**Program: Artificial Intelligence with Machine Learning**
**Course: AIGC-5002 Machine Learning and Deep Learning**

# Project Report: Reinforcement Learning through AWS DeepRacer

**Team: CodeSpeed**
**Nishit Shaileshbhai Rathod – N01586439,**
**Dhyey Chauhan – N01581918,**
**Sai Chandu Yalangandula – N01587694**
**Saurabh – N01584674**

# Reinforcement Learning through AWS DeepRacer

Before we dive more into DeepRacer, we will briefly talk about cloud computing, machine learning, and, most importantly, reinforcement learning. Later, we will explain more about the components of DeepRacer and how it actually works.

## Cloud Computing on AWS

Cloud computing is considered to be one of the hot topics in technology nowadays. The “pay-as-you-go” approach provided by cloud services is a deal-breaker for many individuals, businesses, and corporations. With cloud services, no longer on-premises infrastructures are needed. They can provide on-demand delivery of several computing services through the internet. The type of services could be databases, servers, storage, software, networking and many more that can help businesses scale and expand.

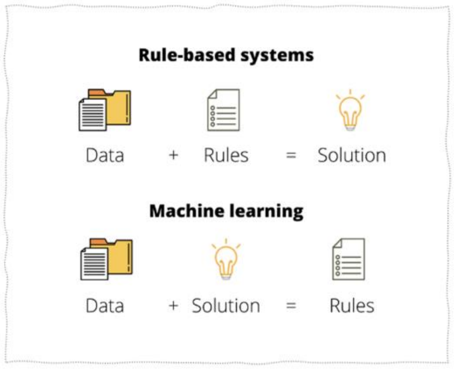
Amazon Web Services (AWS) is leading the industry in providing cloud computing services on different scales. Nowadays, various organizations are using AWS for various use cases such as disaster recovery, data backup, virtual desktop, emails, software development and testing. Have you ever been wondering how Netflix manages to stream more than 100 million hours of content every single day? The answer is using Amazon Kinesis Data Streams on AWS.

One of the important applications on AWS platforms is training and evaluating machine learning models. With services such as SageMaker and RoboMaker, the process became much easier and faster than ever.

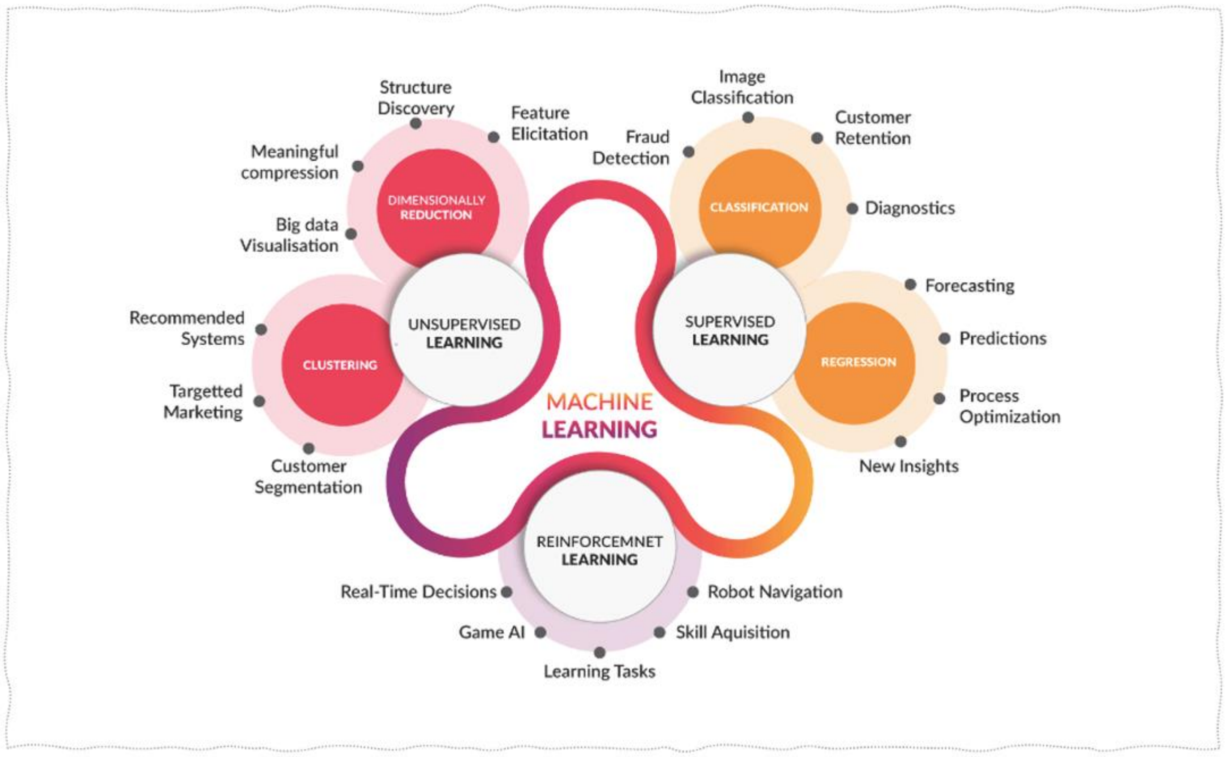
## Machine Learning

Machine learning is one of the most famous and robust technologies nowadays. Its applications can be seen in many aspects of daily life, from predicting and forecasting the weather to the sentimental analysis of social media posts.

Before machine learning, the computers would take rules from humans, process the data according to those rules, then provide the solution. With machine learning, the process is slightly different where the data is given along with the solution. The machine learns the data to build a model that can be used to predict the future. The design begins with gaining domain knowledge and the problem of interest of the research to produce a mathematical model which can capture the physics of the problem under study. The following figure illustrates the difference between manual process and machine learning.



Machine learning covers a wide range of applications that we can see around us every day. We can see it while unlocking our smartphones using face recognition, calling a virtual assistant that can recognize our voices, such as Siri and Alexa, and forecasting and predicting how the weather will be tomorrow afternoon. The following figure shows different types of machine learning models and their applications.



From the figure, there are three main types of machine learning models, supervised, unsupervised, and reinforcement learning. Following, we will explain each type in the context of autonomous cars.

The first type is supervised learning. In this approach, the goal is to build a model that learns the association between inputs and outputs. For example, in autonomous cars, the inputs could be anything from the sensors or navigation information, such as the temperature outside the vehicle, the current speed, the speed limit, school zone, distance from the traffic light, etc. The output could be categorical values such as [speed up, speed down] or a numerical value such as the exact speed. Although this is not the best example of the practical application of supervised learning in autonomous cars, we will explore further machine learning applications later. The drawback of this type of model is that it only knows what was given in data, leading to biased results. Also, it requires a massive amount of data to train the model.

The second type is Unsupervised Learning. In this type of learning, there is only an input feed to the model, and no output is provided. The main goal here is to find a pattern that can show insights that are invisible otherwise. One of the famous approaches is clustering, where objects with similar features are put together. In the case of autonomous cars, unsupervised learning can help identify the colour of the traffic lights in terrible weather conditions where supervised learning fails.

Both supervised, and unsupervised learning needs data to train the model. Therefore, the question comes up:

What if we don’t have enough data to train our model?

## Reinforcement learning

Reinforcement Learning (RL) is a sophisticated type of machine learning for autonomous cars. Unlike other machine learning algorithms, RL does not require any dataset. Its superpower is that it learns very complex behaviours without requiring any labelled training data. It can make short term decisions while optimising for a longer-term goal.

Reinforcement learning works by learning the optimal behaviour in an environment to obtain the maximum reward. To learn the optimal behaviour, the learner must independently discover the sequence of actions that maximise the reward. The rewards are specified in the reward function. It determines the circumstances in which a learner gets a reward or penalty. Therefore, finding the optimal behaviour is a trial-and-error process. For example, let's consider a basic reinforcement learning model for an autonomous car. If the car is less than a meter away from an object, penalise the agent with a certain amount. Otherwise, reward the agent. This way, the agent will learn by trial and error that staying away from objects is better than getting closer without any previous training data.

## Hyperparameters in Reinforcement Learning

In the reinforcement learning model, there are some hyperparameters that can be tuned to produce a better model performance.

One way to improve the model is to tune the hyperparameters according to the track and reward function. By tuning these parameters, we can improve the learning method of the agent. There are seven hyperparameters for the machine learning process in DeepRacer.

### Gradient descent batch size

The batch size is the number of samples passed to the network at once. This controls the number of training samples to work through before updating the underlying deep-learning neural network weights. Random sampling helps reduce correlations inherent in the input data. Using a larger batch size, we can get more stable and smooth updates to the neural network weights, increasing the training time.

### The number of epochs

The number of passes through the training data to update the neural network weights during gradient descent. The training data corresponds to random samples from the experience buffer. Using more epochs allows the agent to get stable updates, but we will need to increase the training time. We can choose the number of epochs depending on the batch size. A smaller number of epochs for a smaller batch size.

### Learning rate

A portion of the new weight can be from the gradient-descent contribution during each update and the rest from the current weight value. The learning rate controls how much a gradient-descent/ascent update contributes to the network weights. We need to use a higher learning rate to include more gradient-descent contributions for faster training. But rewards may not converge if the learning rate is too large. We suggest using a higher learning rate at the first training of a new model to analyse the graph. We can clone the model and use a lower learning rate to converge the reward if the chart looks good.

**Entropy**

The degree of uncertainty is used to determine when to add randomness to the policy distribution. This added uncertainty helps the agent explore the action space more broadly. A more considerable entropy value encourages the agent to explore the action space thoroughly, but this can be less effective for a less complicated environment. This function sometimes contradicts the reward function.

**Discount factor**

A discount factor specifies how much of the future rewards contribute to the expected reward. If the Discount factor value is higher, the agent considers moving farther out of contributions, but it will slow the training. The agent includes rewards from an order of 1000 future steps to make a move for a discount factor of 0.999. The discount factor of 0.99 means the agent will consider rewards from an order of 10 future steps to make a move.

**Loss type**

A good training algorithm should make incremental changes to the agent's strategy. An agent should gradually transit from taking random actions to taking strategic actions to increase its reward. But if the agent makes big changes, the training becomes unstable and will not learn the track. The Huber loss and Mean squared error loss types behave similarly for minor updates. But as the updates become larger, Huber loss takes smaller increments compared to Mean squared error loss. For convergence problems, use the Huber loss type. But for faster training, use the Mean squared error loss type.

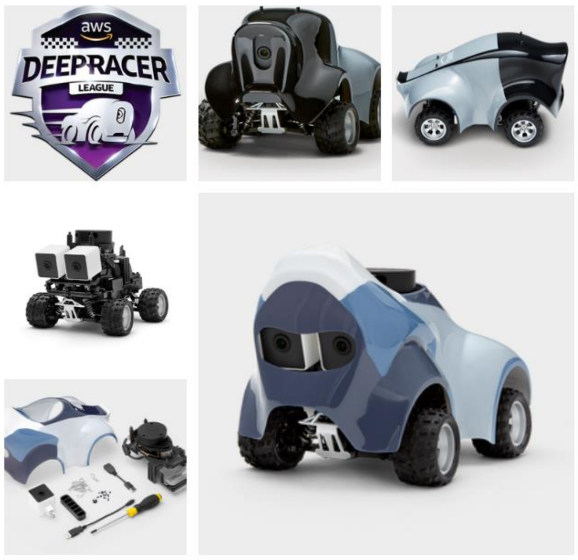
## AWS DeepRacer

With DeepRacer developers of all skill levels can get hands on with machine learning through a cloud-based 3D racing simulator, fully autonomous 1/18th scale race car driven by reinforcement learning, and global racing league.
AWS DeepRacer gives us an interesting and fun way to get started with reinforcement learning (RL). RL is an advanced machine learning (ML) technique that takes a very different approach to training models than other machine learning methods. Its super power is that it learns very complex behaviors without requiring any labelled training data, and can make short term decisions while optimizing for a longer-term goal.

Simulator: Build models in Amazon SageMaker and train, test, and iterate quickly and easily on the track in the AWS DeepRacer 3D racing simulator.

Car: Experience the thrill of the race in the real-world when you deploy your reinforcement learning model onto AWS DeepRacer.

League: Compete in the world's first global, autonomous racing league, to race for prizes and glory and a chance to advance to the Championship Cup.



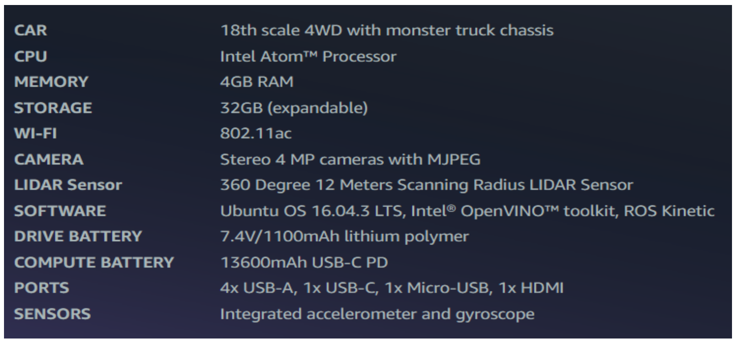
The car has a camera mounted on its head and an onboard computing module. Driven by inference, the compute module follows a track. Dedicated batteries are used for powering the compute module and the vehicle chassis, respectively. In terms of RL, the car is the agent that learns from the environment, while the racing track is the environment.

In the console of AWS, the models can be trained and controlled with a virtual car and tracks. Amazon Web Services (AWS) manages the entire infrastructure, including model training and virtualization of racing circuits, with the AWS console. It also allows developers from all over the world to join the thrill and compete in the league, the world's first autonomous racing league.

AWS DeepRacer comes with a preconfigured cloud environment for training Reinforcement Learning models. It makes use of Amazon SageMaker's new Reinforcement Learning capability, as well as a 3D simulation environment driven by AWS RoboMaker. You can train an autonomous driving model on a set of predefined race tracks included in the simulator. Then evaluate it digitally or download it to an AWS DeepRacer car and test it in the real world.

## DeepRacer Specifications

DeepRacer hardware and software specifications consist of an Intel Atom processor, a 4-megapixel camera with 1080p quality, fast (802.11ac) Wi-Fi, several USB ports, and a battery life of roughly 2 hours. The following figure shows the full specification that lays under the hood in a DeepRacer.



| | |
|---|---|
| CAR | 18th scale 4WD with monster truck chassis |
| CPU | Intel Atom™ Processor |
| MEMORY | 4GB RAM |
| STORAGE | 32GB (expandable) |
| WI-FI | 802.11ac |
| CAMERA | Stereo 4 MP cameras with MJPEG |
| LIDAR Sensor | 360 Degree 12 Meters Scanning Radius LIDAR Sensor |
| SOFTWARE | Ubuntu OS 16.04.3 LTS, Intel® OpenVINO™ toolkit, ROS Kinetic |
| DRIVE BATTERY | 7.4V/1100mAh lithium polymer |
| COMPUTE BATTERY | 13600mAh USB-C PD |
| PORTS | 4x USB-A, 1x USB-C, 1x Micro-USB, 1x HDMI |
| SENSORS | Integrated accelerometer and gyroscope |

## The Reward Function

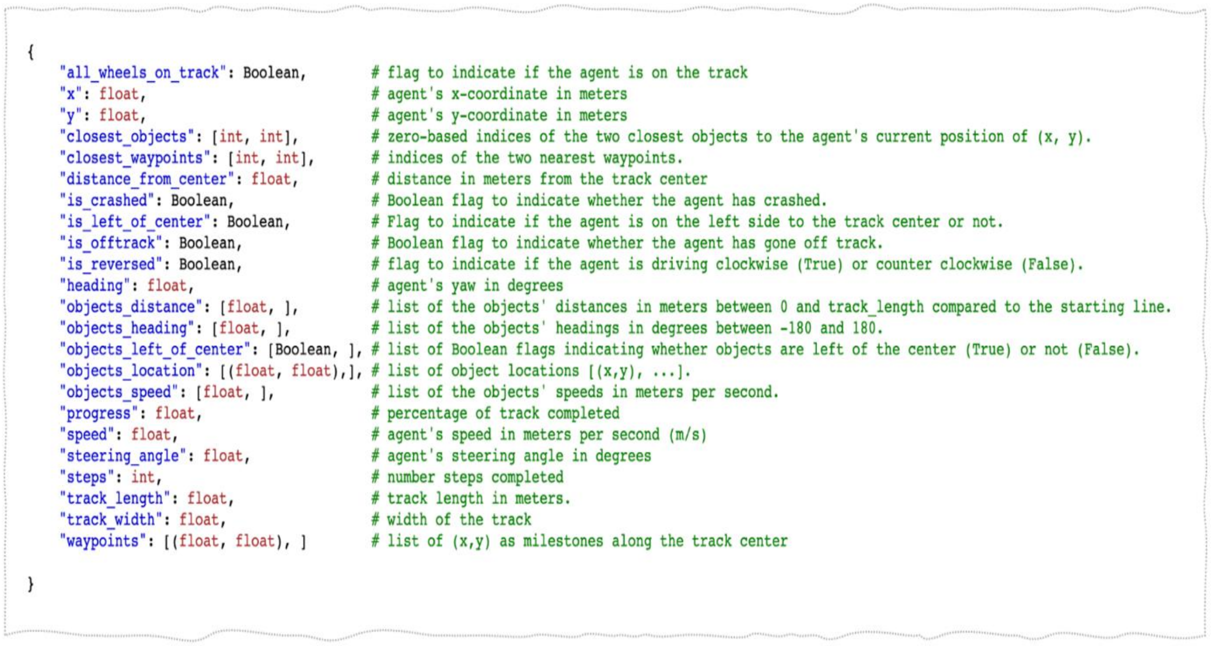
In Reinforcement Learning, reward functions are a programming logic that can help the agent judge a specific action taken during the learning process based on the observed environment. The ultimate goal is to assist the agent in learning which actions contribute to maximising the agent's rewards. Simply put, the reward function is how an agent can know how well or bad it is performing.

The agent starts by driving randomly, forward, zig-zag, or even backward at different speeds when the training begins. It does not know how to get to the end of the track correctly. Here, the reward function will help the car to take the correct path to complete the race. The agent will learn more about the track and maximise the rewards in each iteration by a trial-and-error approach. The final reward is the accumulation of rewards in each step.

The reward function in AWS DeepRacer is essential to optimise and improve the performance of the agent. At a single given state, the agent will evaluate the action it took by the rules stated on the function for a reward or penalty.

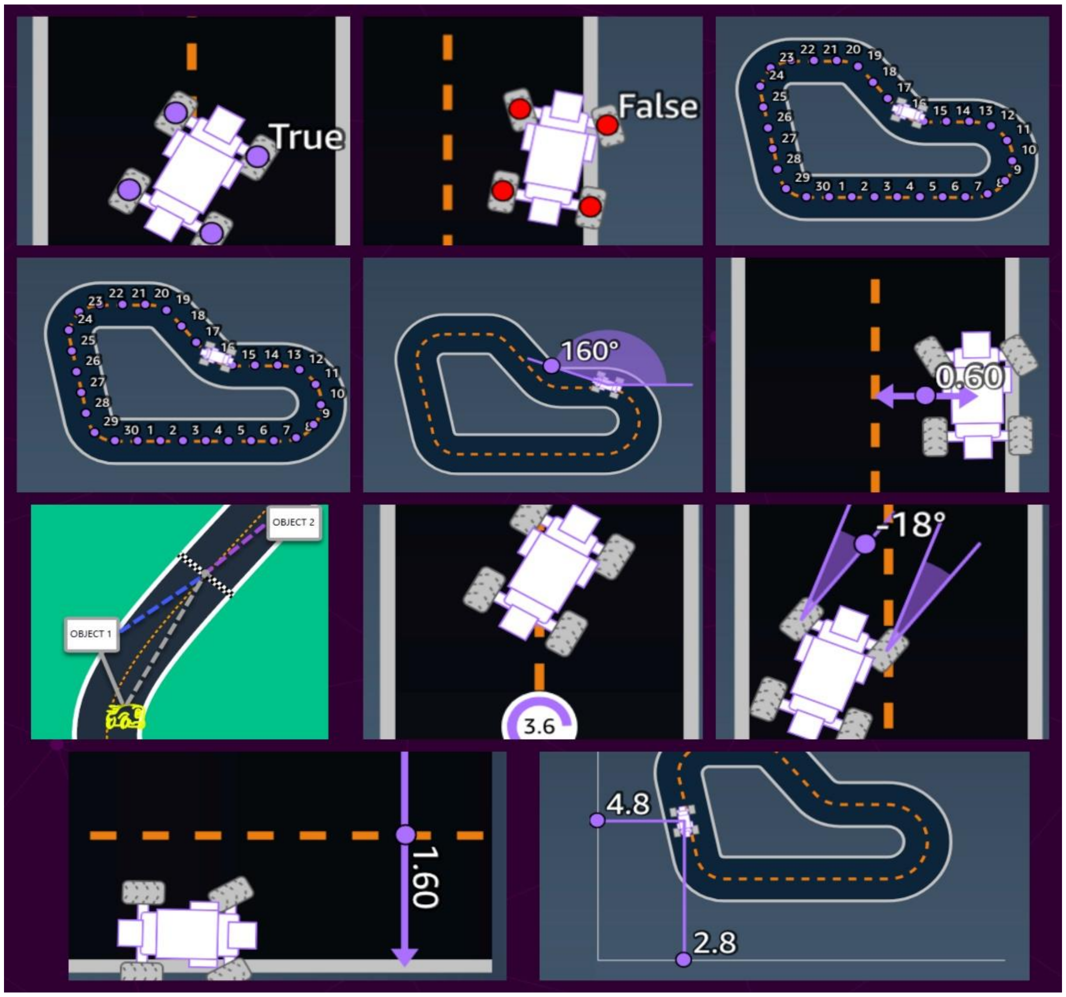
***But how do we specify the rules?***

The AWS DeepRacer reward function takes a dictionary object as the input. This dictionary object contains several parameters that we can use to understand the agent's current state and the environment. The following figure shows all the parameters that we can utilise in our function.

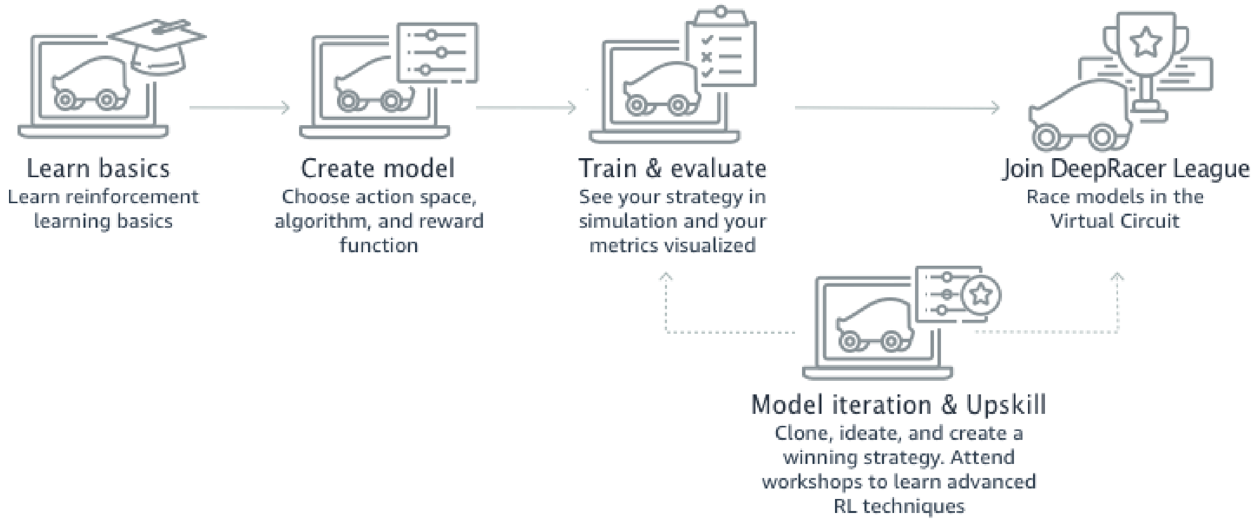


```
{
    "all_wheels_on_track": Boolean,        # flag to indicate if the agent is on the track
    "x": float,                            # agent's x-coordinate in meters
    "y": float,                            # agent's y-coordinate in meters
    "closest_objects": [int, int],         # zero-based 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 compared 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 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

}
```

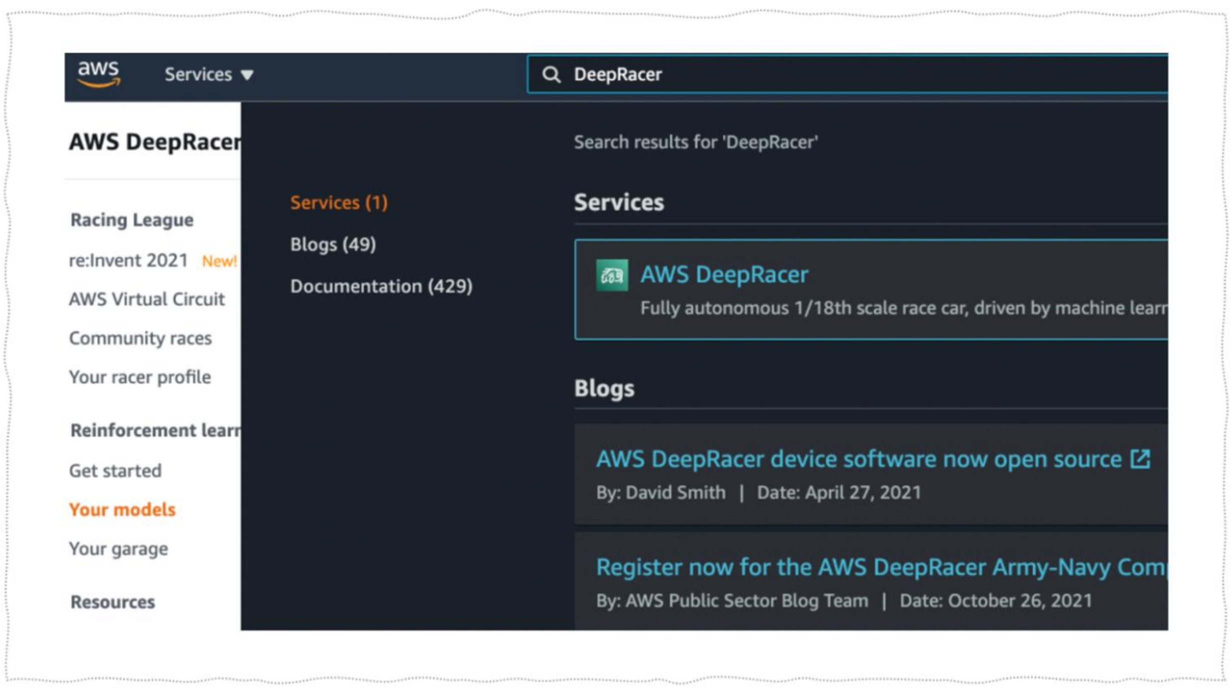


**AWS DeepRacer Process**


Learn basics
Learn reinforcement learning basics
Create model
Choose action space, algorithm, and reward function
Train & evaluate
See your strategy in simulation and your metrics visualized
Join DeepRacer League
Race models in the Virtual Circuit
Model iteration & Upskill
Clone, ideate, and create a winning strategy. Attend workshops to learn advanced RL techniques

**Create a model**

The first step to start with reinforcement learning on DeepRacer is creating a model. To start, we go to the AWS Console and type DeepRacer on the search bar as follows:



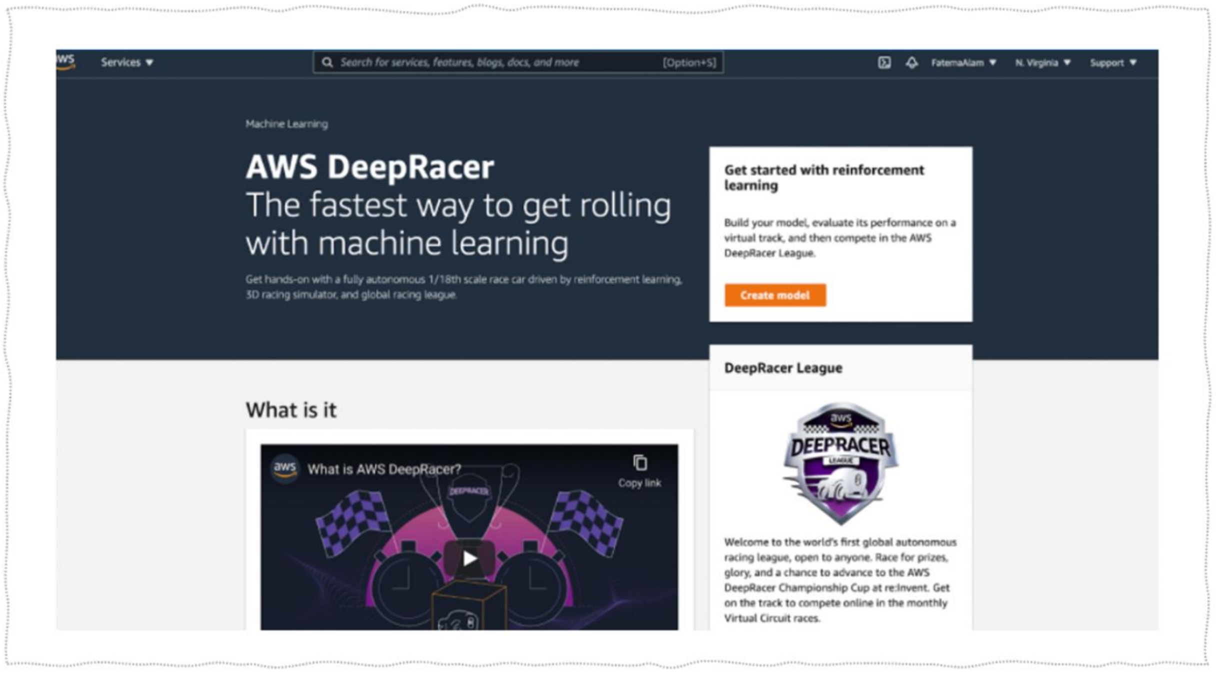
Fully autonomous 1/18th scale race car, driven by machine learn

AWS DeepRacer device software now open source
By: David Smith | Date: April 27, 2021

Register now for the AWS DeepRacer Army-Navy Com
By: AWS Public Sector Blog Team | Date: October 26, 2021

From the DeepRacer console, select “Create model”.

.



**AWS DeepRacer**
The fastest way to get rolling with machine learning

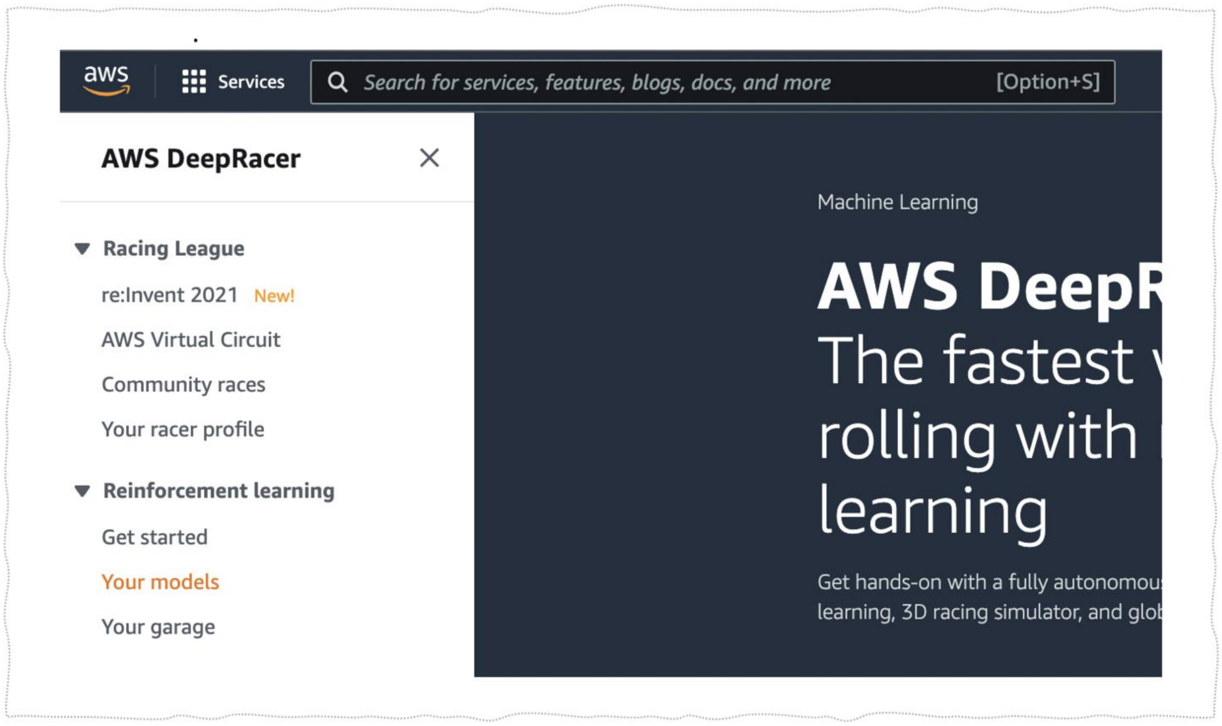
Get hands-on with a fully autonomous 1/18th scale race car driven by reinforcement learning, 3D racing simulator, and global racing league.

Get started with reinforcement learning

Build your model, evaluate its performance on a virtual track, and then compete in the AWS DeepRacer League.

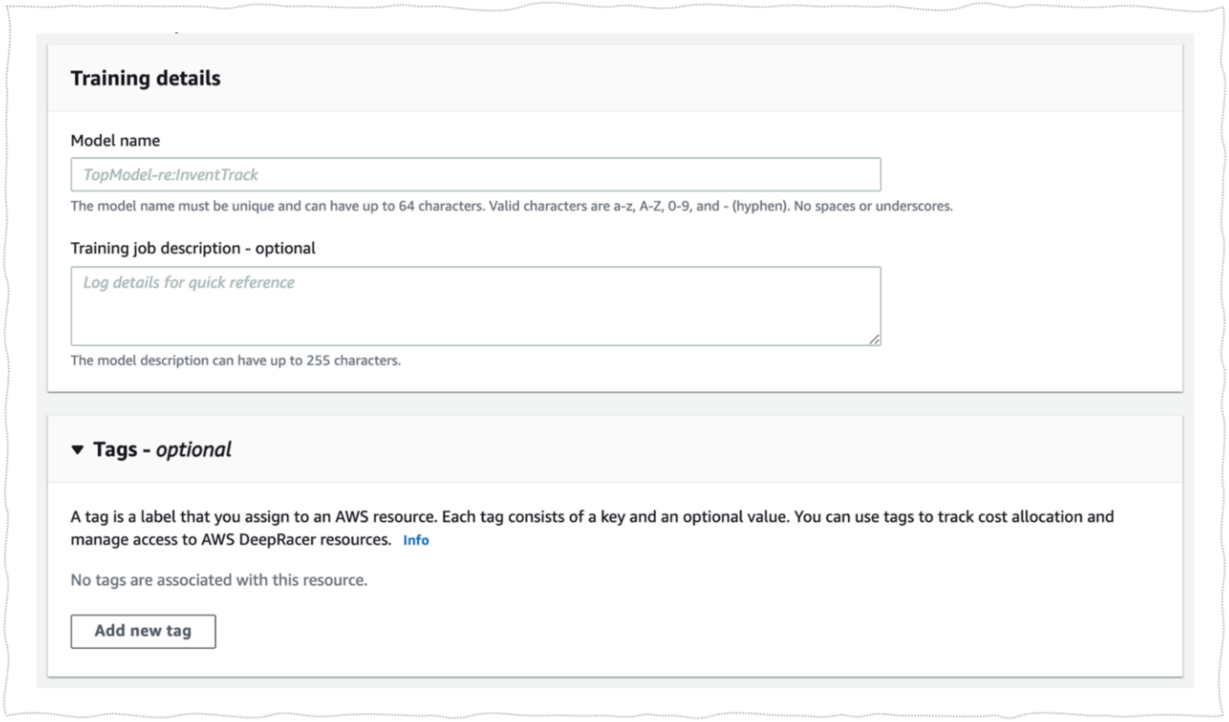
Welcome to the world's first global autonomous racing league, open to anyone. Race for prizes, glory, and a chance to advance to the AWS DeepRacer Championship Cup at re:Invent. Get on the track to compete online in the monthly Virtual Circuit races.

Another option is to use the side menu bar in the DeepRacer console and select "Your mod”, and then select "Create model".



**Step 1: Specifying the model's name and environment**

On the "Create model" page, we have to enter a name for the model under the training details. We can also add training job descriptions, but it is optional. Visit the Tagging page to learn more about tags.


Training details

Model name

TopModel-re:InventTrack

The model name must be unique and can have up to 64 characters. Valid characters are a-z, A-Z, 0-9, and - (hyphen). No spaces or underscores.

Training job description - optional

Log details for quick reference

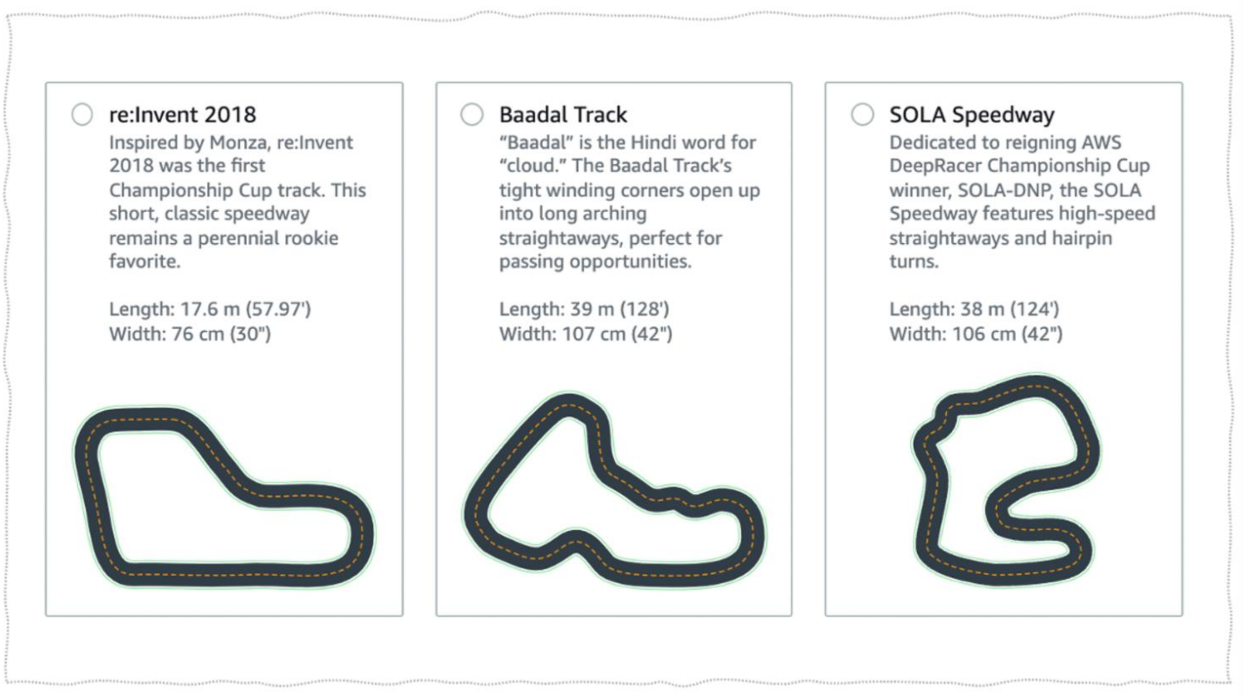
The model description can have up to 255 characters.

Tags - *optional*

A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to track cost allocation and manage access to AWS DeepRacer resources. Info

No tags are associated with this resource.

The next part is to select a racing track as a training environment. A training environment specifies the conditions that the agent will be trained with. There are many shapes of tracks that can be used as a training environment which varies in complexity. As beginners, we can start with a simple track that consisted of basic shapes and smooth turns. We can gradually increase the track complexity when we become more familiar with DeepRacer.



re:Invent 2018
Inspired by Monza, re:Invent 2018 was the first Championship Cup track. This short, classic speedway remains a perennial rookie favorite.

Length: 17.6 m (57.97')
Width: 76 cm (30")

Baadal Track
"Baadal" is the Hindi word for "cloud." The Baadal Track's tight winding corners open up into long arching straightaways, perfect for passing opportunities.

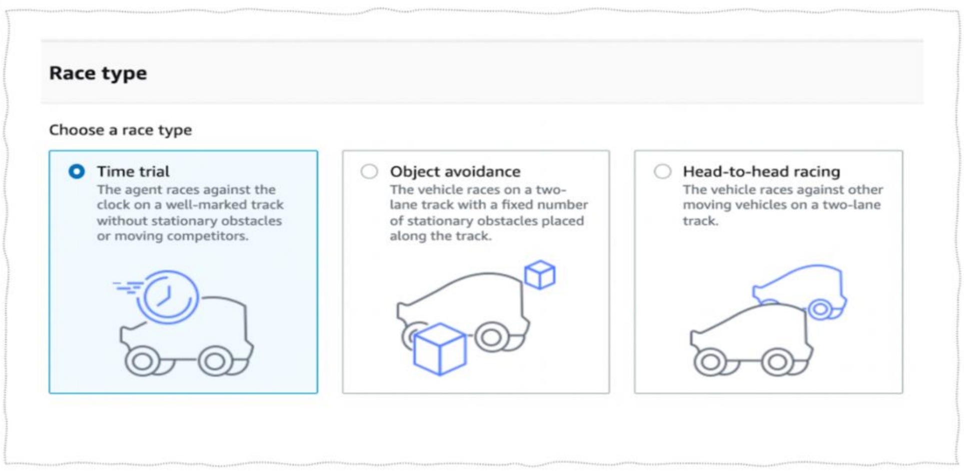
Length: 39 m (128')
Width: 107 cm (42")

SOLA Speedway
Dedicated to reigning AWS DeepRacer Championship Cup winner, SOLA-DNP, the SOLA Speedway features high-speed straightaways and hairpin turns.

Length: 38 m (124')
Width: 106 cm (42")

**Step 2: Choosing race type and training algorithm**

After naming the model, we need to select a race type that we will train the model upon. AWS offers three different types of races. The "Time trial" is the easiest race type where it only considers completing the track in the least amount of time. In the "Object avoidance" race, we aim to complete the track while avoiding random static objects placed along the track. Finally, "Head-to-head" racing is the most challenging where we will face moving objects along the race, which are other players racing on the same track.



Race type

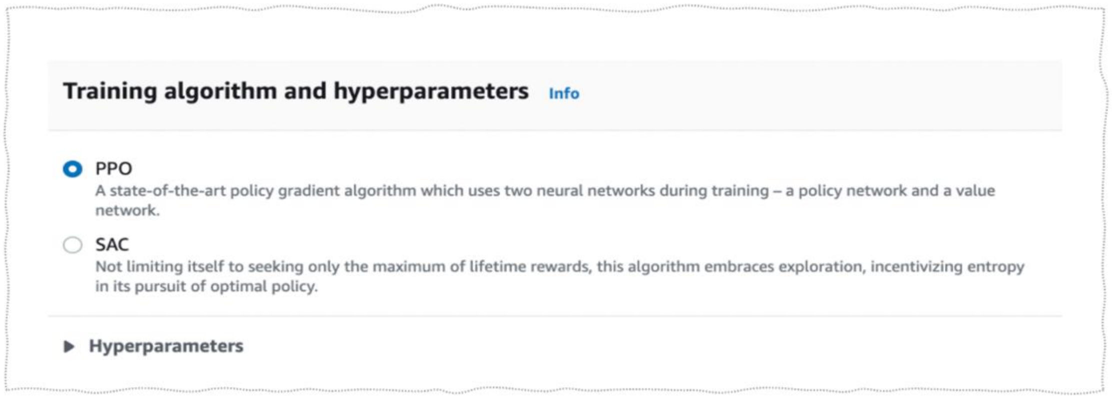
Choose a race type

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.

We will consider the time trail race as the selected option for the remaining instructions. Once the race type is selected, we need to choose the training algorithm. DeepRacer provides two different types of training algorithms, Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). We can select PPO algorithms for continuous or discrete action spaces, while SAC is only for continuous action spaces. You can learn more about the training algorithms from DeepRacer documentation.



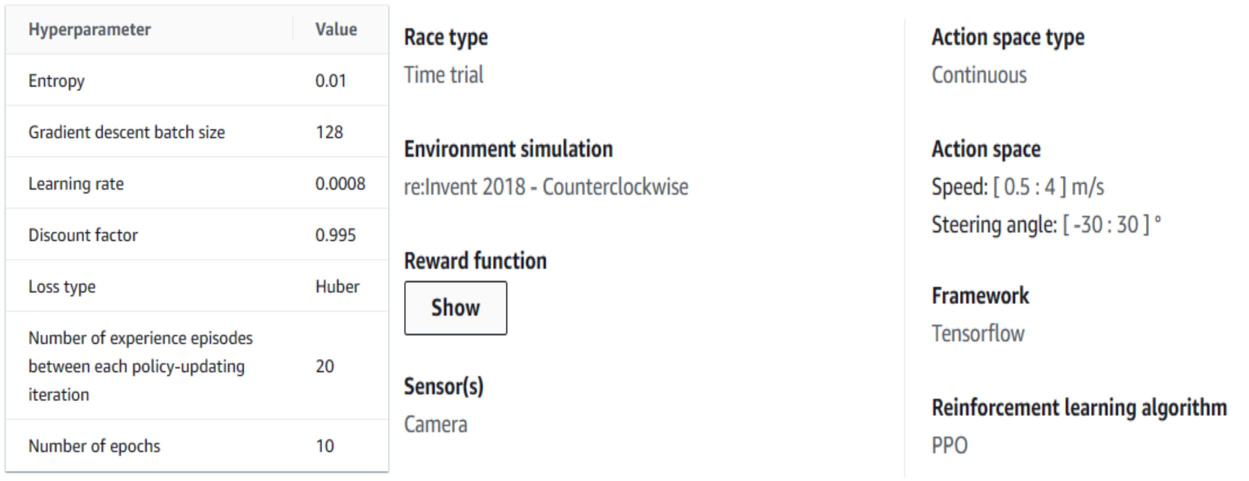
**Training algorithm and hyperparameters** Info

[✓ PPO]
A state-of-the-art policy gradient algorithm which uses two neural networks during training – a policy network and a value network.

SAC
Not limiting itself to seeking only the maximum of lifetime rewards, this algorithm embraces exploration, incentivizing entropy in its pursuit of optimal policy.

▶ Hyperparameters

Besides the training algorithms, we need to select the hyperparameters (see this link for more details about hyperparameters). You can watch this video to learn more about how to choose a proper hyperparameter for our model.



| Hyperparameter | Value |
|---|---|
| Entropy | 0.01 |
| Gradient descent batch size | 128 |
| Learning rate | 0.0008 |
| Discount factor | 0.995 |
| Loss type | Huber |
| Number of experience episodes between each policy-updating iteration | 20 |
| Number of epochs | 10 |

**Race type**
Time trial

**Environment simulation**
re:Invent 2018 - Counterclockwise

**Reward function**
Show

**Sensor(s)**
Camera

**Action space type**
Continuous

**Action space**
Speed: [ 0.5 : 4 ] m/s
Steering angle: [ -30 : 30 ] °

**Framework**
Tensorflow

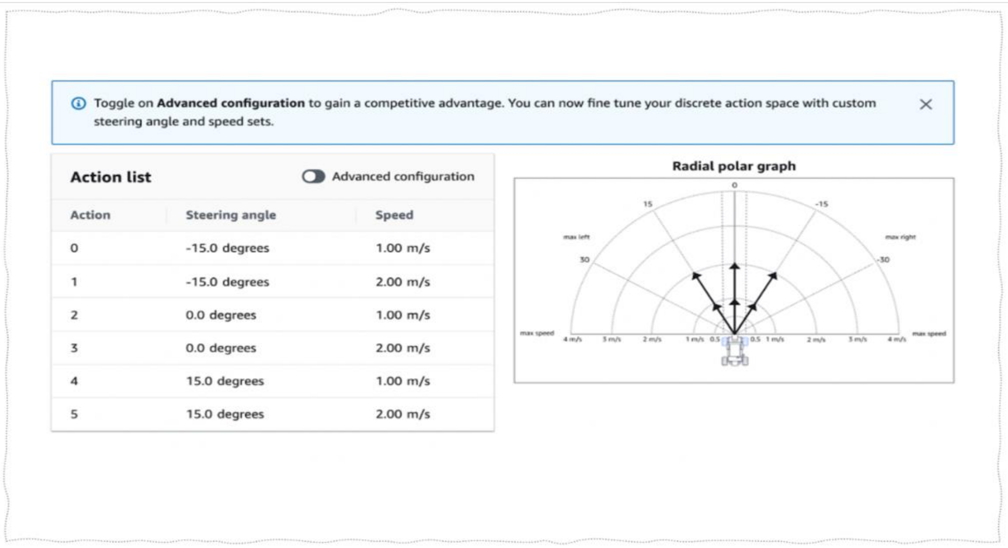
**Reinforcement learning algorithm**
PPO

The reinforcement learning algorithm used in our model is PPO as mentioned earlier. We have tuned the hyperparameters in a different way than the default one. It helped us explore the result for changes in each hyperparameter. The batch size was changed to 128 to get a smoother and more stable update to the neural network.

We have kept the default value for the entropy and reduced the discount factor. We have used Huber as a loss-type since it is more subtle. We have kept the other parameters the same as default except for the learning rate. Since we have increased the batch size, the learning time has also increased. So, we increased the learning rate to shorten the learning time.

**Step 3: Defining Action Space**

Action space specifies the actions the agent can take inside the environment. For DeepRacer, it means in the range of the steering angle and speed of the vehicle. There are two types of action spaces, discrete and continuous. The discrete action space allows us to select a discrete number of angles that the agent can steer and speeds the car can aim for at a single state. The final action space is the steering angles, and speed of every possible action based on the values specified earlier. The following figure shows six possible actions that the agent can take at a single state of the car.



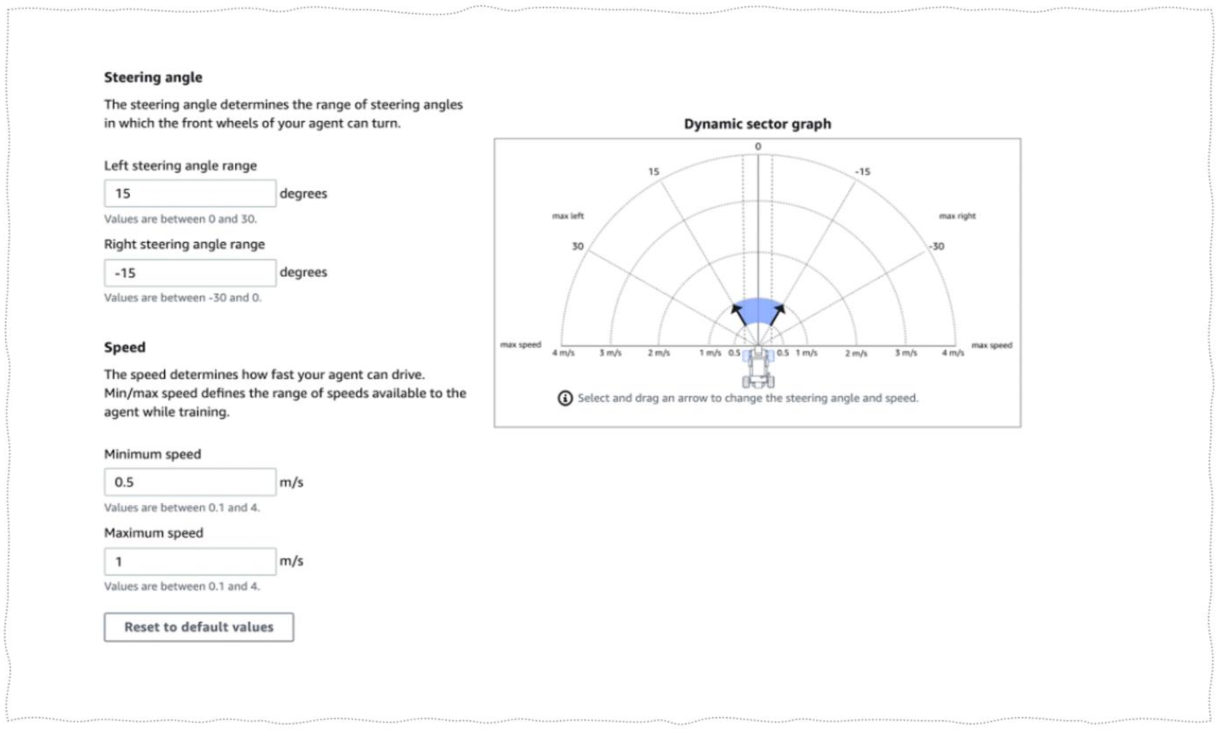
Toggle on **Advanced configuration** to gain a competitive advantage. You can now fine tune your discrete action space with custom steering angle and speed sets.

**Action list** Advanced configuration

| Action | Steering angle | Speed |
|---|---|---|
| 0 | -15.0 degrees | 1.00 m/s |
| 1 | -15.0 degrees | 2.00 m/s |
| 2 | 0.0 degrees | 1.00 m/s |
| 3 | 0.0 degrees | 2.00 m/s |
| 4 | 15.0 degrees | 1.00 m/s |
| 5 | 15.0 degrees | 2.00 m/s |

**Radial polar graph**

Unlike discrete action space, continuous action space does not have a discrete number of actions. We can only specify the minimum and maximum for both the steering angle and the speed. The following figure shows how does the continuous action space look by default.



**Steering angle**

The steering angle determines the range of steering angles in which the front wheels of your agent can turn.

Left steering angle range

15 degrees

Values are between 0 and 30.

Right steering angle range

-15 degrees

Values are between -30 and 0.

**Speed**

The speed determines how fast your agent can drive. Min/max speed defines the range of speeds available to the agent while training.

Minimum speed

0.5 m/s

Values are between 0.1 and 4.

Maximum speed

1 m/s

Values are between 0.1 and 4.

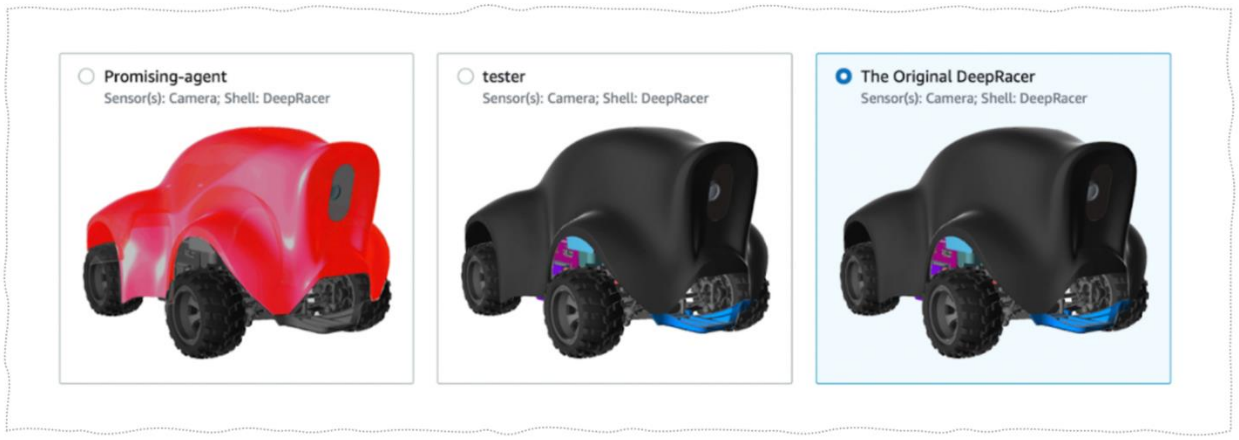
Reset to default values

**Dynamic sector graph**

Select and drag an arrow to change the steering angle and speed.

**Step 4: Choose vehicle**

Here we can specify the vehicle shell and sensor configuration that will run on the track. DeepRacer provides two agents by default with different specifications. We can also create our own custom agents and set the configuration, such as the camera type and adding the LIDAR sensor. The original DeepRacer vehicle was selected for the model, as shown in the following figure.



Promising-agent
Sensor(s): Camera; Shell: DeepRacer

tester
Sensor(s): Camera; Shell: DeepRacer

[✓ The Original DeepRacer]
Sensor(s): Camera; Shell: DeepRacer

**Step 5: Customising reward function**

The last step in creating a model is to choose a reward function and training time. AWS provides some simple examples of reward functions. We can either select a pre-existing reward function from the examples and modify it or create our own reward function from scratch.

The reward function we coded checks three points to consider giving a reward to the model. These points are specified as three steps:

**Step 5.1: Distance from the centre**

The reward function checks the agent's distance from the centre with five markers. Depending on the position of the agent's distance from the centre, the reward will vary. The closer to the centre, the more the reward. We have used five markers to motivate the agent to collect the reward points as well as more options to finish the track faster.

```
# Calculate 5 marks father away from the center line

marker_1 = 0.1 * track_width

marker_2 = 0.20 * track_width

marker_3 = 0.30 * track_width

marker_4 = 0.40 * track_width

marker_5 = 0.5 * track_width
```

**Step 5.2: All Wheels on Track**

We have used this checkpoint to make sure the agents always stay on track. As shown in the figure below, we combined these checkpoints using "if-else" statements.

```
# Give higher reward if the car is closer to center line
if distance_from_center <= marker_1 and all_wheels_on_track:
    reward = 3.0
elif distance_from_center <= marker_2 and all_wheels_on_track:
    reward = 2.5
elif distance_from_center <= marker_3 and all_wheels_on_track:
    reward = 1.5
elif distance_from_center <= marker_4 and all_wheels_on_track:
    reward = 1
elif distance_from_center <= marker_5 and all_wheels_on_track:
    reward = 0.5
else:
    reward = 1e-3  # likely crashed/ close to off track
```

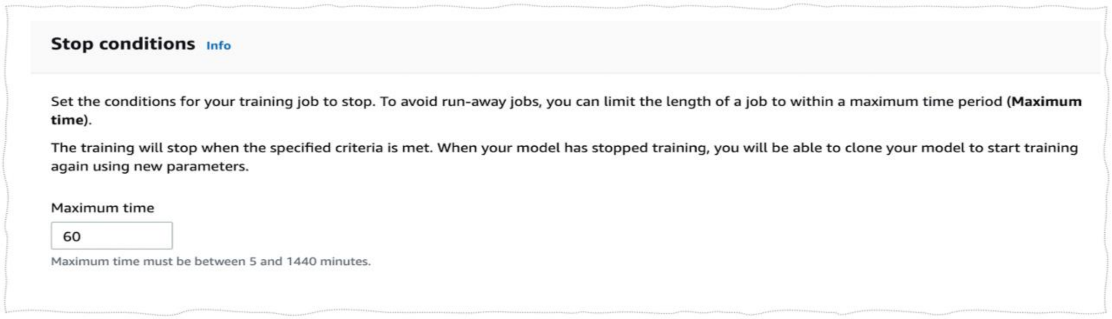
**Step 3: Reasonable Speed Threshold**

This condition checks whether the agent is speeding properly. We have used a speed threshold value of 1, which means if the speed is slow, it will get less reward than higher speeds.

```
if speed < SPEED_THRESHOLD:
    # Penalize if the car goes too slow
    reward = reward + 0.5
else:
    # High reward if the car stays on track and goes fast
    reward = reward + 1.0

return float(reward)
```

Finally, depending on the track and reward function, we need to choose a perfect training time.



**Stop conditions** Info

Set the conditions for your training job to stop. To avoid run-away jobs, you can limit the length of a job to within a maximum time period (**Maximum time**).

The training will stop when the specified criteria is met. When your model has stopped training, you will be able to clone your model to start training again using new parameters.

Maximum time

60

Maximum time must be between 5 and 1440 minutes.

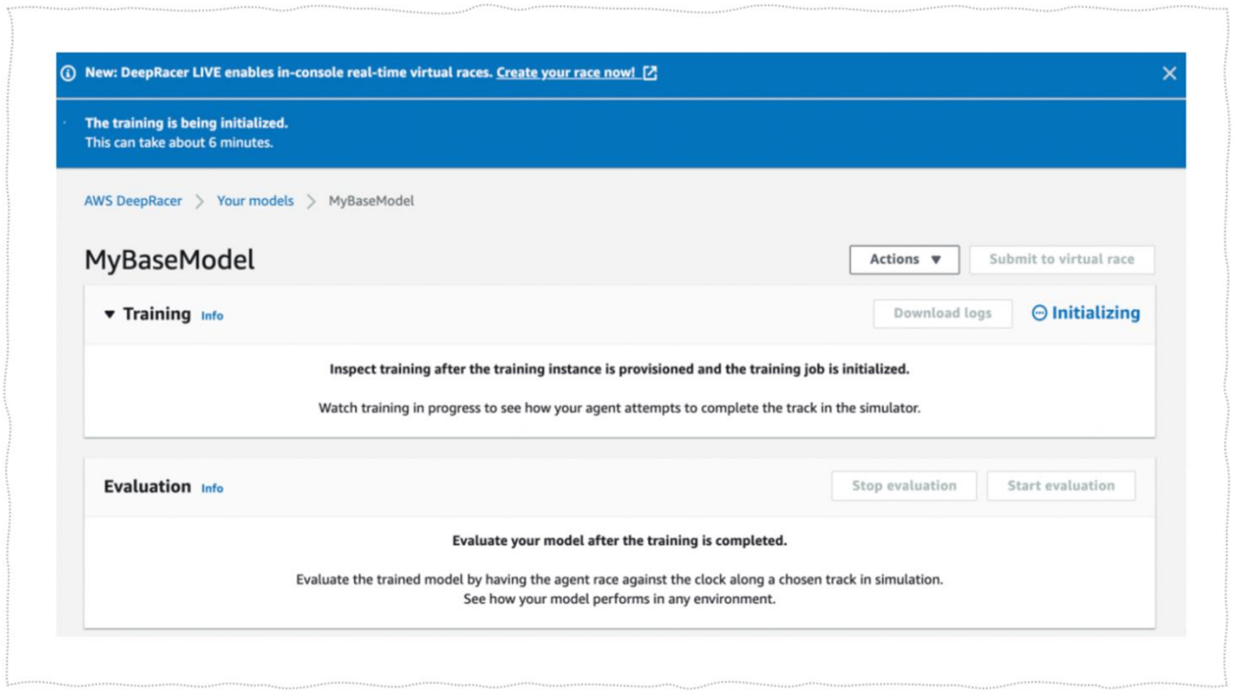
Also, we select the maximum time we want our model to train. It helps to monitor the cost of the training by setting a stopping time. Usually, the more a model is trained, the better it performs. But it can also cause overfitting where the model is trained in a level that can not be generalized in testing environments. Therefore, selecting the proper training time is crucial for the model's performance.

Finally, before we press on "create model", there is an option that allows submitting the model in the DeepRacer league automatically after completion of training. After confirming the choices, we can click the create model. AWS will start building the model and start the training process for the selected amount of time.

**Training**

AWS DeepRacer utilises SageMaker to train the model behind the scenes and leverages  RoboMaker to simulate the agent's interaction with the environment.

Once you've submitted your training task, please wait for it to be initialised and then executed. To change the state from "Initialising" to "In Progress", the initialisation process takes roughly 6 minutes.



New: DeepRacer LIVE enables in-console real-time virtual races. Create your race now!

The training is being initialized.
This can take about 6 minutes.

AWS DeepRacer > Your models > MyBaseModel

MyBaseModel

Inspect training after the training instance is provisioned and the training job is initialized.

Watch training in progress to see how your agent attempts to complete the track in the simulator.

Evaluate your model after the training is completed.

Evaluate the trained model by having the agent race against the clock along a chosen track in simulation.
See how your model performs in any environment.

To track the progress of your training job, look at the Reward graph and Simulation video stream.

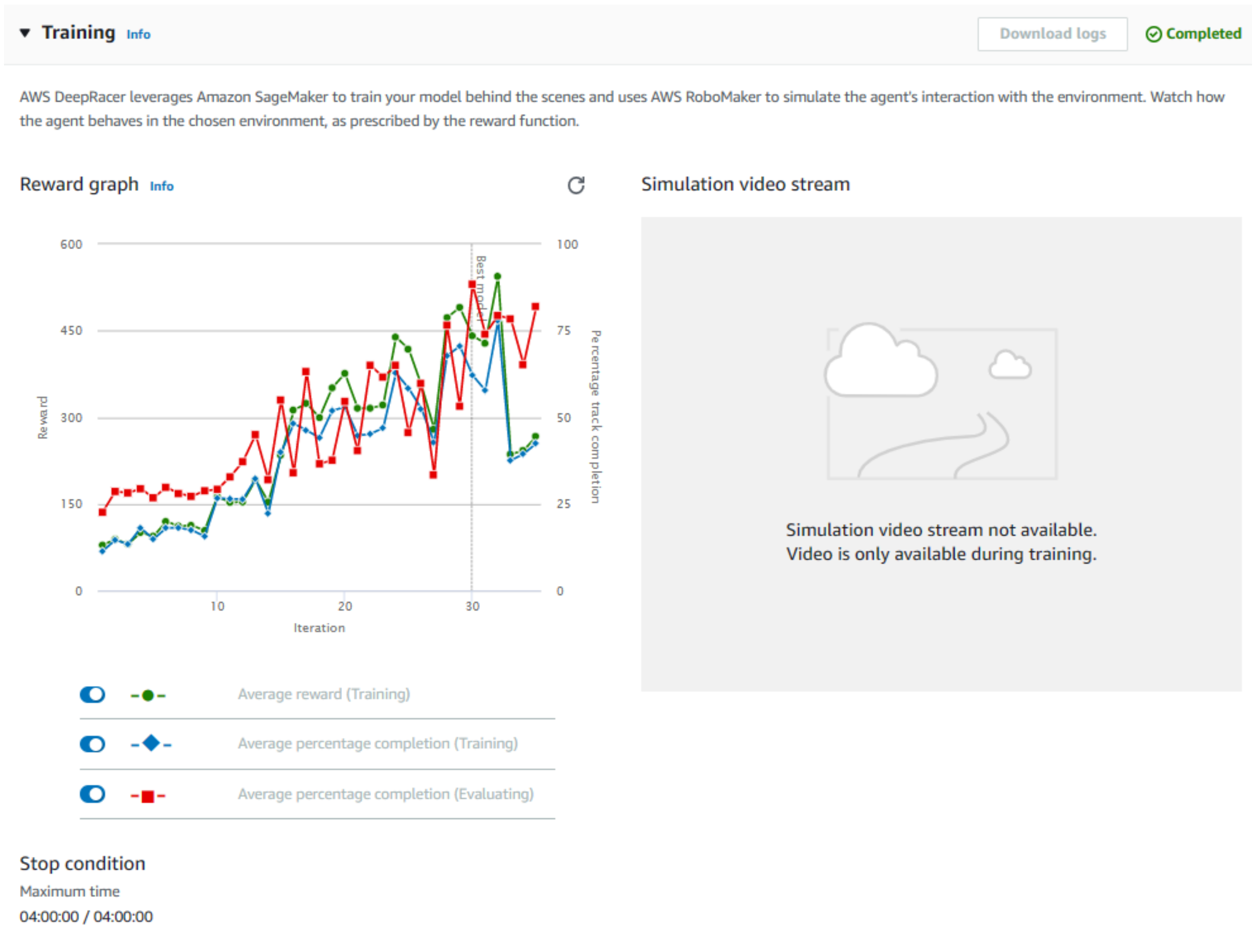
You can refresh the Reward graph by using the refresh button next to it until the training job is completed. In the chart generated, it shows three lines:

- Average reward
- Average percentage completion (Training)
- Average percentage completion (Evaluating)

Each of these metrics is an indicator of how the model is performing during the training phase.

The video shows a training demo where it shows how the model is learning over time by trial-and-error at each run.

**Reward graph**


▼ Training Info
Download logs
Completed
Reward graph Info
Simulation video stream
Reward
Percentage track completion
Iteration
Best model
600
450
300
150
0
100
75
50
25
10
20
30
-●- Average reward (Training)
-◆- Average percentage completion (Training)
-■- Average percentage completion (Evaluating)

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

Simulation video stream not available.
Video is only available during training.

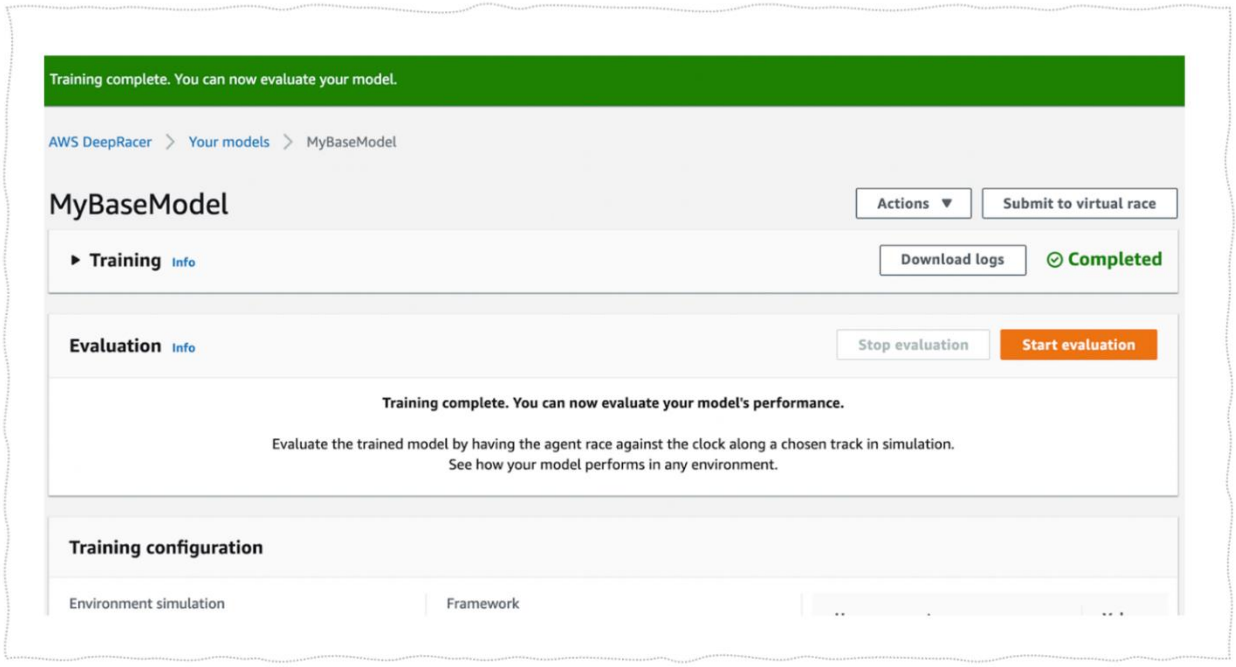
Stop condition
Maximum time
04:00:00 / 04:00:00

**Best Model after Training for 4 hours.**
Average reward (Training) = 441.1
Average percentage completion (Training) = 62.15
Average percentage completion (Evaluating) = 88.4
Still it is seen that there is much scope of improving the model by setting the hyperparameters, reward function, speed and angles of the car locomotion.

**Evaluation results**

Once the training gets completed, it allows you to evaluate the model. When the training task is finished, a model is ready. If the training isn't finished, the model can be in a Ready state if trained up to the point of failure.

**Step 1:** Click on Start evaluation as shown below.


Training complete. You can now evaluate your model.

AWS DeepRacer > Your models > MyBaseModel

MyBaseModel

▸ Training Info ✓ Completed

Evaluation Info

Training complete. You can now evaluate your model's performance.

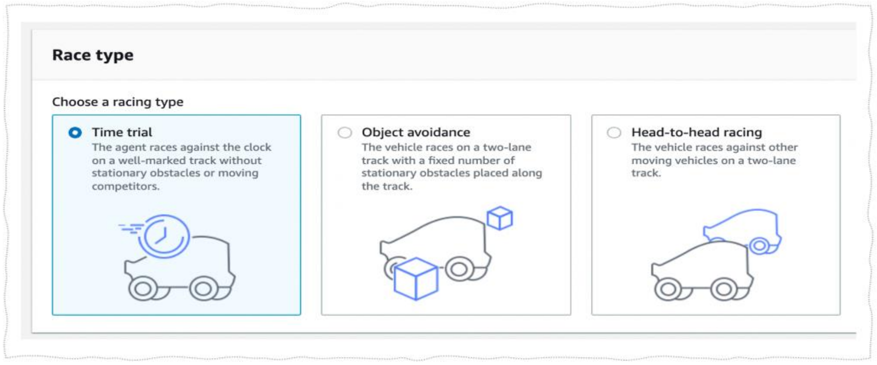
Evaluate the trained model by having the agent race against the clock along a chosen track in simulation.
See how your model performs in any environment.

Training configuration

Environment simulation | Framework

**Step 2:** Choose a track under Evaluation criteria on the Evaluate model page's Evaluate criteria section. You can choose a track to evaluate that you have used while training a model. You can evaluate your model on any track. However, selecting the most similar track to the one used in training will yield the best results.

**Step 3:** Choose the racing type you used to train the model under Race type on the Evaluate model page.


Race type

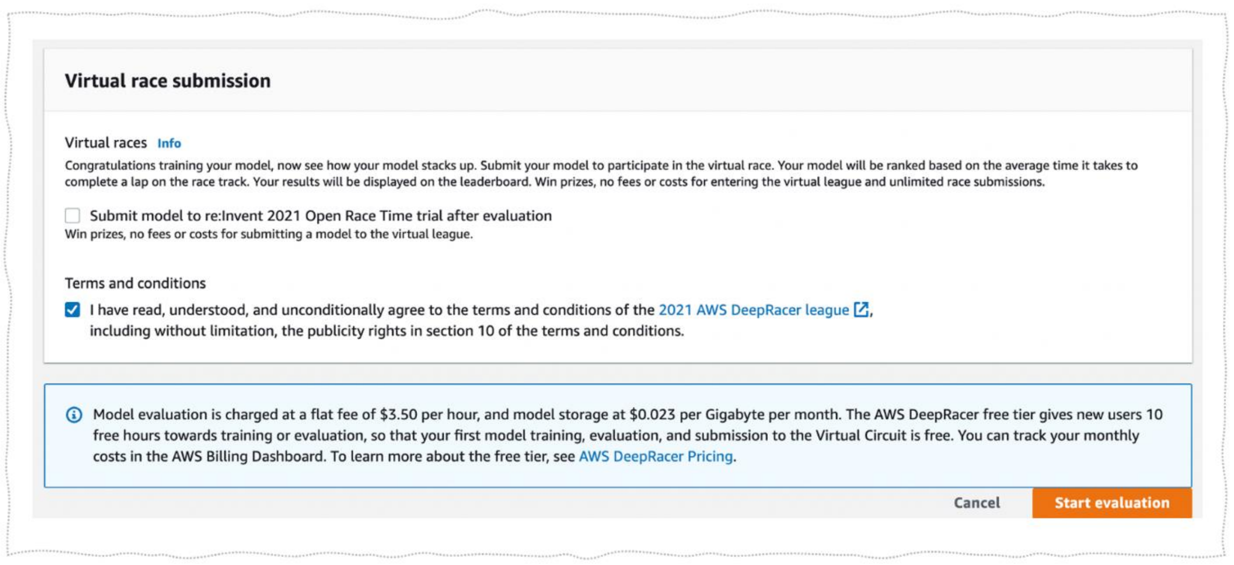
Choose a racing type

[✓ 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.

**Step 4:** Turn off the Submit model after evaluation option on the Evaluate model page under Virtual Race Submission for your initial model. Leave this option enabled later if you want to participate in a racing event. Then choose Start evaluation on the Evaluate model page to begin generating and initialising the evaluation job. It takes roughly 3 minutes to complete the startup process.


**Virtual race submission**

Virtual races Info

Congratulations training your model, now see how your model stacks up. Submit your model to participate in the virtual race. Your model will be ranked based on the average time it takes to complete a lap on the race track. Your results will be displayed on the leaderboard. Win prizes, no fees or costs for entering the virtual league and unlimited race submissions.

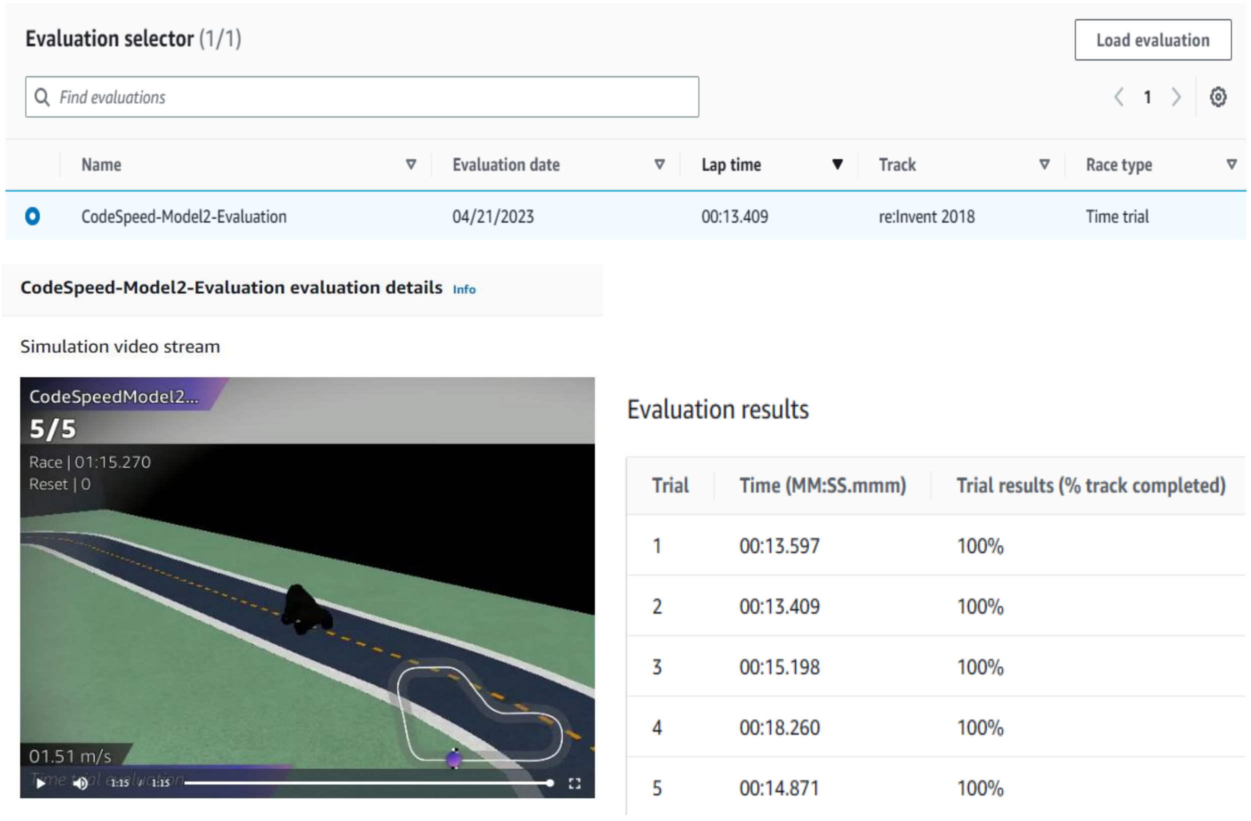
☐ Submit model to re:Invent 2021 Open Race Time trial after evaluation
Win prizes, no fees or costs for submitting a model to the virtual league.

Terms and conditions

☑ I have read, understood, and unconditionally agree to the terms and conditions of the 2021 AWS DeepRacer league, including without limitation, the publicity rights in section 10 of the terms and conditions.

Model evaluation is charged at a flat fee of $3.50 per hour, and model storage at $0.023 per Gigabyte per month. The AWS DeepRacer free tier gives new users 10 free hours towards training or evaluation, so that your first model training, evaluation, and submission to the Virtual Circuit is free. You can track your monthly costs in the AWS Billing Dashboard. To learn more about the free tier, see AWS DeepRacer Pricing.

**Step 5:** The evaluation outcome, including the trial time and track completion rate, are displayed under "Evaluation" after each trial as the evaluation advances. You may watch how the agent performs on the chosen track in the simulation video stream window.


Evaluation selector (1/1)

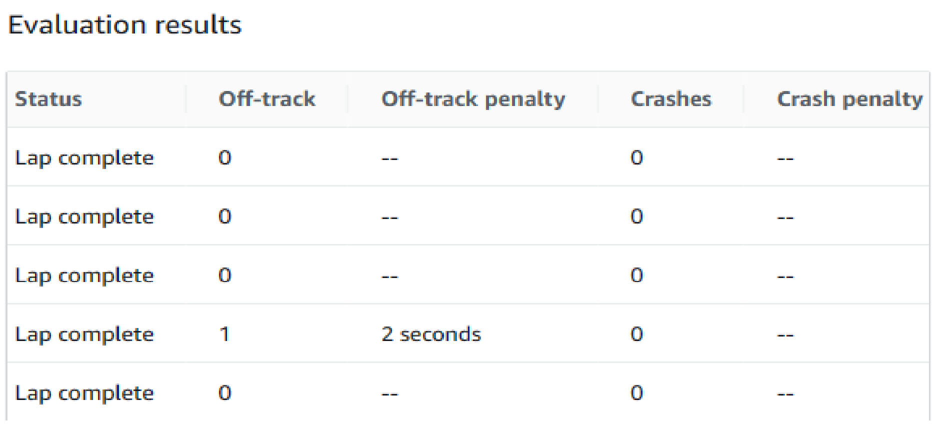
| | Name | Evaluation date | Lap time | Track | Race type |
|---|---|---|---|---|---|
| ● | CodeSpeed-Model2-Evaluation | 04/21/2023 | 00:13.409 | re:Invent 2018 | Time trial |

**CodeSpeed-Model2-Evaluation evaluation details** Info

Simulation video stream

Evaluation results

| Trial | Time (MM:SS.mmm) | Trial results (% track completed) |
|---|---|---|
| 1 | 00:13.597 | 100% |
| 2 | 00:13.409 | 100% |
| 3 | 00:15.198 | 100% |
| 4 | 00:18.260 | 100% |
| 5 | 00:14.871 | 100% |


Evaluation results

| Status | Off-track | Off-track penalty | Crashes | Crash penalty |
|---|---|---|---|---|
| Lap complete | 0 | -- | 0 | -- |
| Lap complete | 0 | -- | 0 | -- |
| Lap complete | 0 | -- | 0 | -- |
| Lap complete | 1 | 2 seconds | 0 | -- |
| Lap complete | 0 | -- | 0 | -- |

The track completion during the evaluation has received 100% track completion with five laps per evaluation. Still, the completion time was much higher. The lowest track completion time we have got from this model was 13.409 seconds. In lap 4 the car went off track for the first time and got the off-track penalty of 2 seconds. Luckily there were no crash and crash penalty implied during the 5 laps.

**Improvements**

Based on the results I got, we can make some possible improvements that we can consider to optimize our model:

1. Adjust the Reward Function: We can fine-tune the reward function to provide more rewards for faster completion times. For example, we can add a penalty for staying on the track for too long or a reward for completing the lap in a shorter amount of time. This can encourage the model to take more risks and finish the lap faster.
2. Increase the Entropy: We can try increasing the entropy of the model to encourage exploration and prevent the model from getting stuck in a suboptimal solution. A higher entropy value will result in the model making more random decisions, which can help it find better solutions.
3. Increase the Learning Rate: We can try increasing the learning rate of the model to make it learn faster. However, we must be careful not to set the learning rate too high, as this can cause the model to converge too quickly to suboptimal solutions.
4. Adjust the Action Space: We can try adjusting the action space to allow for more aggressive driving. For example, we can decrease the speed range or can adjust steering angle range to allow the model to take sharper turns or drive faster on straightaways.
5. Add additional Sensors: We can try adding additional sensors to the car, such as lidar or radar, to provide more accurate information about the environment. This can help the model make more informed decisions and drive more efficiently.
6. Increase the number of iterations: We can try increasing the number of iterations to allow the model to train for longer and find better solutions. However, we have to be careful not to overfit the model to the training data, as this can cause the model to perform poorly on new data.

## Comments and Conclusion

Based on the results, it seems that the trained model was able to perform reasonably well, achieving an average best reward of 441.1 during training and an average best completion rate of 62.15%. During evaluation, the model achieved an average best completion rate of 88.4% and was able to complete the track in 13.409 seconds at its fastest.

It is worth noting that the model went off-track during lap 4 and received a penalty of 2 seconds, indicating that there is room for improvement in terms of the model's ability to stay on track consistently.

Overall, the results suggests that the reward function and hyperparameters used in training were effective in training a model that can complete the track with reasonable performance. However, there is an opportunity for further optimization and refinement of the model to improve its overall performance and consistency.

## References

**[1]** https://aws.amazon.com/deepracer/
**[2]** https://aws.amazon.com/deepracer/league/
**[3]** https://docs.aws.amazon.com/deepracer/latest/developerguide/what-is-deepracer.html
**[4]** https://refactored.ai/microcourse/notebook?path=content%2FDeepRacer%2FAWS_DeepRacer_Reward_function_Additional_material.ipynb