 0.0002 would probably have been a better choice, because lowering the learning rate, the model’s learning in this iteration is slower. The graphs still look quite good in terms of the average completion rate, progress and reward growing nicely (see Figure 40).

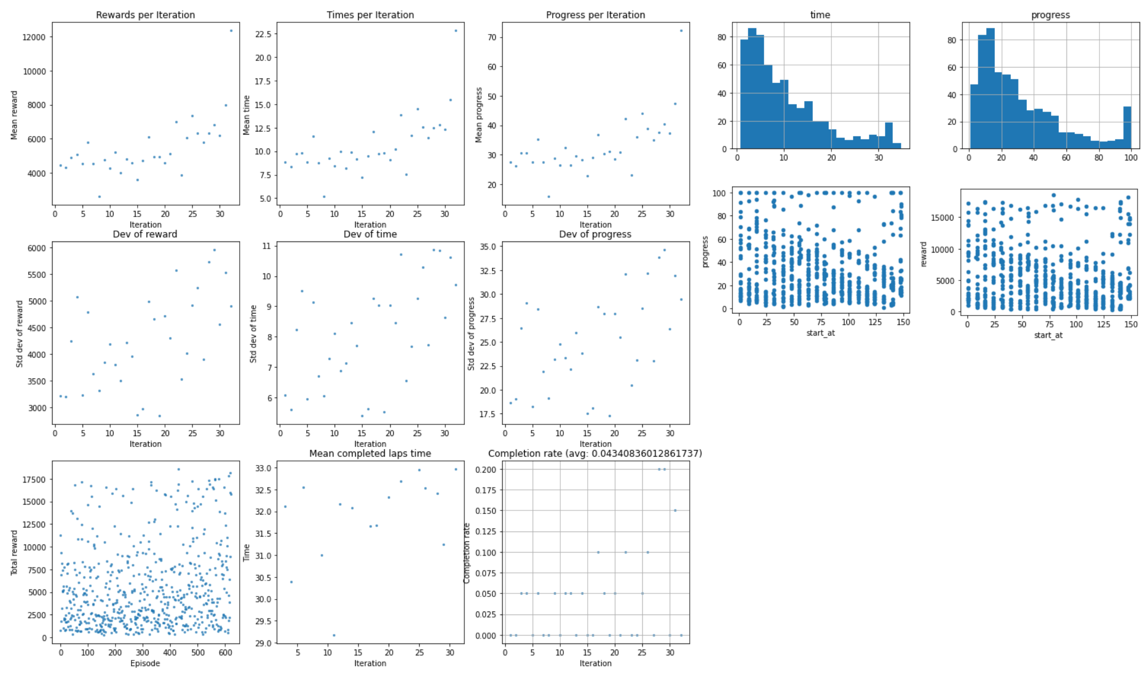
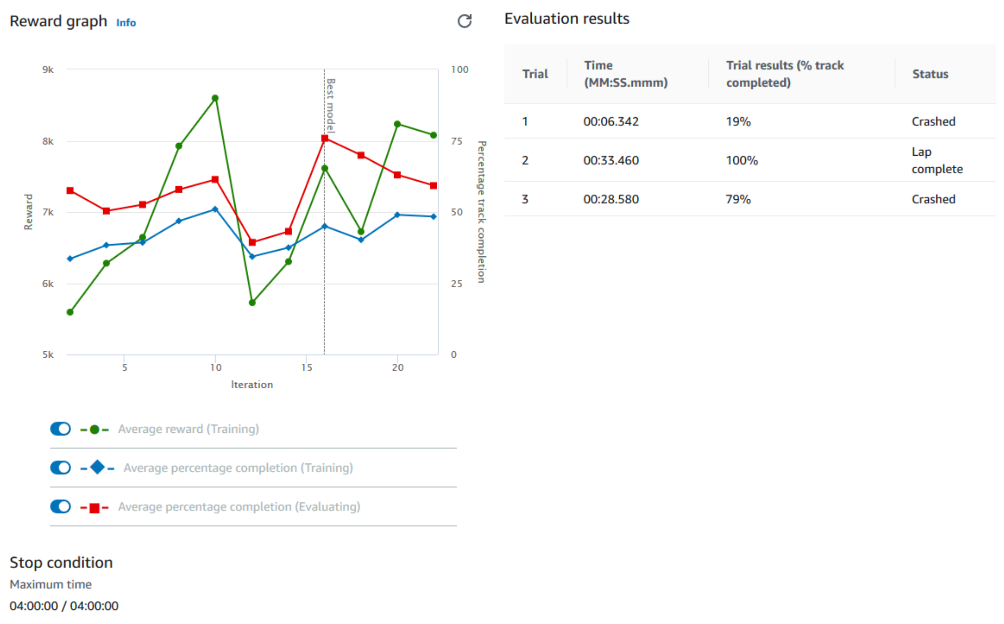


Figure 40. Model training performance graphs. Training session 2.

For next training session, the learning rate is kept at 0.0001. After the third training session, the model completes one out of three laps in evaluation (see Figure 41). A lap was completed around 3 seconds slower, which raises questions. For answers, the graphs in Figure 42 should be taken a look at.


Reward graph Info
Reward
Percentage track completion
Iteration
Best model
Average reward (Training)
Average percentage completion (Training)
Average percentage completion (Evaluating)

Evaluation results

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:06.342	19%	Crashed
2	00:33.460	100%	Lap complete
3	00:28.580	79%	Crashed

Stop condition
Maximum time
04:00:00 / 04:00:00

Figure 41. Model reward graph and evaluation results. Training session 3.

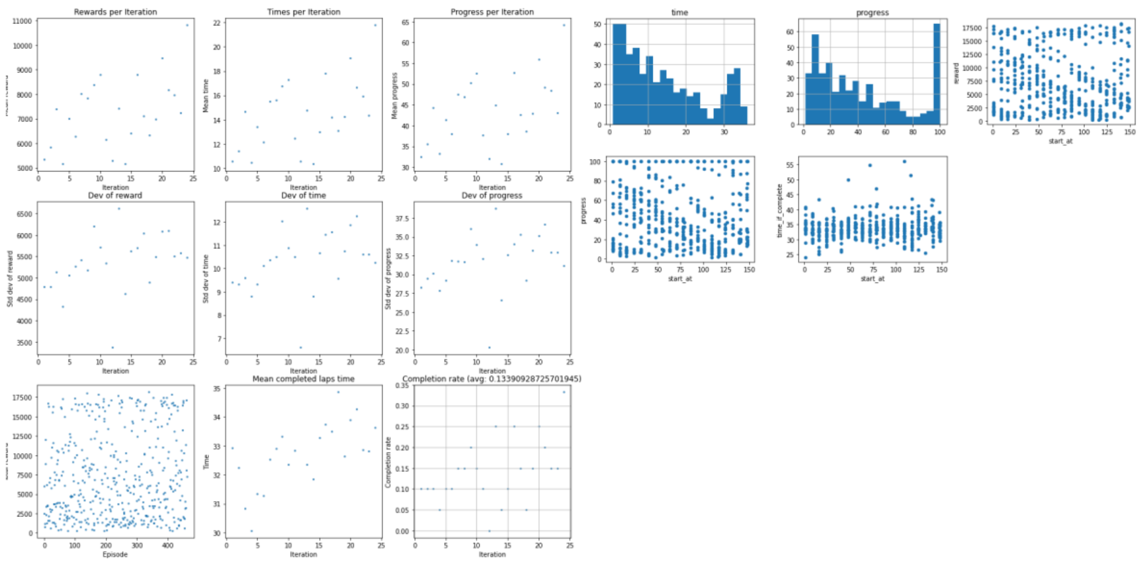


Figure 42. Model training performance graphs. Training session 3.

As the graphs from the third training session show, reward and progress grow over training time. The average completion rate is 9.05 % larger than after the second training session. The model has definitely improved over the training period. There is no question about it. What caused the 3-second increase in lap time was probably caused by deviation in the lap times, as can be seen in Figure 43. In session 2, the completion times are more apart from each other than in session 3. In addition, the completed laps have larger gaps between them in session 2. In session 3, most of the completion times are within 31-34 seconds. In session 2, they are within 31 to 33 seconds. Therefore, there seems to be a small time shift towards slower lap times in session 3 but its completion times are more consistent compared to session 2 which is more spread apart in its completion times as can be seen from the graph. Probably the slower lap should improve over training time.

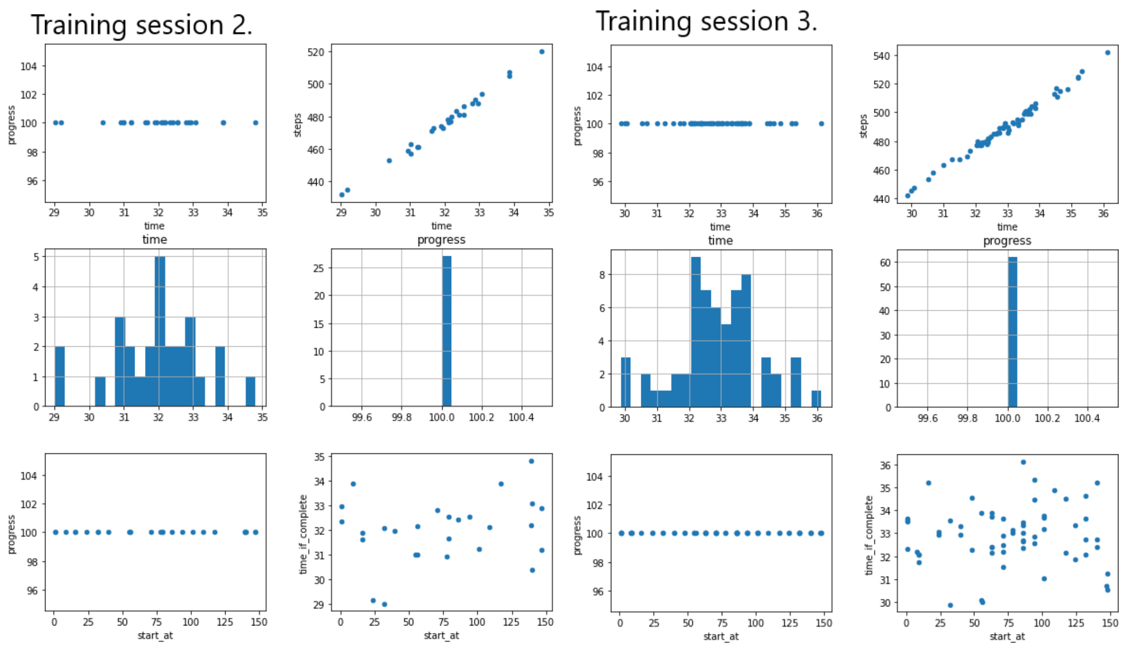
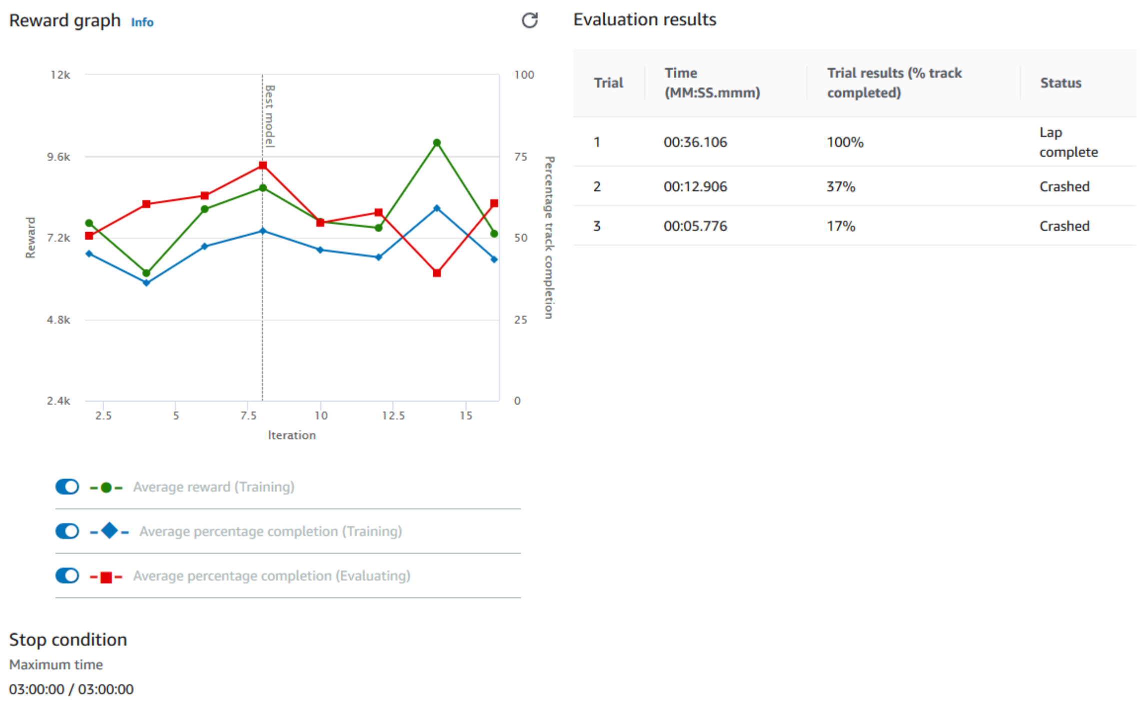


Figure 43. Comparison of lap completion time between training sessions 2 and 3.

For the fourth training session, the learning rate was increased back to 0.0003, which is in fact quite questionable since in most cases, you only want to decrease your learning rate as your training proceeds. Training did not turn out that bad and in evaluation the lap time improved in the one lap the model completed (see figure 44).


Reward graph
Info
12k
9.6k
7.2k
4.8k
2.4k
Reward
100
75
50
25
0
Percentage track completion
Best model
2.5
5
7.5
10
12.5
15
Iteration
Average reward (Training)
Average percentage completion (Training)
Average percentage completion (Evaluating)

Evaluation results

Trial	Time (MM:SS.mmm)	Trial results (% track completed)	Status
1	00:36.106	100%	Lap complete
2	00:12.906	37%	Crashed
3	00:05.776	17%	Crashed

Stop condition
Maximum time
03:00:00 / 03:00:00

Figure 44. Model reward graph and evaluation results. Training session 4.

The increasement of learning rate can clearly be seen in Figure 45 as increased scattering between the rewards, times and progress per iteration. The average completion rate has risen to 15.9 %. Progress from 0 to 50 % starts to even out more which is promising because the model is by then completing more of the track with each training episode than before. This means that the training model moves towards higher stable completion rates and convergence. This is the last training session for this model, but there is ample proof that continuing to train this model will lead to better results. For further training, the learning rate should be lowered.

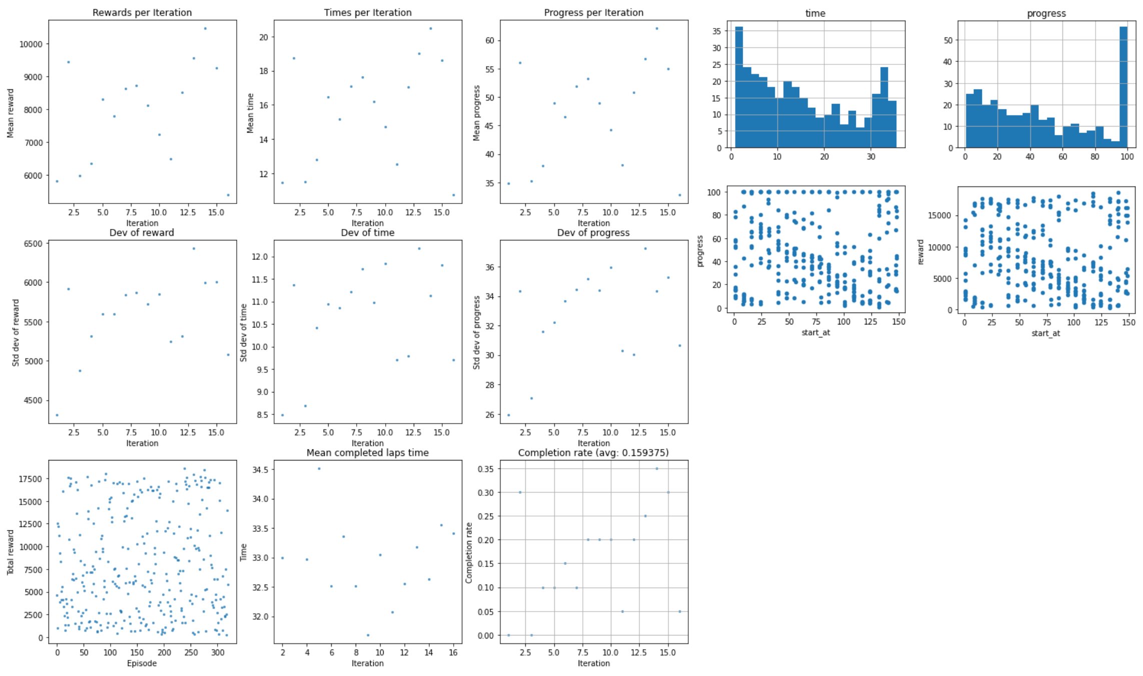


Figure 45. Model training performance graphs. Training session 4.

Figure 46 demonstrates the fastest completed lap time from training session 4. Figure 47 shows the optimized racing line. Concerning this training session in Figure 46, it is important to remember that there are six randomly placed objects in the track meaning that some steering which might be otherwise considered as oversteering is probably just the agent avoiding an object. Still at multiple places at the track, the agent is driving inefficiently because of the unnecessary steering actions. By training the model more, these inefficiencies will at least lessen or disappear entirely.

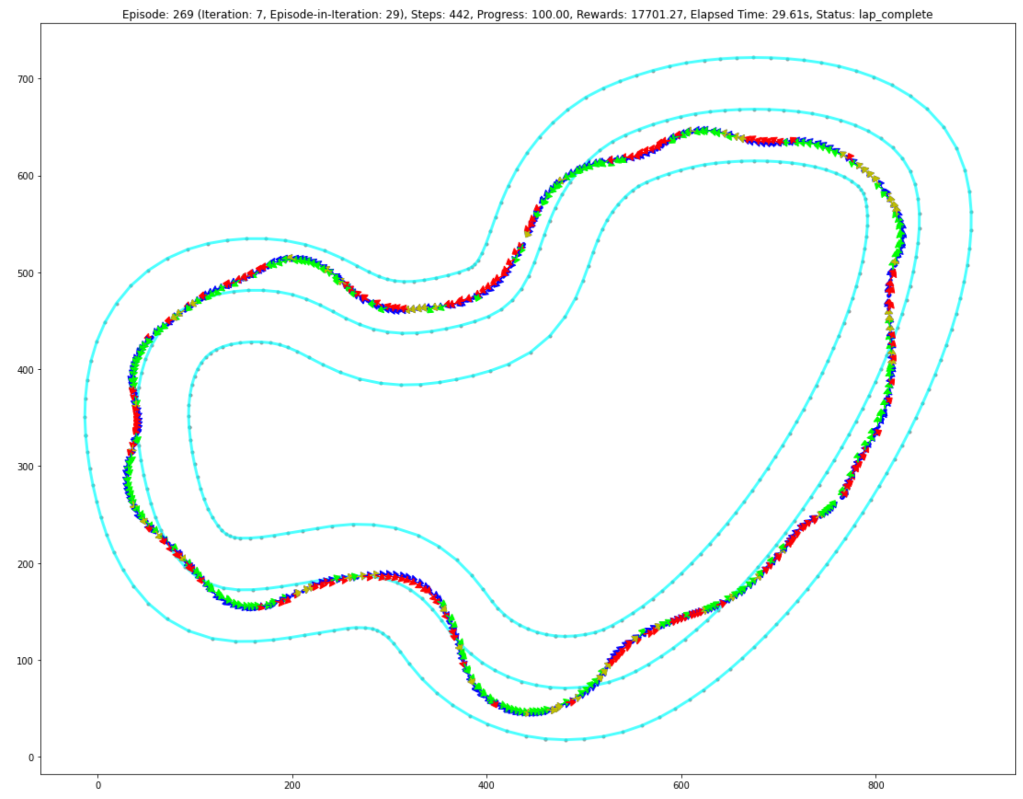
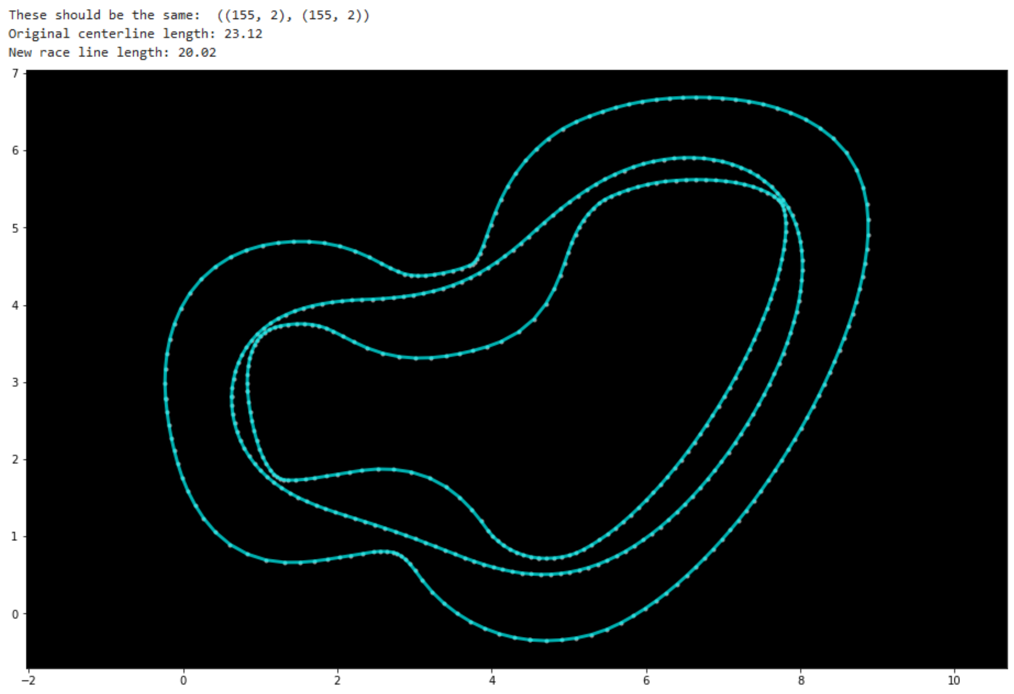


Figure 46. Fastest lap in model's 4th training session. 29.61 seconds.



```
These should be the same:  ((155, 2), (155, 2))
Original centerline length: 23.12
New race line length: 20.02
```

Figure 47. Optimized racing line on the 2019 DeepRacer Championship Cup track.

4.3 Head-to-head racing

In this thesis, not a single model for the head-to-head racing was trained. Head-to-head racing seems to be like object avoidance but, in turn, the objects you must avoid are moving, possibly even at varying speeds. A recommendation cannot be made based on any evidence, but probably reasonable results could be gotten from head-to-head racing with just regular randomly placed object avoidance in mind.

4.4 Hyperparameters and training

Lastly a few words about hyperparameters. In general, if you want more stable training use larger batch sizes but do keep in mind that training is going to be slower. Learning rate tuning over the training is the most important part of the hyperparameter tuning if you want consistent improvement to your model. All in all, you definitely should try tuning various hyperparameters but remember that changing hyperparameters will not save a bad model but with too much tinkering you will ruin a good model for sure.

4.5 What to look for with log analysis

Sections 4.1 and 4.2 discussed the log analysis and what should be paid attention to when doing the log analysis. Preferably you want your model to improve over the training session with smooth updates to progress. What was not mentioned before was that you should also pay attention to how fast you agent goes at various parts of the track and does your agent understeer or oversteer. The faster your agent is, more likely it is to drift or even lose control if the agent’s speed is high and especially when the agent steers too much. Optimising your action space is also a good way to improve your model’s performance.

5 Conclusions

5.1 What was achieved?

The goal of this project was to take a general look into AWS DeepRacer, including training models, evaluating them and analysing their data in order to improve them. The models discussed in this thesis were either time trial or object avoidance models. Head-to-head racing models were not built at all. In this project, estimates have been given about the required training time to know if the model’s performance is good and how much to initially train a model when training with models requiring longer training times. Agent parameters were looked into and data was presented in order to show the significance of the agent’s speed in model stability. How to train models more efficiently was looked at and recommendations were given based on the gathered data. The models log files were also analysed and it was showed which parameters should be taken into account when training a model. As a result, multiple good practises were suggested with evidence from the data. The models discussed in sections 4.1 and 4.2 should have been trained until convergence since a model is as good as it can be proved to be. However, enough evidence was presented that these models can and should be trained for more and that their performance is still improving. All in all, the initial goals of this project were met.

5.2 Next steps

After training and evaluating the models in simulation, the next step would be to test DeepRacer in a physical environment. As briefly mentioned in section 3.2, there is lot of preparation to be done in order to get the physical DeepRacer working. For example, it is necessary to build the track on which DeepRacer will be driven.

5.3 Suggestions for further examination

Since the scope, resources and time for this project were limited, there was not enough time to look at everything. Reward function customization was not looked well into. A possible point of investigation is reward function optimization. If there were already have a very good reward function, the model would converge faster while also learning a better policy. It is very important to reward the model for correct actions.

Another option would be to build a model for the AWS DeepRacer community race.

References

1 Lonza Andrea. 2019. Reinforcement Learning Algorithms with Python. Packt.

2 Chollet Francois. 2018. Deep Learning with Python. Manning Publications Co.

3 Amazon. 2021. AWS DeepRacer: Developer Guide. Amazon Web services.

4 Ali Mousavi, AI Resident and Lihong Li. 2020. Off-Policy Estimation for Infinite-Horizon Reinforcement Learning. Google AI Blog. <https://ai.googleblog.com/2020/04/off-policy-estimation-for-infinite.html> 24 April 2021.

5 Albert Lai. 2019. Reinforcement Learning with AWS DeepRacer. Toward Data Science. <https://towardsdatascience.com/reinforcement-learning-with-aws-deepracer-99b5dd2557c8> 24 April 2021.

6 Amazon Web Services. 2021. AWS DeepRacer Reward Function Examples. Amazon Web Services. <https://docs.aws.amazon.com/deepracer/latest/developerguide/deepracer-reward-function-examples.html> 25 April 2021.

7 O'Brien Tim. 2020. DeepRacer Expert Bootcamp: How Much Training Is Enough? <https://www.youtube.com/watch?v=qOEqW-xTClE> 26 April 2021

8 Ptak Tomasz. 2020. DeepRacer Expert Bootcamp: Through the log & into the AWS DeepRacer's mind with log analysis tools <https://www.youtube.com/watch?v=v7SXfYSWvBU> 26 April 2021

9 OpenAI. 2018. Proximal Policy Optimization. OpenAI. <https://spinningup.openai.com/en/latest/algorithms/ppo.html> 27 April 2021.

10 OpenAI. 2018. Soft Actor-Critic. OpenAI. <https://spinningup.openai.com/en/latest/algorithms/sac.html> 27 April 2021.

11 Will Koehrsen. 2018. Overfitting vs. Underfitting: A Complete Example. Towards data science. <https://towardsdatascience.com/overfitting-vs-underfitting-a-complete-example-d05dd7e19765> 28 April 2021.

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

<https://aws.amazon.com/blogs/machine-learning/using-log-analysis-to-drive-experiments-and-win-the-aws-deepracer-f1-proam-race/>
28 April 2021.

13 Amazon. 2021. Amazon AWS client. Amazon Web Services. <https://console.aws.amazon.com/deepracer/home?region=us-east-1#createModel> 5 May 2021.